第十章 批流融合系统Flink

徐 辰
cxu@dase.ecnu.edu.cn
華東師範大學

Flink简介

2

- 2008年
  - 德国科学基金会（DFG）资助Stratosphere研究项目
    - 负责人为柏林工业大学Volker Markl教授
    - 成员单位包括柏林洪堡大学、波茨坦大学HPI等
  - 批处理系统，扩充MapReduce算子，引入流水线方式进行数据传输

Flink简介

3

- Stratosphere: MapReduce & Database++

Taken from database Technology		Taken from MapReduce technology
• Declarativity • Query optimization • Robust out-of-core	• Iterations • Advanced DAG • General APIs	• Scalability • User-defined functions • Complex data types • Schema on read

The Stratosphere platform for big data analytics. VLDB Journal, 2014.

Flink简介

4

- 2014年
  - 德国联邦教研部资助柏林大数据中心（Berlin Big Data Center）研究项目
    - Speaker：Volker Markl
    - 柏林工业大学、德国人工智能研究中心（DFKI）等
  - Apache孵化项目Flink，开始支持流计算
  - 成立创业公司DataArtisan，后改名Ververica

Flink简介

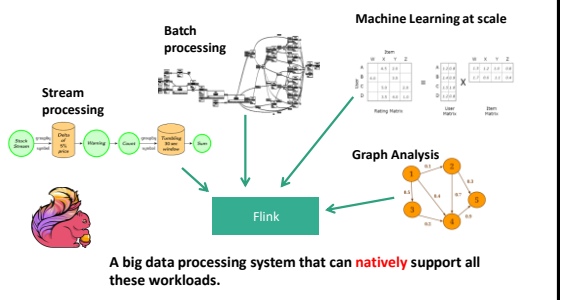
5

- 2015年
  - 谷歌发表Dataflow模型的论文
  - Flink逐步定位为批流一体化的执行引擎并且支持Dataflow模型中定义的批流融合操作
- 2019年
  - 阿里巴巴集团收购Ververica

The Dataflow Model: A Practical Approach to Balancing Correctness, Latency, and Cost in Massive-Scale, Unbounded, Out-of-Order Data Processing. *PVLDB 2015.*

Flink简介

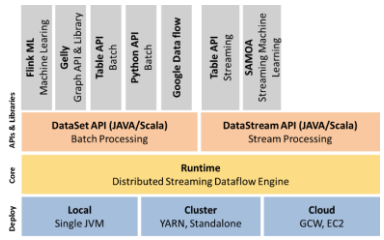
6



A big data processing system that can natively support all these workloads.

Flink简介

7



大纲

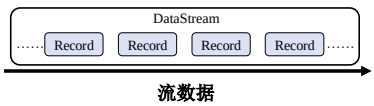
8

- 设计思想
  - 数据模型
  - 计算模型
  - 迭代模型
- 体系架构
- 工作原理
- 容错机制
- 编程示例

数据模型

9

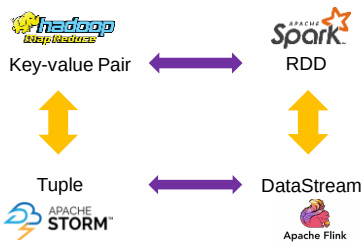
- 与Storm类似，Flink将输入数据看作是一个不间断的、无界的连续记录序列
- 有所不同的是，Flink将这一系列的记录抽象成DataStream
  - 类似于RDD，DataStream是不可变的



流数据

数据模型的比较

10



大纲

11

- 设计思想
  - 数据模型
  - 计算模型
  - 迭代模型
- 体系架构
- 工作原理
- 容错机制
- 编程示例

DataStream操作算子

12

- 操作算子对DataStream进行变换
  - DataStream经变换操作得到新的DataStream
  - 与Spark中RDD经变换生成新的RDD类似
- 一系列的变换操作构成一张有向无环图，即描述计算过程的DAG
  - 数据源（DataSource）
  - 转换（Transformation）
  - 数据池（DataSink）

DataSource

13

- 描述DataStream数据的来源

操作算子	含义
fromElements(elements)	从相同类型的记录创建DataStream
fromCollection(collection)	从内存集合创建DataStream
readTextFile(path)	逐行读取文件内容来创建DataStream
socketTextStream(hostname, port)	接收来自套接字的内容来创建DataStream
addSource(customer-source-func)	使用用户自定义source func来创建DataStream

Transformation

14

- 描述DataStream在系统中的转换逻辑

操作算子	含义
map(func)	对DataStream中的每个记录都使用func进行转化，返回一个新的DataStream
filter(func)	过滤出对DataStream中的每个记录都使用func后返回值为true的记录
flatMap(func)	与map类似，但是对DataStream中的每个记录可以映射成0个或多个新的记录
union(otherDataStreams)	若干个数据类型相同的DataStream取并集得到一个新的DataStream
connect(otherDataStream)	两个数据类型可能不同的DataStream取并集得到一个新的DataStream
keyBy(key)	以指定的key划分DataStream来创建KeyedStream
window(WindowAssigner)	对KeyedStream中按键按分组的记录根据WindowAssigner将其划分为多个窗口，返回一个WindowedStream
windowAll(WindowAssigner)	对DataStream中的记录根据WindowAssigner划分为多个窗口，返回一个AllWindowedStream
reduce(func)	通过使用func简化DataStream中的记录，返回一个新DataStream
aggregate(func)	对DataStream中每个窗口中记录使用func聚合为结果记录，返回一个新DataStream
join(otherDataStream)	$[K, V1]$和$[K, V2]$分别属于两个DataStream，返回一个$[K, (V1, V2)]$组成的JoinedStream
coGroup(otherDataStream)	$[K, V1]$和$[K, V2]$分别属于两个DataStream，返回一个$[K, [Iterable < V1 >, Iterable < V2 >]]$组成的CoGroupedStream

