

个人资料



奔跑的小象



访问： 51413次  
积分： 2481  
等级： **BLOG > 5**  
排名： 第10554名

原创： 168篇  
转载： 183篇  
译文： 4篇  
评论： 3条

文章搜索

文章分类

- Java开发 (54)
- Hadoop (43)
- Hive (20)
- Pig (1)
- HBase (20)
- Web前端 (0)
- Python (3)
- R (7)
- 数据库 (13)
- Zookeeper (5)
- kafka (15)
- HDFS (1)
- MapReduce (8)
- flume (18)
- Linux运维管理 (33)
- storm (3)
- 大数据方案 (16)
- CDH (7)
- Impala (2)
- 机器学习 (15)
- 数据源 (1)
- 大数据系列文章 (9)

**【公告】** 博客系统优化升级 [Unity3D学习，离VR开发还有一步](#) [博乐招募开始啦](#) [虚拟现实，一探究竟](#)

## Apache Flink：详细入门

2016-05-23 14:01 398人阅读 评论(0) 收藏 举报

分类：

Flink

Apache Flink是一个面向分布式数据流处理和批量数据处理的开源计算平台，它能够基于同一个Flink运行时（Flink Runtime），提供支持流处理和批处理两种类型应用的功能。现有的开源计算方案，会把流处理和批处理作为两种不同的应用类型，因为它们它们所提供的SLA是完全不相同的：流处理一般需要支持低延迟、Exactly-once保证，而批处理需要支持高吞吐、高效处理，所以在实现的时候通常是分别给出两套实现方法，或者通过一个独立的开源框架来实现其中每一种处理方案。例如，实现批处理的开源方案有MapReduce、Tez、Crunch、Spark，实现流处理的开源方案有Samza、Storm。

Flink在实现流处理和批处理时，与传统的一些方案完全不同，它从另一个视角看待流处理和批处理，将二者统一起来：Flink是完全支持流处理，也就是说作为流处理看待时输入数据流是无界的；批处理被作为一种特殊的流处理，只是它的输入数据流被定义为有界的。基于同一个Flink运行时（Flink Runtime），分别提供了流处理和批处理API，而这两种API也是实现上层面向流处理、批处理类型应用框架的基础。

### 基本特性

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

- Maven (6)
- Spark (21)
- Hue (3)
- 云计算 (1)
- ELK (6)
- JavaScript (1)
- Docker (6)
- kubernetes (1)
- nifi (4)
- Redis (2)
- Cassandra (6)
- Lucene (1)
- Flink (1)
- Mahout (1)
- Sqoop (0)

文章存档

- 2016年08月 (8)
- 2016年07月 (36)
- 2016年06月 (9)
- 2016年05月 (22)
- 2016年04月 (33)

展开

阅读排行

- hbase数据库错误总结 — (2753)
- CDH5.7快速离线安装教i (1927)
- Flume与Elasticsearch整 (883)
- Spark通过Java Web提交 (790)
- 大数据架构师必读：常见 (613)
- namenode启动不起来--5 (533)
- cloudera hue安装及Ooz (495)
- 近200篇机器学习&深度学 (451)
- “医学数据银行”——临床 (420)
- nifi实例集合： (409)

评论排行

- hbase数据库错误总结 — (1)
- 大数据架构师必读：常见 (1)
- SQL on Hadoop TPCDS (1)
- 分布式消息系统Kafka初 (0)
- Zookeeper伪分布式安装 (0)
- 安装oracle 11g R2的时 (0)
- phoenix实战（hadoop2 (0)
- HBase查询引擎——Pho (0)
- 快速理解 Phoenix：SQL (0)
- “=”和“equals()”的区别 (0)

推荐文章

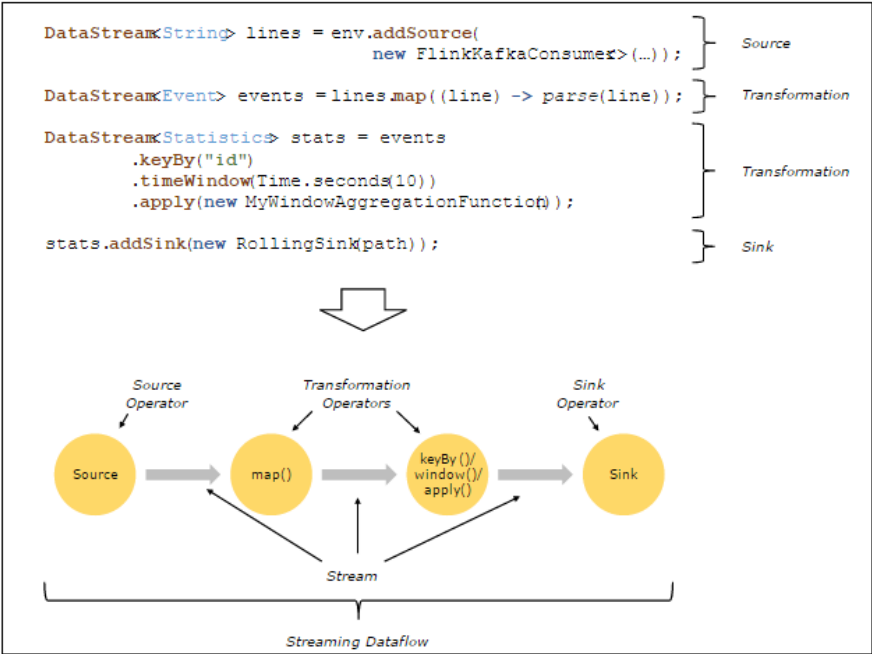
- \* 致JavaScript也将征服的物联网世界
- \* 从苏宁电器到卡巴斯基：难忘的三年硕士时光
- \* 作为一名基层管理者如何利用情商管理自己和团队（一）
- \* Android CircleImageView圆形ImageView
- \* 高质量代码的命名法则