DataSink

15

- 描述DataStream数据的走向，标志着计算DAG的结束

操作算子	含义
print()	将DataStream写入标准输出
writeAsText(path)	将DataStream以文本格式写入指定路径的文件中
writeToSocket(hostname, port, schema)	将DataStream作为字节数组写入套接字，输出格式由schema指定
addSink(customer-sink-func)	使用用户自定义sink func作为数据池操作

逻辑计算模型

16

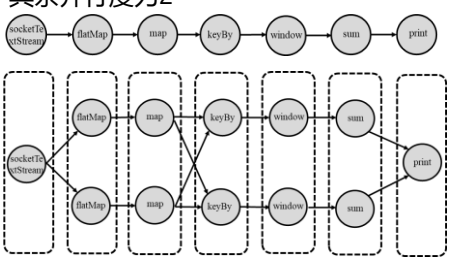
- 一系列操作算子构成DAG，描述计算过程

- 该例较为简单，图中每个顶点的出度最多为1，但Flink支持以更为复杂的有向无环图来描述逻辑计算模型

物理计算模型

17

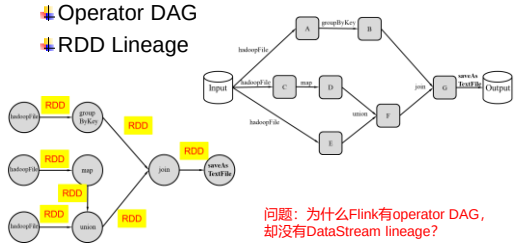
- 假定socketTextStream和print并行度为1，其余并行度为2



回顾：Spark逻辑计算模型

18

- 对于spark，有两种方式描述逻辑计算模型
  - Operator DAG
  - RDD Lineage



问题：为什么Flink有operator DAG，却没有DataStream lineage?

大纲

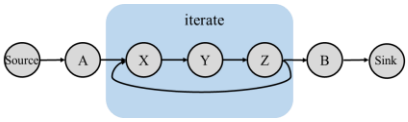
19

- 设计思想
  - 数据模型
  - 计算模型
  - 迭代模型
- 体系架构
- 工作原理
- 容错机制
- 编程示例

迭代模型

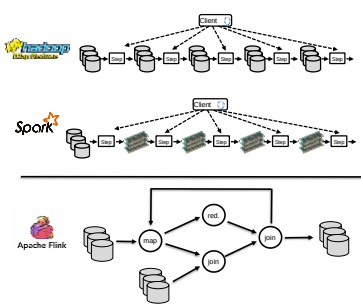
20

- 迭代过程内部必然存在环路
- 将迭代部分整体视为一个算子，计算的过程仍然是DAG（有向无环图）



迭代比较

21



每一迭代步骤(Step)结束时将结果写入HDFS，下一步将该结果再次从HDFS读出迭代次数决定了MapReduce作业的个数

每一迭代步骤 (Step)结束时生成一个RDD，直接作为下一轮迭代的输入，整个迭代可以是一个作业

系统提供迭代算子，用户编程时不需要使用for、while等循环，整个迭代是一个作业

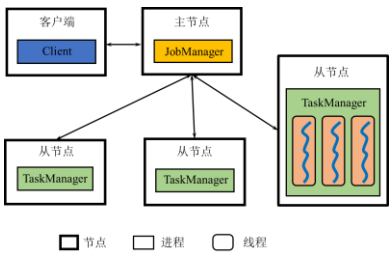
大纲

22

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

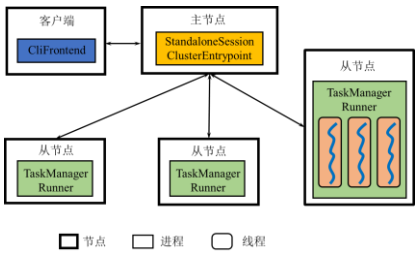
抽象架构图

23



Standalone模式架构图

24



Client

25

- 客户端
  - 将用户编写的DataStream程序翻译为逻辑执行图并进行优化，并将优化后的逻辑执行图提交到JobManager
  - Standalone模式下，Client的进程名为CliFrontend

JobManager

26

- 作业管理器
  - 根据逻辑执行图产生物理执行图，负责协调系统的作业执行，包括任务调度，协调检查点和故障恢复等
  - Standalone模式下
    - JobManager还负责Flink系统的资源管理
    - JobManager的进程名为StandaloneSessionClusterEntrypoint

TaskManager

27

- 任务管理器
  - 用来执行JobManager分配的任务，并且负责读取数据、缓存数据以及与其它TaskManager进行数据传输
  - Standalone模式下
    - TaskManager还负责所在节点的资源管理，将内存等资源抽象成若干个TaskSlot用于任务的执行，
    - TaskManager的进程名为TaskManagerRunner

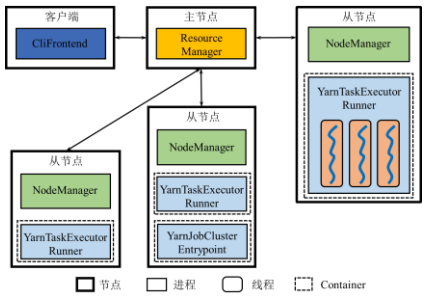
与MapReduce/Spark/Storm比较

28

	MapReduce	Storm	Spark	Flink
系统进程	JobTracker	Nimbus	Master	JobManager
	TaskTracker	Supervisor	Worker	TaskManager
	Child	Worker	CoarseGrained ExecutorBackend	
工作线程	/	Executor	Task	Task
任务代码	Task	Task		
基础接口	Map/Reduce	Spout/Bolt	RDD API	DataStream API

Yarn模式架构图

29



Standalone vs. Yarn模式

30

模式	Standalone	Yarn
资源管理	StandaloneSessionClusterEntrypoint TaskManagerRunner	ResourceManager
		NodeManager
作业管理		YarnJobClusterEntrypoint
任务执行	TaskManagerRunner	YarnTaskExecutorRunner

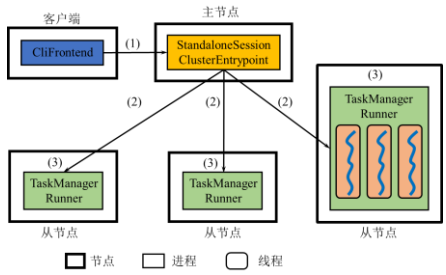
大纲

31

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

Standalone模式应用程序执行流程

32



Standalone模式应用程序执行流程

33

1. 客户端将用户编写的程序进行解析，并将解析后的作业描述交给StandaloneSess-ionClusterEntrypoint
2. StandaloneSessionClusterEntrypoint根据作业的描述进行任务分解，确定各个TaskManagerRunner所负责执行的任务
3. TaskManagerRunner执行所负责的任务

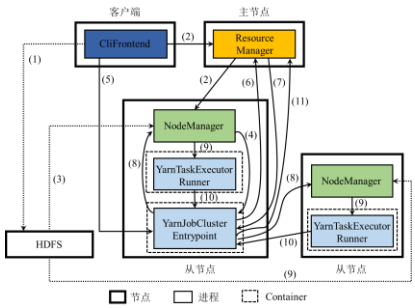
提交方式

34

- 在Standalone模式下，当用户使用客户端提交Flink应用程序时，可以选择Attached方式或者Detached方式
  - Attached提交方式：客户端与JobManager保持连接，可以获取关于应用程序执行的信息
  - Detached提交方式：客户端与JobManager断开连接，无法获得关于应用程序执行的信息

Yarn模式应用程序执行流程

35



Yarn模式应用程序执行流程

36

1. 客户端启动CliFrontend进程，CliFrontend将用户编写的程序进行解析，并将运行Flink系统的jar包以及配置文件上传至HDFS
2. CliFrontend向ResourceManager申请启动YarnJobClusterEntrypoint (ApplicationMaster)，ResourceManager确定启动YarnJobClusterEntrypoint的节点
3. 需启动YarnJobClusterEntrypoint进程的节点上的NodeManager将HDFS中的jar包与配置文件下载到该节点
4. NodeManager启动YarnJobClusterEntrypoint进程
5. CliFrontend进程将解析后的作业描述交给YarnJobClusterEntrypoint
6. YarnJobClusterEntrypoint向ResourceManager注册，这样客户端可以通过 ResourceManager查看Flink应用程序的资源使用情况。YarnJobClusterEntrypoint根据作业的描述进行任务分解，并向ResourceManager申请启动这些任务的资源

Yarn模式应用程序执行流程

37

7. ResourceManager以Container形式向提出申请的YarnJobClusterEntrypoint分配资源。得到资源后，它在多个任务间进行资源分配
8. YarnJobClusterEntrypoint确定资源分配方案后，便与对应的NodeManager通信
9. 如果该NodeManager所在节点尚未下载，则将HDFS中的jar包与配置文件下载到本地，并在相应的Container中启动相应的YarnTaskExecutorRunner进程用于执行任务
10. 各个任务向YarnJobClusterEntrypoint汇报自己的状态和进度，以便让 YarnJobClusterEntrypoint随时掌握各个任务的运行状态
11. 随着部分任务执行结束，YarnJobClusterEntrypoint逐步释放所占用的资源，最终向ResourceManager注销并关闭自己

提交方式

38

- 在Yarn模式下，用户也可以选择Attached方式或者Detached方式提交应用程序
  - Attached提交方式：CliFrontend将与YarnJobClusterEntrypoint保持连接，可以获取关于应用程序执行的信息
  - Detached提交方式：CliFrontend将与YarnJobClusterEntrypoint断开连接，无法获得关于应用程序执行的信息

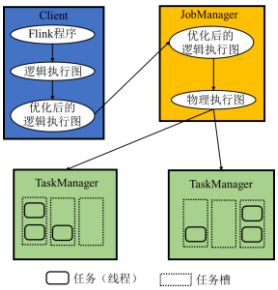
大纲

39

- 设计思想
- 体系架构
- 工作原理
- 容错机制
- 编程示例

内部工作过程

40



大纲

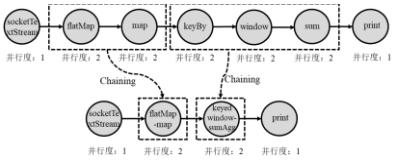
41

- 设计思想
- 体系架构
- 工作原理
  - 逻辑执行图的生成与优化
  - 物理执行图的生成与任务分配
  - 非迭代任务间的数据传输
  - 迭代任务内部的数据传输
- 容错机制
- 编程示例