- 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

基本概念  
Stream & Transformation & Operator

用户实现的Flink程序是由Stream和Transformation这两个基本构建块组成，其中Stream是一个中间结果数据，而Transformation是一个操作，它对一个或多个输入Stream进行计算处理，输出一个或多个结果Stream。当一个Flink程序被执行的时候，它会被映射为Streaming Dataflow。一个Streaming Dataflow是由一组Stream和Transformation Operator组成，它类似于一个DAG图，在启动的时候从一个或多个Source Operator开始，结束于一个或多个Sink Operator。

下面是一个由Flink程序映射为Streaming Dataflow的示意图，如下所示：



上图中，FlinkKafkaConsumer是一个Source Operator，map、keyBy、timeWindow、apply是Transformation Operator，RollingSink是一个Sink Operator。

Parallel Dataflow

在Flink中，程序天生是并行和分布式的：一个Stream可以被分成多个Stream分区（Stream Partitions），一个Operator可以被分成多个Operator Subtask，每一个Operator Subtask是在不同的线程中独立执行的。一个Operator的并行度，等于Operator Subtask的个数，一个Stream的并行度总是等于生成它的Operator的并行度。

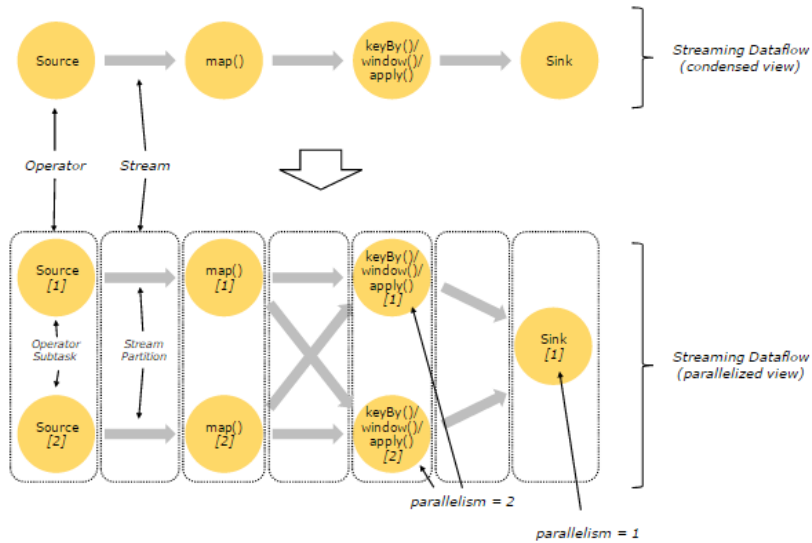
有关Parallel Dataflow的实例，如下图所示：

## 最新评论

SQL on Hadoop TPCDS性能测试  
qq363960630: 博主在不, 我想请教一些关于 hive tpcds测试的问题, 方便加一下我的QQ吗? 我的名字就是我的Q...

hbase数据库错误总结 ——ERR  
sinat\_35371268: 我的还是不行, 原来的没解决又有了新问题!

大数据架构师必读: 常见的七种H  
zhuoyr: mark, 做了很久的“往有更多的HBase, 定制非SQL代码”



上图Streaming Dataflow的并行视图中, 展现了在两个Operator之间的Stream的两种模式:

- One-to-one模式

比如从Source[1]到map()[1], 它保持了Source的分区特性 (Partitioning) 和分区内元素处理的有序性, 也就是说map()[1]的Subtask看到数据流中记录的顺序, 与Source[1]中看到的记录顺序是一致的。

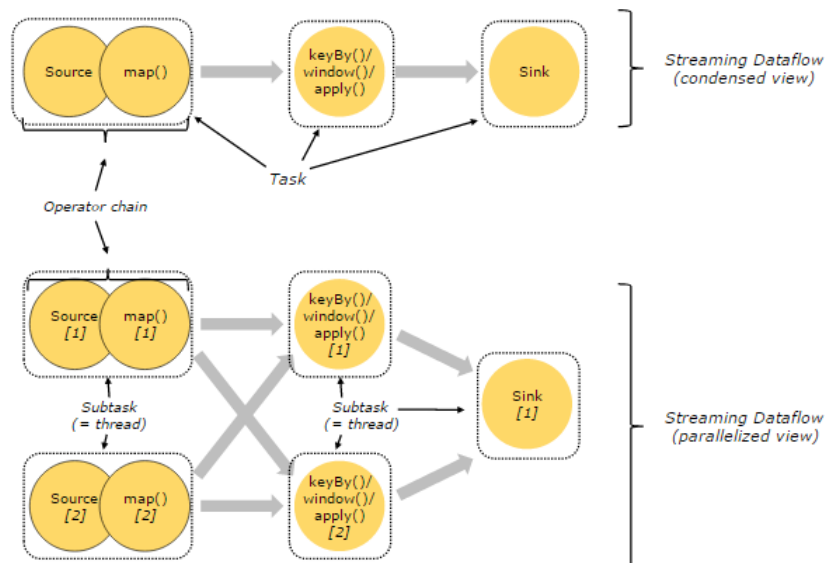
- Redistribution模式

这种模式改变了输入数据流的分区, 比如从map()[1]、map()[2]到keyBy()/window()/apply()[1]、keyBy()/window()/apply()[2], 上游的Subtask向下游的多个不同的Subtask发送数据, 改变了数据流的分区, 这与实际应用所选择的Operator有关系。

另外, Source Operator对应2个Subtask, 所以并行度为2, 而Sink Operator的Subtask只有1个, 故而并行度为1。

## Task & Operator Chain

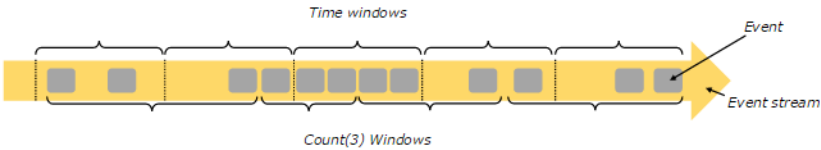
在Flink分布式执行环境中, 会将多个Operator Subtask串起来组成一个Operator Chain, 实际上就是一个执行链, 每个执行链会在TaskManager上一个独立的线程中执行, 如下图所示:



上图中上半部分表示的是一个Operator Chain, 多个Operator通过Stream连接, 而每个Operator在运行时对应一个Task; 图中下半部分是上半部分的一个并行版本, 也就是对每一个Task都并行化为多个Subtask。

## Time & Window

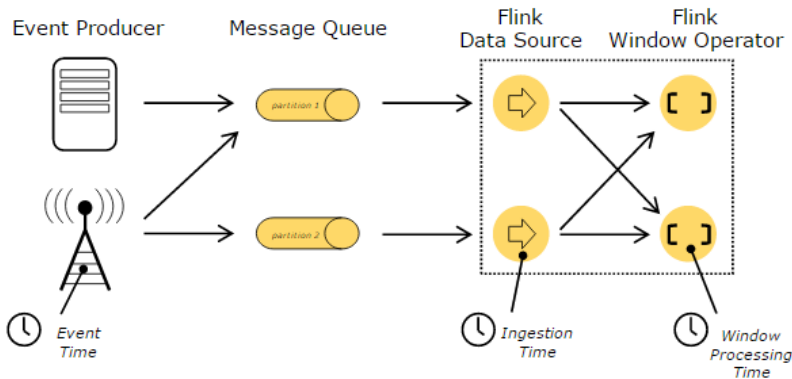
Flink支持基于时间窗口操作, 也支持基于数据的窗口操作, 如下图所示:



上图中，基于时间的窗口操作，在每个相同的时间间隔对Stream中的记录进行处理，通常各个时间间隔内的窗口操作处理的记录数不固定；而基于数据驱动的窗口操作，可以在Stream中选择固定数量的记录作为一个窗口，对该窗口中的记录进行处理。

有关窗口操作的不同类型，可以分为如下几种：倾斜窗口（Tumbling Windows，记录没有重叠）、滑动窗口（Slide Windows，记录有重叠）、会话窗口（Session Windows），具体可以查阅相关资料。

在处理Stream中的记录时，记录中通常会包含各种典型的时间字段，Flink支持多种时间的处理，如下图所示：

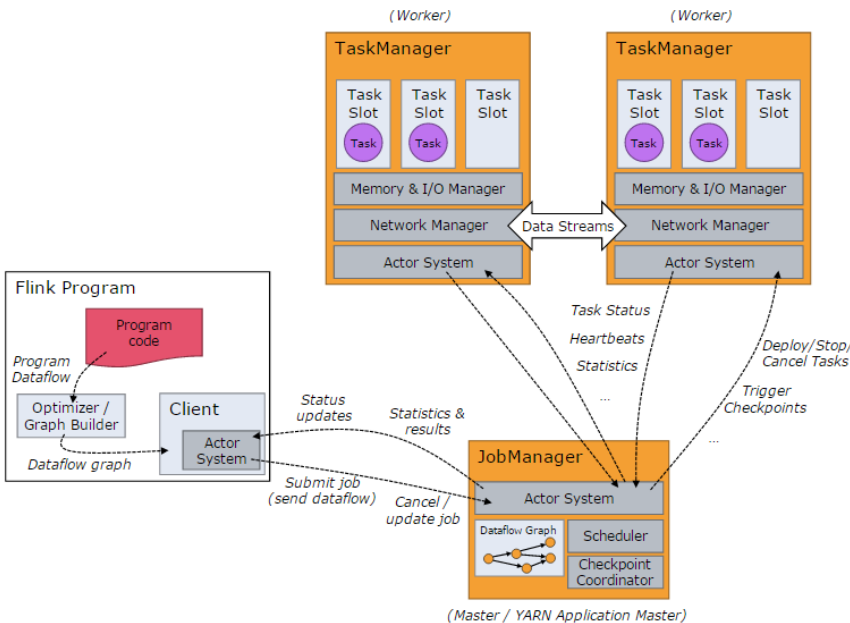


上图描述了在基于Flink的流处理系统中，各种不同的时间所处的位置和含义，其中，Event Time表示事件创建时间，Ingestion Time表示事件进入到Flink Dataflow的时间，Processing Time表示某个Operator对事件进行处理事的本地系统时间（是在TaskManager节点上）。这里，谈一下基于Event Time进行处理的问题，通常根据Event Time会给整个Streaming应用带来一定的延迟性，因为在一个基于事件的处理系统中，进入系统的事件可能会基于Event Time而发生乱序现象，比如事件来源于外部的多个系统，为了增强事件处理吞吐量会将输入的多个Stream进行自然分区，每个Stream分区内部有序，但是要保证全局有序必须同时兼顾多个Stream分区的处理，设置一定的时间窗口进行暂存数据，当多个Stream分区基于Event Time排列对齐后才能进行延迟处理。所以，设置的暂存数据记录的时间窗口越长，处理性能越差，甚至严重影响Stream处理的实时性。