逻辑执行图与Chaining优化

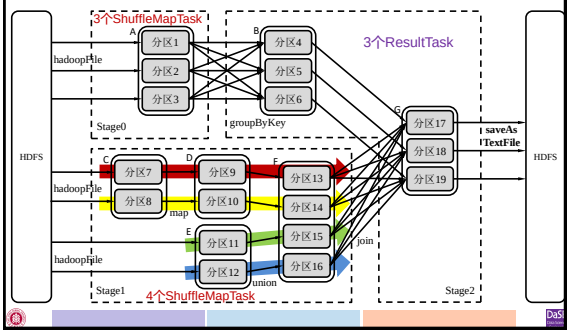
42

- 给定用户编写的DataStream程序，Flink的Client将其解析产生逻辑执行图，即DAG
- Chaining优化：将“窄依赖”算子合并起来形成一个“大”算子



回顾：Spark Pipeline

43



大纲

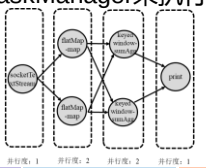
44

- 设计思想
- 体系架构
- 工作原理
  - 逻辑执行图的生成与优化
  - 物理执行图的生成与任务分配
  - 非迭代任务间的数据传输
  - 迭代任务内部的数据传输
- 容错机制
- 编程示例

物理执行图的生成

45

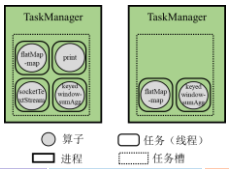
- JobManager收到Client提交的逻辑执行图之后，根据算子的并行度，将逻辑执行图转换为物理执行图
- 物理执行图中的一个结点对应一个任务，将分配给TaskManager来执行



任务分配

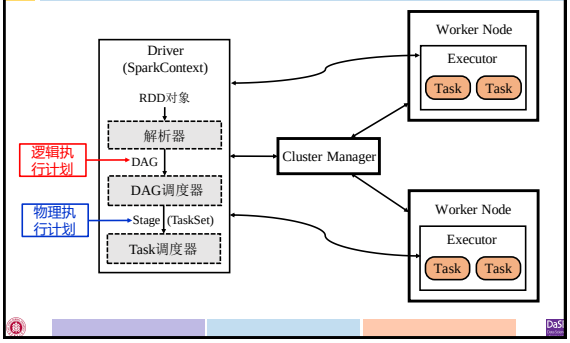
46

- JobManager将各算子的任务分配给TaskManager
  - 根据任务槽(TaskSlot)的容量，尽可能将存在数据传输关系的算子实例放在同一个任务槽，保持数据传输的本地性



回顾：Spark Driver内部工作过程

47



大纲

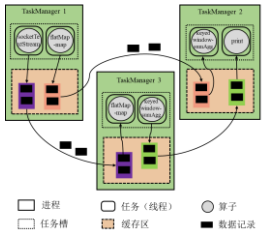
48

- 设计思想
- 体系架构
- 工作原理
  - 逻辑执行图的生成与优化
  - 物理执行图的生成与任务分配
  - 非迭代任务间的数据传输
  - 迭代任务内部的数据传输
- 容错机制
- 编程示例

流水线机制

49

- 上游的Task将数据存在buffer中，一旦buffer满了或者超时，就向下游Task发送



Pipeline in Flink vs. Spark

50

- Flink中的Pipeline
  - 不同Task之间的数据传输方式
- Spark中的Pipeline
  - Stage内部同一个Task实现多个不同算子间的数据传输方式
- Spark Pipeline和Flink Chaining类似

Task间的数据传输方式

51

- 阻塞式数据传输
  - 一个Task（运行某个或某些算子）将所有需要处理的数据计算完，甚至要将结果写入磁盘，才会发送给位于下游Task或被其读取
- 非阻塞式数据传输
  - 一个Task处理一条或一部分数据，通常将计算结果放在缓存里，就会发送给位于下游Task或被其读取

不同算子任务间数据传输方式比较

52

系统	阻塞式数据传输	非阻塞式数据传输
Hadoop MapReduce	√	
Spark	√	
Apache Storm		√
Apache Flink		√

大纲

53

- 设计思想
- 体系架构
- 工作原理
  - 逻辑执行图的生成与优化
  - 物理执行图的生成与任务分配
  - 非迭代任务间的数据传输
  - 迭代任务内部的数据传输
- 容错机制
- 编程示例

迭代的实现

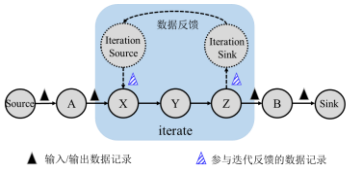
54

- 迭代算子
  - 嵌套在描述计算过程DAG中的一个整体
  - 迭代算子内部存在数据反馈的环路
- 数据反馈的实现
  - 迭代前端（Iteration Source）和迭代末端（Iteration Sink）两类特殊的任务
  - 两类任务成对处于同一个TaskManager，迭代末端任务的输出可以再次作为迭代前端任务的输入

流式迭代的特征

55

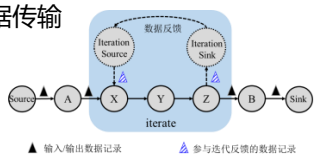
- 在流式迭代计算中，通常每一轮迭代计算的部分结果作为输出向后传递，而另一部分结果作为下一轮迭代计算的输入，并且迭代过程会一直进行下去



流式迭代的数据传输

56

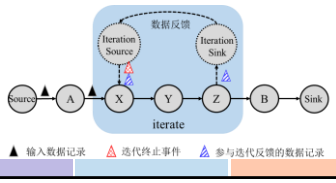
- 流式迭代计算中，迭代前端下一轮的计算并不依赖于迭代末端在前一轮迭代得到的所有记录
- 迭代末端收到迭代前端的反馈后可以立即进行新一轮迭代计算，因而采用流水线方式进行数据传输