有关基于时间的Streaming处理，可以参考官方文档，在Flink中借鉴了Google使用的WaterMark实现方式，可以查阅相关资料。

### 基本架构

Flink系统的架构与Spark类似，是一个基于Master-Slave风格的架构，如下图所示：



Flink集群启动时，会启动一个JobManager进程、至少一个TaskManager进程。在Local模式下，会在同一个JVM内部启动一个JobManager进程和TaskManager进程。当Flink程序提交后，会创建一个Client来进行预处理，并转换为一个并行数据流，这是对应着一个Flink Job，从而可以被JobManager和TaskManager执行。在实现上，Flink基于Actor实现了JobManager和TaskManager，所以JobManager与TaskManager之间的信息交换，都是通过事件的方式来进行处理。

如上图所示，Flink系统主要包含如下3个主要的进程：

## JobManager

JobManager是Flink系统的协调者，它负责接收Flink Job，调度组成Job的多个Task的执行。同时，JobManager还负责收集Job的状态信息，并管理Flink集群中从节点TaskManager。JobManager所负责的各项管理功能，它接收到并处理的事件主要包括：

- RegisterTaskManager

在Flink集群启动的时候，TaskManager会向JobManager注册，如果注册成功，则JobManager会向TaskManager回复消息AcknowledgeRegistration。

- SubmitJob

Flink程序内部通过Client向JobManager提交Flink Job，其中在消息SubmitJob中以JobGraph形式描述了Job的基本信息。

- CancelJob

请求取消一个Flink Job的执行，CancelJob消息中包含了Job的ID，如果成功则返回消息Cancellation，失败则返回消息CancellationFailure。

- UpdateTaskExecutionState

TaskManager会向JobManager请求更新ExecutionGraph中的ExecutionVertex的状态信息，更新成功则返回true。

- RequestNextInputSplit

运行在TaskManager上面的Task，请求获取下一个要处理的输入Split，成功则返回NextInputSplit。

- JobStatusChanged

ExecutionGraph向JobManager发送该消息，用来表示Flink Job的状态发生的变化，例如：RUNNING、CANCELING、FINISHED等。

## TaskManager

TaskManager也是一个Actor，它是实际负责执行计算的Worker，在其上执行Flink Job的一组Task。每个TaskManager负责管理其所在节点上的资源信息，如内存、磁盘、网络，在启动的时候将资源的状态向JobManager汇报。TaskManager端可以分成两个阶段：

- 注册阶段

TaskManager会向JobManager注册，发送RegisterTaskManager消息，等待JobManager返回AcknowledgeRegistration，然后TaskManager就可以进行初始化过程。

- 可操作阶段

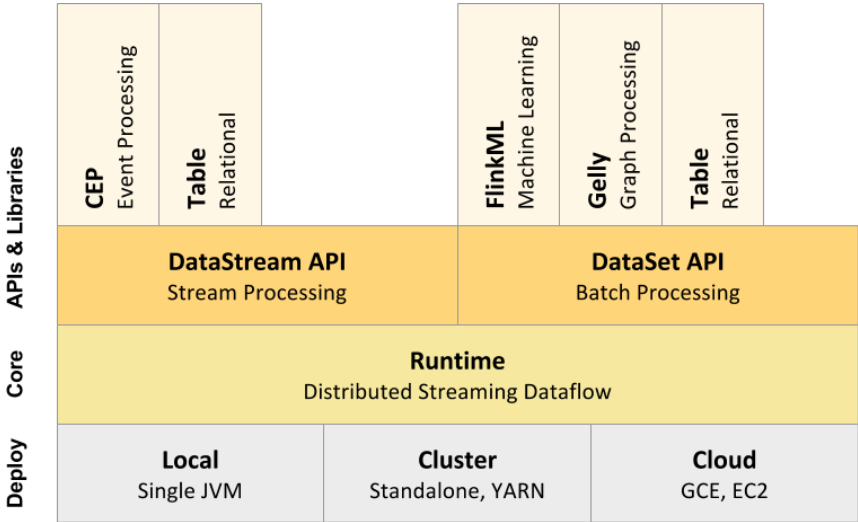
该阶段TaskManager可以接收并处理与Task有关的消息，如SubmitTask、CancelTask、FailTask。如果TaskManager无法连接到JobManager，这是TaskManager就失去了与JobManager的联系，会自动进入“注册阶段”，只有完成注册才能继续处理Task相关的消息。

## Client

当用户提交一个Flink程序时，会首先创建一个Client，该Client首先会对用户提交的Flink程序进行预处理，并提交到Flink集群中处理，所以Client需要从用户提交的Flink程序配置中获取JobManager的地址，并建立到JobManager的连接，将Flink Job提交给JobManager。Client会将用户提交的Flink程序组装一个JobGraph，并且是以JobGraph的形式提交的。一个JobGraph是一个Flink Dataflow，它由多个JobVertex组成的DAG。其中，一个JobGraph包含了一个Flink程序的如下信息：JobID、Job名称、配置信息、一组JobVertex等。

## 组件栈

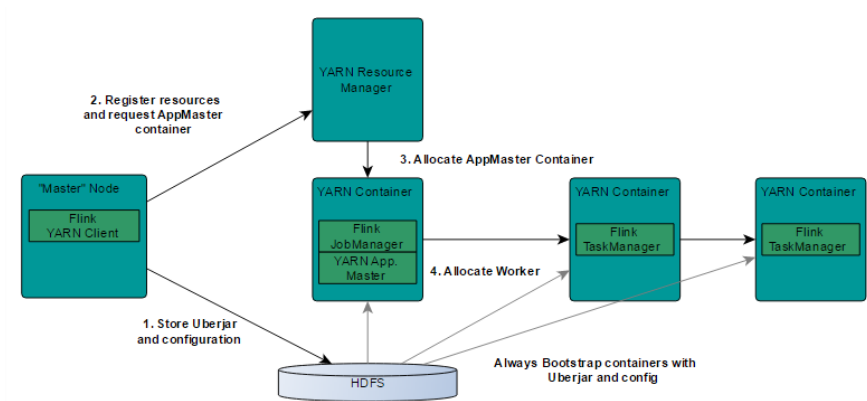
Flink是一个分层架构的系统，每一层所包含的组件都提供了特定的抽象，用来服务于上层组件。Flink分层的组件栈如下图所示：



下面，我们自下而上，分别针对每一层进行解释说明：

- Deployment层

该层主要涉及了Flink的部署模式，Flink支持多种部署模式：本地、集群（Standalone/YARN）、云（GCE/EC2）。Standalone部署模式与Spark类似，这里，我们看一下Flink on YARN的部署模式，如下图所示：



了解YARN的话，对上图的原理非常熟悉，实际Flink也实现了满足在YARN集群上运行的各个组件：Flink YARN Client负责与YARN RM通信协商资源请求，Flink JobManager和Flink TaskManager分别申请到Container去运行各自的进程。通过上图可以看到，YARN AM与Flink JobManager在同一个Container中，这样AM可以知道Flink JobManager的地址，从而AM可以申请Container去启动Flink TaskManager。待Flink成功运行在YARN集群上，Flink YARN Client就可以提交Flink Job到Flink JobManager，并进行后续的映射、调度和计算处理。

- Runtime层

Runtime层提供了支持Flink计算的全部核心实现，比如：支持分布式Stream处理、JobGraph到ExecutionGraph的映射、调度等等，为上层API层提供基础服务。

- API层

API层主要实现了面向无界Stream的流处理和面向Batch的批处理API，其中面向流处理对应DataStream API，面向批处理对应DataSet API。

- Libraries层

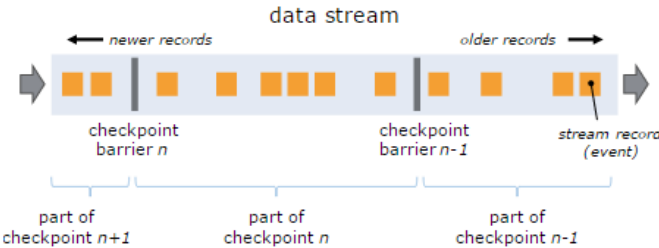
该层也可以称为Flink应用框架层，根据API层的划分，在API层之上构建的满足特定应用的实现计算框架，也分别对应于面向流处理和面向批处理两类。面向流处理支持：CEP（复杂事件处理）、基于SQL-like的操作（基于Table的关系操作）；面向批处理支持：FlinkML（机器学习库）、Gelly（图处理）。

## 内部原理

### 容错机制

Flink基于Checkpoint机制实现容错，它的原理是不断地生成分布式Streaming数据流Snapshot。在流处理失败时，通过这些Snapshot可以恢复数据流处理。理解Flink的容错机制，首先需要了解一下Barrier这个概念：Stream Barrier是Flink分布式Snapshotting中的核心元素，它会作为数据流的记录被同等看待，被插入到数据流中，将数据流中记录的进行分组，并沿着数据流的方向向前推进。每个Barrier会携带一个Snapshot ID，属于该Snapshot的记录会被推向该Barrier的前方。因为Barrier非常轻量，所以并不会中断数据流。带有Barrier的数据流，如下图所示：



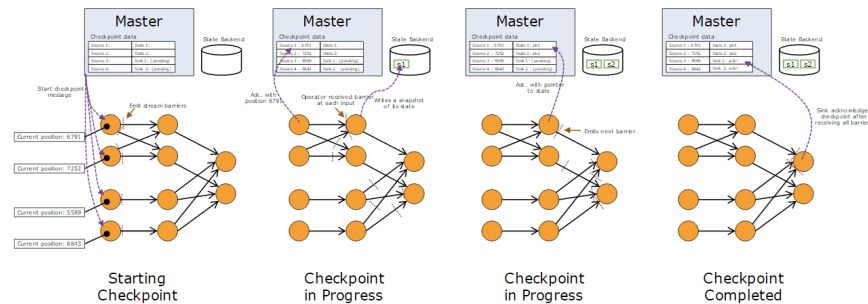


- 基于上图，我们通过如下要点来说明：
- 出现一个Barrier，在该Barrier之前出现的记录都属于该Barrier对应的Snapshot，在该Barrier之后出现的记录属于下一个Snapshot
  - 来自不同Snapshot多个Barrier可能同时出现在数据流中，也就是说同一个时刻可能并发生成多个Snapshot
  - 当一个中间（Intermediate）Operator接收到一个Barrier后，它会发送Barrier到属于该Barrier的Snapshot的数据流中，等到Sink Operator接收到该Barrier后会向Checkpoint Coordinator确认该Snapshot，直到所有的Sink Operator都确认了该Snapshot，才被认为完成了该Snapshot

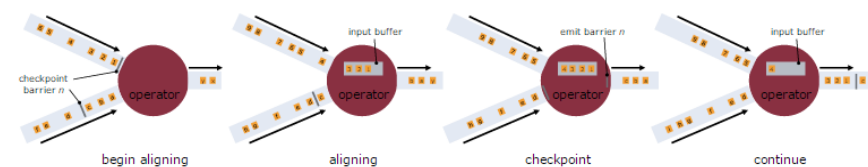
这里还需要强调的是，Snapshot并不仅仅是对数据流做了一个状态的Checkpoint，它也包含了一个Operator内部所持有的状态，这样能够在保证在流处理系统失败时能够正确地恢复数据流处理。也就是说，如果一个Operator包含任何形式的状态，这种状态必须是Snapshot的一部分。

Operator的状态包含两种：一种是系统状态，一个Operator进行计算处理的时候需要对数据进行缓冲。缓冲区的状态是与Operator相关联的，以窗口操作的缓冲区为例，Flink系统会收集或聚合记录数据并放到缓冲区中，直到该缓冲区中的数据被处理完成；另一种是用户自定义状态（状态可以通过转换函数进行创建和修改），它可以是函数中的Java对象这样的简单变量，也可以是与函数相关的Key/Value状态。