批式迭代的特征

57

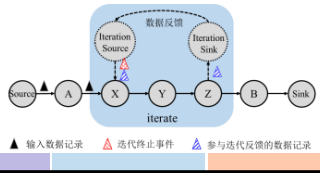
- 在批式迭代计算中，每一轮迭代计算的全部结果通常都是下一轮迭代计算的输入，直到迭代过程在满足收敛条件时停止迭代
- 迭代前端中发出特殊的控制事件（control event），即特殊的记录，表示迭代计算的结束



批式迭代的数据传输

58

- 批式迭代计算中，迭代末端必须收到迭代前端反馈的所有记录后才可以开始新一轮迭代计算
- 迭代前端存在等待迭代末端反馈所有记录的阻塞过程，无法采用流水线机制进行数据传输



大纲

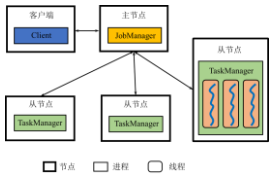
59

- 设计思想
- 体系架构
- 工作原理
- 容错机制
- 编程示例

故障类型

60

- JobManager故障
  - 怎么办?
- TaskManager故障
  - 运行了非迭代Task
  - 运行了迭代Task



大纲

61

- 设计思想
- 体系架构
- 工作原理
- 容错机制
  - 状态管理
  - 非迭代计算过程的容错
  - 迭代计算过程的容错
- 编程示例

什么是状态？

62

- 在输入数据是无界的场景中，数据会源源不断地流入Flink系统
- 例如，在某一窗口中统计单词的个数，
  - 窗口需要将原始的单词记录保存起来直到窗口触发时一并进行统计
  - 或者将单词以及当前观察的的个数保存起来并逐步累加
  - 窗口这个算子所需维护的内容就是状态
- 注意区分算子的状态与进程/节点的状态

为什么需要管理状态？

63

- 用户程序管理状态
  - 用户编写一个HashMap，记录计数
    - 一旦该算子所在task发生故障，内存中的HashMap就丢失了☹
  - 为了支持容错，需要编写程序将HashMap写入磁盘等可靠的存储设备，故障恢复后读取
    - 不同数据结构都需要编写相应的保存、读取代码☹
- 状态管理应该交给系统，而不是用户程序

状态管理

64

- 状态：系统定义的特殊的数据结构，用于记录需要保存的算子计算结果
  - ValueState<T>：状态保存的是每个key的一个值，可以通过update(T)来更新，T.value()获取
  - ListState<T>：状态保存的是每个key的一个列表，通过add(T)添加数据，Iterable.get()获取
  - ReducingState<T>：状态保存的是关于每个key的经过聚合之后的值列表，通过add(T)添加数据，通过指定的聚合方法获取
  - ……

有状态算子 vs. 无状态算子

65

- 有状态算子：具备记忆能力的算子
  - 可以保留已经处理记录的结果，并对后续记录的处理造成影响
  - 例如，Window、Sum
- 无状态算子：不具备记忆能力的算子
  - 只考虑到当前处理的记录，不会受到已处理记录的影响，也不会影响到后续待处理的记录
  - 例如，Map

状态管理与容错

66

- 算子级别的容错
  - 运行时保存其状态，在发生故障时重置状态，并继续处理结果尚未保存到状态当中的记录
- DAG级别的容错
  - 如果我们可以在“同一时刻”将所有算子的状态保存起来形成检查点，一旦出现故障则所有算子都根据检查点重置状态，并处理尚未保存到检查点中的记录
  - 要求所有节点的物理时钟绝对同步

绝对的时钟同步是不可能的

大纲

67

- 设计思想
- 体系架构
- 工作原理
- 容错机制
  - 状态管理
  - 非迭代计算过程的容错
  - 迭代计算过程的容错
- 编程示例

系统中的记录

68

- 在某一时刻，流计算系统所处理的记录可以分为三种类型
  - 已经处理完毕的记录，即所有算子都已经处理了这些记录
  - 正在处理的记录，即部分算子处理了这些记录
  - 尚未处理的记录，即没有算子处理过这些记录
- 虽然绝对同步的时钟是不存在的，但是同一时刻保存所有算子状态到检查点的目的是区分第一种记录与后两种记录

异步屏障快照

69

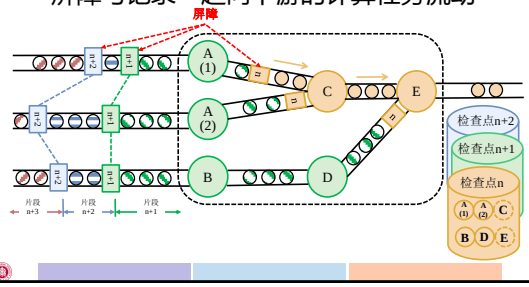
- Chandy-Lamport算法：分布式系统中用于保存系统状态

- 异步屏障快照（Asynchronous Barrier Snapshotting）算法
  - 所保存的快照就是检查点
  - 通过在输入数据中注入屏障，并异步地保存快照，达到和在同一时刻保存所有算子状态到检查点相同的目的

屏障

70

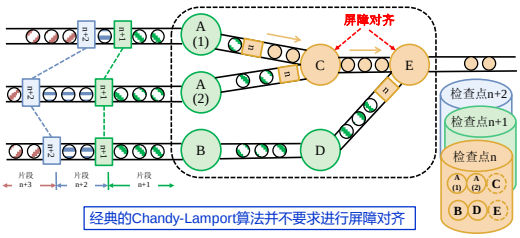
- JobManager在输入记录中插入屏障，这些屏障与记录一起向下游的计算任务流动