对于具有轻微状态的Streaming应用，会生成非常轻量的Snapshot而且非常频繁，但并不会影响数据流处理性能。Streaming应用的状态会被存储到一个可配置的存储系统中，例如HDFS。在一个Checkpoint执行过程中，存储的状态信息及其交互过程，如下图所示：



在Checkpoint过程中，还有一个比较重要的操作——Stream Aligning。当Operator接收到多个输入的数据流时，需要在Snapshot Barrier中对数据流进行排列对齐，如下图所示：

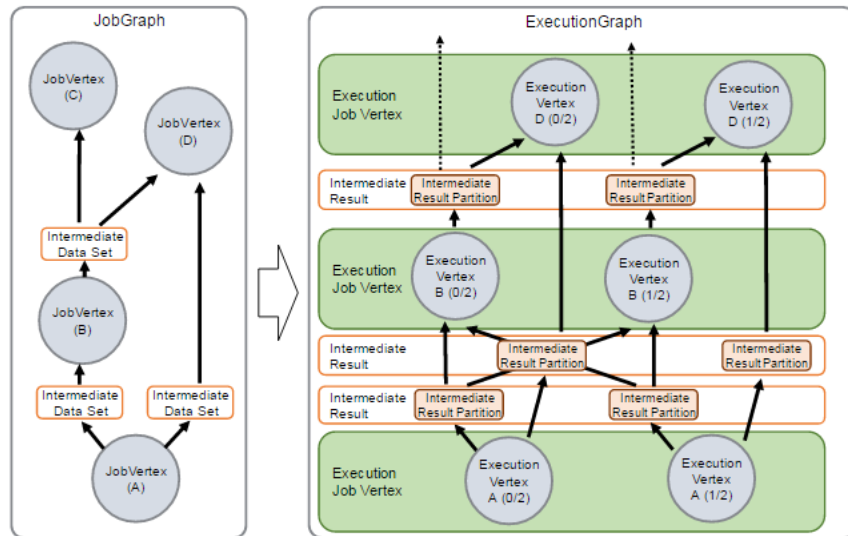


- 具体排列过程如下：
- Operator从一个incoming Stream接收到Snapshot Barrier n，然后暂停处理，直到其它的incoming Stream的Barrier n（否则属于2个Snapshot的记录就混在一起了）到达该Operator
  - 接收到Barrier n的Stream被临时搁置，来自这些Stream的记录不会被处理，而是被放在一个Buffer中
  - 一旦最后一个Stream接收到Barrier n，Operator会emit所有暂存在Buffer中的记录，然后向Checkpoint Coordinator发送Snapshot n
  - 继续处理来自多个Stream的记录

基于Stream Aligning操作能够实现Exactly Once语义，但是也会给流处理应用带来延迟，因为为了排列对齐Barrier，会暂时缓存一部分Stream的记录到Buffer中，尤其是在数据流并行度很高的场景下可能更加明显，通常以最迟对齐Barrier的一个Stream为处理Buffer中缓存记录的时刻点。在Flink中，提供了一个开关，选择是否使用Stream Aligning，如果关掉则Exactly Once会变成At least once。

### 调度机制

在JobManager端，会接收到Client提交的JobGraph形式的Flink Job，JobManager会将一个JobGraph转换映射为一个ExecutionGraph，如下图所示：

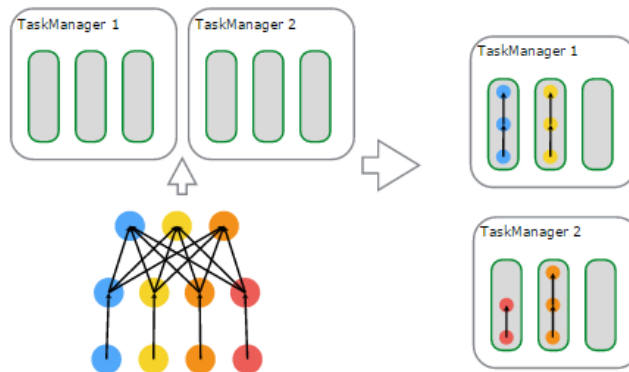


通过上图可以看出：

JobGraph是一个Job的用户逻辑视图表示，将一个用户要对数据流进行的处理表示为单个DAG图（对应于JobGraph），DAG图由顶点（JobVertex）和中间结果集（IntermediateDataSet）组成，其中JobVertex表示了对数据流进行的转换操作，比如map、flatMap、filter、keyBy等操作，而IntermediateDataSet是由上游的JobVertex所生成，同时作为下游的JobVertex的输入。

而ExecutionGraph是JobGraph的并行表示，也就是实际JobManager调度一个Job在TaskManager上运行的逻辑视图，它也是一个DAG图，是由ExecutionJobVertex、IntermediateResult（或IntermediateResultPartition）组成，ExecutionJobVertex实际对应于JobGraph图中的JobVertex，只不过在ExecutionJobVertex内部是一种并行表示，由多个并行的ExecutionVertex所组成。另外，这里还有一个重要的概念，就是Execution，它是一个ExecutionVertex的一次运行Attempt，也就是说，一个ExecutionVertex可能对应多个运行状态的Execution，比如，一个ExecutionVertex运行产生了一个失败的Execution，然后还会创建一个新的Execution来运行，这时就对应这个2次运行Attempt。每个Execution通过ExecutionAttemptID来唯一标识，在TaskManager和JobManager之间进行Task状态的交换都是通过ExecutionAttemptID来实现的。

下面看一下，在物理上进行调度，基于资源的分配与使用的一个例子，来自官网，如下图所示：



说明如下：

- 左上子图：有2个TaskManager，每个TaskManager有3个Task Slot
- 左下子图：一个Flink Job，逻辑上包含了1个data source、1个MapFunction、1个ReduceFunction，对应一个JobGraph
- 左下子图：用户提交的Flink Job对各个Operator进行的配置——data source的并行度设置为4，MapFunction的并行度也为4，ReduceFunction的并行度为3，在JobManager端对应于ExecutionGraph
- 右上子图：TaskManager 1上，有2个并行的ExecutionVertex组成的DAG图，它们各占用一个Task Slot
- 右下子图：TaskManager 2上，也有2个并行的ExecutionVertex组成的DAG图，它们也各占用一个Task Slot
- 在2个TaskManager上运行的4个Execution是并行执行的

## 迭代机制

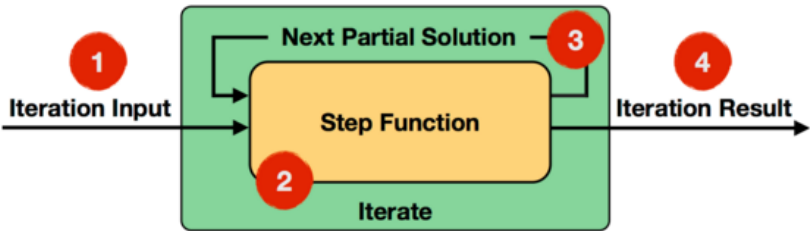
机器学习和图计算应用，都会使用到迭代计算，Flink通过在迭代Operator中定义Step函数来实现迭代算法，这种迭代算法包括Iterate和Delta Iterate两种类型，在实现上它们反复地在当前迭代状态上调用Step函数，直到满足



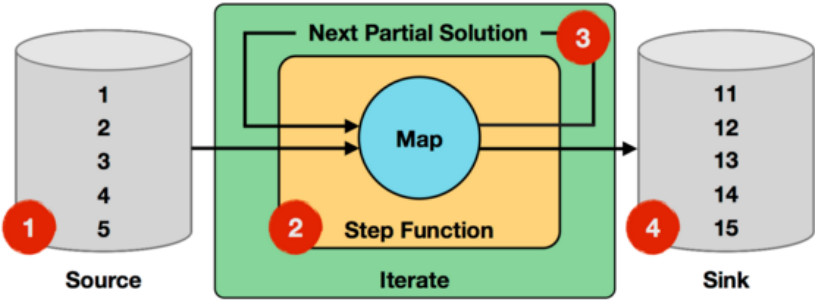
给定的条件才会停止迭代。下面，对Iterate和Delta Iterate两种类型的迭代算法原理进行说明：

- Iterate

Iterate Operator是一种简单的迭代形式：每一轮迭代，Step函数的输入或者是输入的整个数据集，或者是上一轮迭代的结果，通过该轮迭代计算出下一轮计算所需要的输入（也称为Next Partial Solution），满足迭代的终止条件后，会输出最终迭代结果，具体执行流程如下图所示：



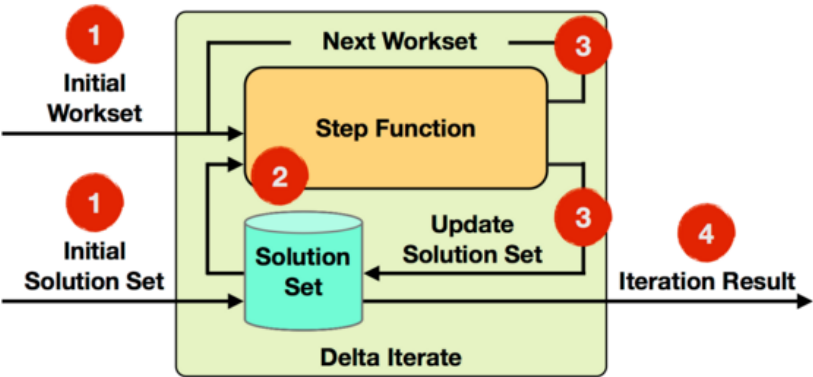
Step函数在每一轮迭代中都会被执行，它可以是由map、reduce、join等Operator组成的数据流。下面通过官网给出的一个例子来说明Iterate Operator，非常简单直观，如下图所示：



上面迭代过程中，输入数据为1到5的数字，Step函数就是一个简单的map函数，会对每个输入的数字进行加1处理，而Next Partial Solution对应于经过map函数处理后的结果，比如第一轮迭代，对输入的数字1加1后结果为2，对输入的数字2加1后结果为3，直到对输入数字5加1后结果为变为6，这些新生成结果数字2~6会作为第二轮迭代的输入。迭代终止条件为进行10轮迭代，则最终的结果为11~15。