屏障对齐

71

- 一个任务需收到来自上游任务中所有标识为n的屏障后才将其状态保存起来

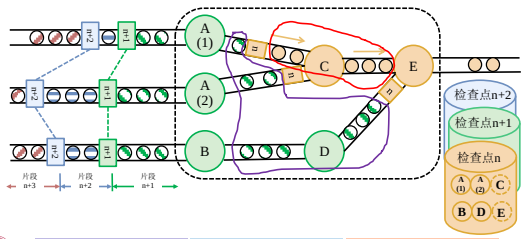


经典的Chandy-Lamport算法并不要求进行屏障对齐

异步

72

- 某一任务将标识为n的屏障对齐后，可继续接收属于检查点n+1的数据



Flink状态存储

73

状态存储方式	正常运行时	写检查点时
MemoryState Backend	本地内存	JobManager内存
FsStateBackend	本地内存	HDFS
RocksDBState Backend	本地RocksDB	HDFS

故障恢复

74

- 当发生故障时
  - Flink选择最近完整的检查点n，将系统中每个算子的状态重置为检查点中保存的状态
  - 从数据源重新读取属于屏障n之后的记录
    - 这要求数据源具备一定的记忆功能
    - 例如，Flink从Kafka中重新读取屏障n对应偏移量之后的记录
- Flink的容错机制能够满足准确一次的容错语义

大纲

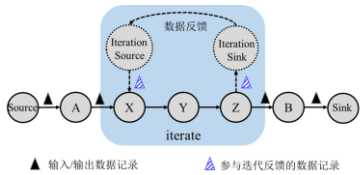
75

- 设计思想
- 体系架构
- 工作原理
- 容错机制
  - 状态管理
  - 非迭代计算过程的容错
  - 迭代计算过程的容错
- 编程示例

异步屏障快照与迭代

76

- 迭代反馈的数据和输入数据将继续进行新的计算，因而在这种情况下仅有屏障无法区分属于检查点n和检查点n+1的记录



Chandy-Lamport算法与迭代

77

- 根据Chandy-Lamport算法，反馈环路中的所有记录需要以日志形式保存起来
- 当故障发生后，系统
  - 需要根据最近完整的检查点n重置各个算子的状态
  - 还需要重新读取属于屏障n之后的记录以及日志中的记录

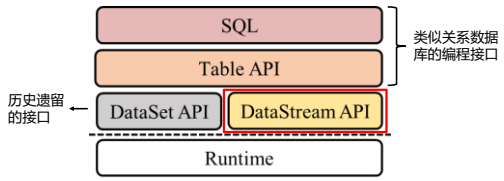
大纲

78

- 设计思想
- 体系架构
- 工作原理
- 容错机制
- 编程示例

Flink API

79



DataStream API编程框架

80

```
import org.apache.flink.streaming.api.scala.StreamExecutionEnvironment
......
object CustomApp {
  def run(args: Array[String]) = {
    /* 步骤1：  创建StreamExecutionEnvironment对象 */
    val env = StreamExecutionEnvironment.getExecutionEnvironment

    /* 步骤2：按应用逻辑使用操作算子编写DAG，操作算子包括数据源、转换和数据池等 */
    ......

    /* 步骤3：触发程序执行 */
    env.execute("jobName")
  }

  def main(args: Array[String]) = {
    run(args)
  }
}
```

数据源: socketTextStream(hostname, port)
数据池: print()

大纲

81

- 设计思想
- 体系架构
- 工作原理
- 容错机制
- 编程示例
  - 词频统计
  - 斐波那契数列
  - 支持容错的词频统计

词频统计

82

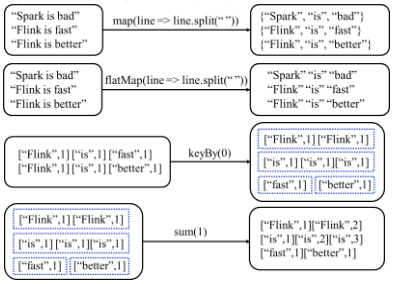
- 输入：来自套接字的文本
- 输出：
  - 每个单词及其出现次数（频数）
  - 每个单词和其频数占一行，单词和频数之间以空格分隔

输入	输出
An An	An 1 An 2
My Me	My 1 Me 1
An An	An 3 An 4
My He	My 2 He 1
My My	My 3 My 4
An My	An 5 My 5
...	...

转换算子回顾

83

- map、flatMap、 keyBy 、sum



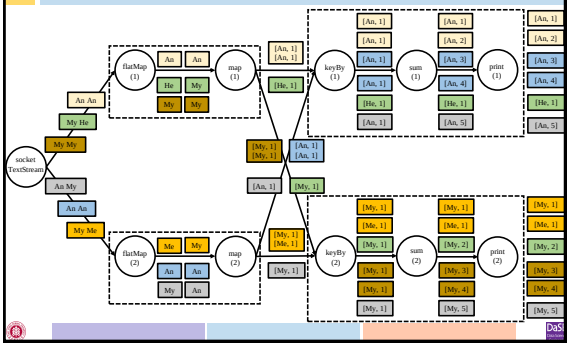
解决方案

84

- 分词过程
  - 通过flatMap将每行文本按分隔符拆分成单词
  - 通过map将每个单词映射为[单词, 1]键值对
- 计数过程
  - 通过keyBy将[单词, 1]键值对按单词进行分组
  - 通过sum对组内每个单词的频数进行求和
  - 通过print输出计数结果

运行过程

85



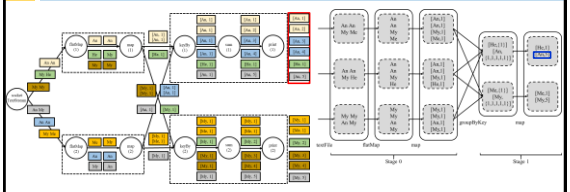
代码

86

```
object WordCount {
  def run(args: Array[String]): Unit = {
    /* 步骤1: 创建StreamExecutionEnvironment对象 */
    val env = StreamExecutionEnvironment.getExecutionEnvironment
    env.setParallelism(2)

    /* 步骤2: 按应用逻辑使用操作算子编写DAG, 操作算子包括数据源、转换和数据池等 */
    // 从指定的主机名和端口号接收文本数据, 创建名为lines的DataStream
    val lines = env.socketTextStream("hostname", port).setParallelism(1)
    // 将lines中的每一个文本行按空格分割成单个单词
    val words = lines.flatMap(w => w.split(" "))
    // 将每个单词的频数设置为1, 即将每个单词映射为(单词, 1)
    val pairs = words.map(word => (word, 1))
    // 按单词聚合, 并对相同单词的频数使用sum进行累计
    val counts = pairs.keyBy(0).sum(1)
    // 输出词频统计结果
    counts.print()

    /* 步骤3: 触发程序执行 */
    env.execute("WordCount")
  }

  def main(args: Array[String]) = {
    run(args)
  }
}
```

对比

87



Flink中词频统计的运行过程　　Spark中词频统计的运行过程

Flink：每处理一条数据更新一次输出结果，直到数据源停止输入
Spark：直接输出最终结果

大纲

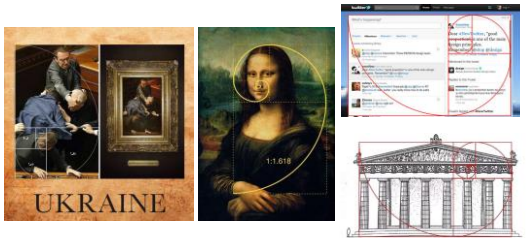
88

- 设计思想
- 体系架构
- 工作原理
- 容错机制
- 编程示例
  - 词频统计
  - 斐波那契数列
  - 支持容错的词频统计

应用

89

- 摄影、绘画、设计



斐波那契数列

90

- 给定的两个数字来计算后续的斐波那契数
  - 输入：
    - 来自套接字的字符标识符以及斐波那契数列的前两个数字
    - 字符标识符用来区分不同的数列
  - 输出：字符标识符和斐波那契数组成的元组

输入	输出
A 1 2	(A, 1) (A, 2) (A, 3) …
B 6 7	(B, 6) (B, 7) (B, 13) …

斐波那契数列

91

- 算法执行过程

迭代1 $F_3 \leftarrow F_2 + F_1$: 3 ← 2 + 1 →

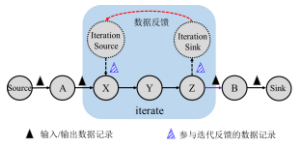
迭代2 $F_4 \leftarrow F_3 + F_2$: 5 ← 3 + 2 →

迭代3 $F_5 \leftarrow F_4 + F_3$: 8 ← 5 + 3 →

...

迭代n $F_n \leftarrow F_{n-1} + F_{n-2}$ →

...



将F_n和F_{n-1}作为下一轮迭代的输入

将F_{n-2}作为输出向后传递

解决方案

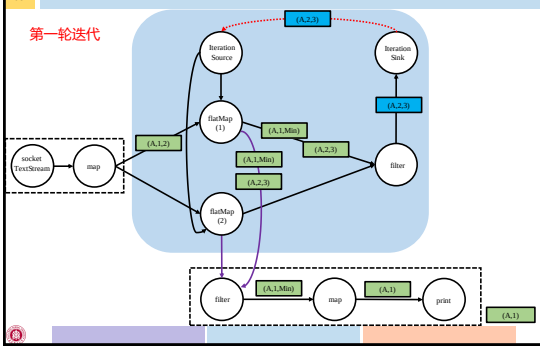
92

- 一轮迭代的处理逻辑
  - 根据F_{n-1}和F_{n-2}计算出F_n，并为F_{n-2}打上标记
    - flatMap
  - 将F_{n-2}作为输出向后传递
    - filter、map
  - 将F_n和F_{n-1}作为输入反馈给迭代前端
    - filter