- Delta Iterate

Delta Iterate Operator实现了增量迭代，它的实现原理如下图所示：

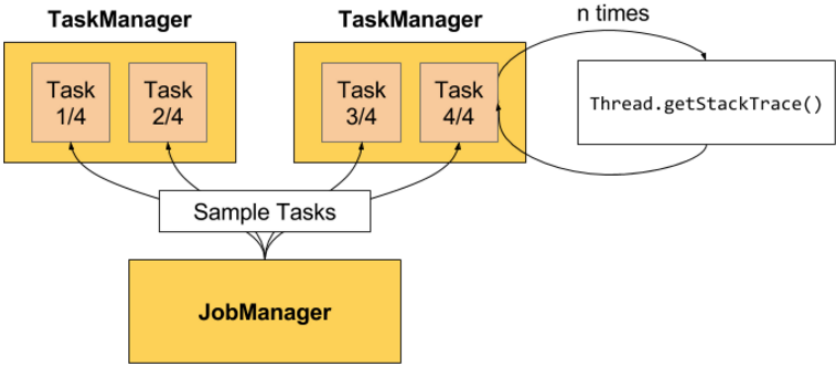


基于Delta Iterate Operator实现增量迭代，它有2个输入，其中一个是初始Workset，表示输入待处理的增量Stream数据，另一个是初始Solution Set，它是经过Stream方向上Operator处理过的结果。第一轮迭代会将Step函数作用在初始Workset上，得到的计算结果Workset作为下一轮迭代的输入，同时还要增量更新初始Solution Set。如果反复迭代知道满足迭代终止条件，最后会根据Solution Set的结果，输出最终迭代结果。比如，我们现在已知一个Solution集合中保存的是，已有的商品分类大类中购买量最多的商品，而Workset输入的是来自线上实时交易中最新达成购买的商品的人数，经过计算会生成新的商品分类大类中商品购买量最多的结果，如果某些大类中商品购买量突然增长，它需要更新Solution Set中的结果（原来购买量最多的商品，经过增量迭代计算，可能已经不是最多），最后会输出最终商品分类大类中购买量最多的商品结果集合。更详细的例子，可以参考官网给出的“Propagate Minimum in Graph”，这里不再赘述。

### Backpressure监控

Backpressure在流式计算系统中会比较受到关注，因为在一个Stream上进行处理的多个Operator之间，它们处理速度和方式可能非常不同，所以就存在上游Operator如果处理速度过快，下游Operator处可能会堆积Stream记录，严重会造成处理延迟或下游Operator负载过重而崩溃（有些系统可能会丢失数据）。因此，对下游Operator处理速度跟不上的情况，如果下游Operator能够将自己处理状态传播给上游Operator，使得上游Operator处理速度慢下来就会缓解上述问题，比如通过告警的方式通知现有流处理系统存在的问题。

Flink Web界面上提供了对运行Job的Backpressure行为的监控，它通过使用Sampling线程对正在运行的Task进行堆栈跟踪采样来实现，具体实现方式如下图所示：



JobManager会反复调用一个Job的Task运行所在线程的Thread.getStackTrace()，默认情况下，JobManager会每隔50ms触发对一个Job的每个Task依次进行100次堆栈跟踪调用，根据调用结果来确定Backpressure，Flink是通过计算得到一个比值（Radio）来确定当前运行Job的Backpressure状态。在Web界面上可以看到这个Radio值，它表示在一个内部方法调用中阻塞（Stuck）的堆栈跟踪次数，例如，radio=0.01，表示100次中仅有1次方法调用阻塞。Flink目前定义了如下Backpressure状态：

- OK: 0 <= Ratio <= 0.10
- LOW: 0.10 < Ratio <= 0.5
- HIGH: 0.5 < Ratio <= 1

另外，Flink还提供了3个参数来配置Backpressure监控行为：

参数名称	默认值	说明
jobmanager.web.backpressure.refresh-interval	60000	默认1分钟，表示采样统计结果刷新时间间隔
jobmanager.web.backpressure.num-samples	100	评估Backpressure状态，所使用的堆栈跟踪调用次数
jobmanager.web.backpressure.delay-between-samples	50	默认50毫秒，表示对一个Job的每个Task依次调用的时间间隔

通过上面个定义的Backpressure状态，以及调整相应的参数，可以确定当前运行的Job的状态是否正常，并且保证不影响JobManager提供服务。

顶 踩  
0 0

上一篇 [Lucene全文搜索原理与使用](#)  
下一篇 [为 Mahout 增加聚类评估功能](#)

猜你在找

- [分布式资源管理系统的前世今生，深入剖析YARN资源](#)
- [Apache Flink流作业提交流程分析](#)
- [Ceph—分布式存储系统的另一个选择](#)
- [Apache Flink fault tolerance源码剖析完结篇](#)
- [高并发集群架构超细精讲](#)
- [Apache Flink源码解析之stream-source](#)
- [【大数据技术公开课】YARN & Docker在Hulu的实战](#)
- [Flink on Yarn快速入门](#)
- [MySQL数据库分布式集群就业培训班](#)
- [Apache Flink源码解析之stream-operator](#)



[查看评论](#)

暂无评论

您还没有登录,请[\[登录\]](#)或[\[注册\]](#)

\* 以上用户言论只代表其个人观点，不代表CSDN网站的观点或立场

核心技术类目

全部主题

Hadoop

AWS

移动游戏

Java

Android

iOS

Swift

智能硬件

Docker

OpenStack

VPN

Spark

ERP

IE10

Eclipse

CRM

JavaScript

数据库

Ubuntu

NFC

WAP

jQuery

BI

HTML5

Spring

Apache

.NET

API

HTML

SDK

IIS

Fedora

XML

LBS

Unity

Splashtop

UML

components

Windows Mobile

Rails

QEMU

KDE

Cassandra

CloudStack

FTC

coremail

OPhone

CouchBase

云计算

iOS6

Rackspace

Web App

SpringSide

Maemo

Compuware

大数据

aptech

Perl

Tornado

Ruby

Hibernate

ThinkPHP

HBase

Pure

Solr

Angular

Cloud Foundry


Redis

Scala

Django

Bootstrap

公司简介 | 招贤纳士 | 广告服务 | 银行汇款帐号 | 联系方式 | 版权声明 | 法律顾问 | 问题报告 | 合作伙伴 | 论坛反馈

网站客服   杂志客服   微博客服   webmaster@csdn.net   400-600-2320 | 北京创新乐知信息技术有限公司 版权所有 | 江苏乐知网络技术有限公司 提供商务支持  
京 ICP 证 09002463 号 | Copyright © 1999-2014, CSDN.NET, All Rights Reserved 

0