前三轮迭代

93

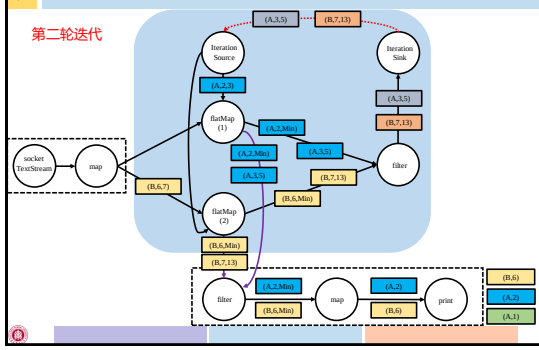
第一轮迭代



前三轮迭代

94

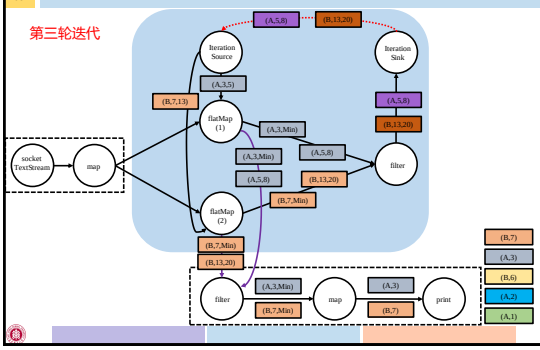
第二轮迭代



前三轮迭代

95

第三轮迭代



代码

96

```
object FibonacciExample {
  def run(args: Array[String]): Unit = {
    /* 步骤1：  创建StreamExecutionEnvironment对象 */
    val env = StreamExecutionEnvironment.getExecutionEnvironment

    /* 步骤2：  按应用逻辑使用操作算子编写DAG，操作算子包括数据源、转换和数据落等 */
    // 接收来自Socket数据，创建名为inputStream的DataStream
    val inputStream = env.socketTextStream("localhost", 9999)
    // 解析inputStream中的数据，创建名为first的DataStream
    val first: DataStream[(Char, Long, Long)] = inputStream.map(lines => (lines.split("
      ")(0).charAt(0),lines.split(" ")(1).toLong, lines.split(" ")(2).toLong))

    val outputStream = inputStream
    // 创建迭代算子
    .iterate(
      (iteration: DataStream[(Char, Long, Long)]) => {
        // 实现迭代步逻辑，计算下一个斐波那契数
        val step = iteration.flatMap(in => {
          // 例如，迭代算子的输入为(a, 1, 2)，此处将转换(a, 1, 2)转换为(a, 2, 3)
          val feedbackValue = (in._1, in._3, in._2 + in._3)
          // 例如，迭代算子的输入为(a, 1, 2)，此处将转换(a, 1, 2)转换为(a, 1, Min)
          val outputValue = (in._1, in._3, Long.MinValue)
          val list = feedbackValue :: outputValue :: Nil
          list.toIterator
        }).setParallelism(2)
        // 创建反馈流，选择第三位置不为Min的元组，例如：(a, 2, 3)
        val feedback = step.filter(_._3 != Long.MinValue)
        // 创建输出流，选择第三位置为Min的元组，例如：(a, 1, 3)，并将其转换为(a, 1)
        val output = step.filter(_._3 == Long.MinValue).map(in => (in._1, in._2))
        (feedback, output)
      }
      // 设置等待反馈输入的最大时间间隔为1ms
      , 1000L
    )
    // 输出迭代计算结果
    outputStream.print()

    /* 步骤3：触发程序执行 */
    env.execute("Streaming Iteration")
  }

  def main(args: Array[String]) = {
    run(args)
  }
}
```

大纲

97

- 设计思想
- 体系架构
- 工作原理
- 容错机制
- 编程示例
  - 词频统计
  - 斐波那契数列
  - 支持容错的词频统计

支持容错的词频统计

98

- 输入：来自套接字的文本
- 输出：
  - 每个单词及其出现次数（频数）
  - 每个单词和其频数占一行，单词和频数之间以空格分隔
- 要求：启用容错机制，期望每秒保存一个检查点

输入	输出
An An	An 1 An 2
My Me	My 1 Me 1
An An	An 3 An 4
My He	My 2 He 1
My My	My 3 My 4
An My	An 5 My 5
…	…

解决方案

99

- 启用检查点机制
  - enableCheckpointing(interval)
- 设置检查点的最大并发数
  - setMaxConcurrentCheckpoints(maxConcurrentCheckpoints)
- 设置状态存储的方式
  - setStateBackend(backend)

代码

100

```
object WordCountWithFaultTolerance {
  def run(args: Array[String]): Unit = {
    /* 步骤1：创建StreamExecutionEnvironment对象 */
    val env = StreamExecutionEnvironment.getExecutionEnvironment
    // 设置checkpoint的周期，每隔1000ms试图启动一个检查点
    env.enableCheckpointing(1000)
    // 设置检查点的最大并发数
    env.getCheckpointConfig.setMaxConcurrentCheckpoints(Long.MAX_VALUE)
    // 设置statebackend，使用FsStateBackend将写检查点时的状态存储至HDFS
    env.setStateBackend(new FsStateBackend("hdfs://hadoop:9000/flink/checkpoints"))
    // 处理程序被cancel后，会保留checkpoint数据
    // env.getCheckpointConfig.enableExternalizedCheckpoints(CheckpointConfig
    //   .ExternalizedCheckpointCleanup.RETAIN_ON_CANCELLATION)

    /* 步骤2：按应用逻辑使用操作算子编写DAG，操作算子包括数据源、转换和数据池等 */
    val lines=env.socketTextStream("hostname",port)
    val words = lines.flatMap(w => w.split(" "))
    val pairs = words.map(word => (word, 1))
    val counts = pairs.keyBy(0).sum(1)
    counts.print()

    /* 步骤3：触发程序执行 */
    env.execute("WordCount")
  }

  def main(args: Array[String]) = {
    run(args)
  }
}
```

课后阅读

101

- 论文
  - Carbone, P., Ewen, S., Haridi, S., Katsifodimos, A., Markl, V., & Tzoumas, K. (2015). Apache Flink: Unified Stream and Batch Processing in a Single Engine. IEEE Data Eng. Bull., 38(4), 28–38.
  - Carbone, P., Ewen, S., Richter, S., & Gyula, F. (2017). State Management in Apache Flink. PVLDB, 10(20), 1718–1729.

本章小结

102

- 设计思想
- 体系架构
- 工作原理
- 容错机制
- 编程示例