

尚硅谷大数据课程之 Flink

(作者：章鹏)

官网：www.atguigu.com

版本：V1.0

第一章 概述

1.1 流处理技术的演变

在开源世界里，Apache Storm 项目是流处理的先锋。Storm 最早由 Nathan Marz 和创业公司 BackType 的一个团队开发，后来才被 Apache 基金会接纳。Storm 提供了低延迟的流处理，但是它为实时性付出了一些代价：**很难实现高吞吐**，并且其正确性没能达到通常所需的水平，换句话说，它**并不能保证 exactly-once**，即便是它能够保证的正确性级别，其开销也相当大。

在低延迟和高吞吐的流处理系统中维持良好的容错性是非常困难的，但是为了得到有保障的准确状态，人们想到了一种替代方法：**将连续时间中的流数据分割成一系列微小的批量作业**。如果分割得足够小（即所谓的微批处理作业），计算就几乎可以实现真正的流处理。因为存在延迟，所以不可能做到完全实时，但是每个简单的应用程序都可以实现仅有几秒甚至几亚秒的延迟。这就是在 Spark 批处理引擎上运行的 Spark Streaming 所使用的方法。

更重要的是，使用微批处理方法，可以实现 exactly-once 语义，从而保障状态的一致性。如果一个微批处理失败了，它可以重新运行，这比连续的流处理方法更容易。**Storm Trident 是对 Storm 的延伸，它的底层流处理引擎就是基于微批处理方法来进行计算的，从而实现了 exactly-once 语义，但是在延迟性方面付出了很大的代价。**

对于 Storm Trident 以及 Spark Streaming 等微批处理策略，只能根据批量作业时间的倍数进行分割，无法根据实际情况分割事件数据，并且，对于一些对延迟比较敏感的作业，往往需要开发者在写业务代码时花费大量精力来提升性能。这些灵活性和表现力方面的缺陷，使得这些微批处理策略开发速度变慢，运维成本变高。

于是，Flink 出现了，这一技术框架可以避免上述弊端，并且拥有所需的诸多功能，还能按照连续事件高效地处理数据，Flink 的部分特性如下图所示：

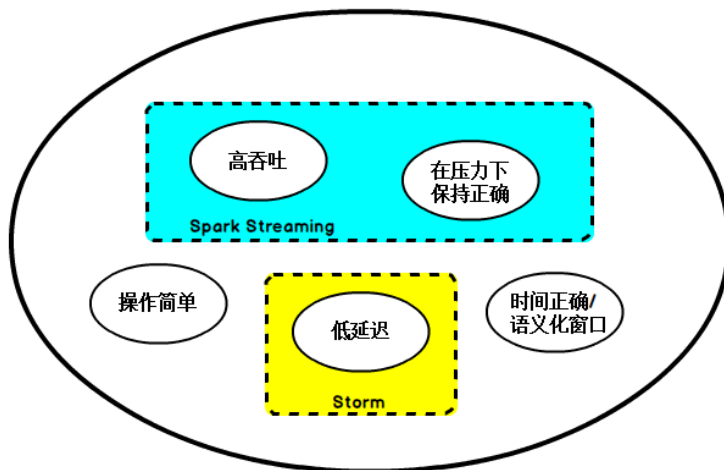


图 Flink 的部分特性

1.2 初识 Flink

Flink 起源于 Stratosphere 项目，Stratosphere 是在 2010~2014 年由 3 所地处柏林的大学和欧洲的一些其他的大学共同进行的研究项目，2014 年 4 月 Stratosphere 的代码被复制并捐赠给了 Apache 软件基金会，参加这个孵化项目的初始成员是 Stratosphere 系统的核心开发人员，2014 年 12 月，Flink 一跃成为 Apache 软件基金会的顶级项目。

在德语中，Flink 一词表示快速和灵巧，项目采用一只松鼠的彩色图案作为 logo，这不仅是因为松鼠具有快速和灵巧的特点，还因为柏林的松鼠有一种迷人的红棕色，而 Flink 的松鼠 logo 拥有可爱的尾巴，尾巴的颜色与 Apache 软件基金会的 logo 颜色相呼应，也就是说，这是一只 Apache 风格的松鼠。



图 Flink Logo

Flink 主页在其顶部展示了该项目的理念：“**Apache Flink 是为分布式、高性能、随时可用以及准确的流处理应用程序打造的开源流处理框架**”。

Apache Flink 是一个框架和分布式处理引擎，用于对无界和有界数据流进行有状态计算。Flink 被设计在所有常见的集群环境中运行，以内存执行速度和任意规模来执行计算。

1.3 Flink 核心计算框架

Flink 的核心计算架构是下图中的 Flink Runtime 执行引擎，它是一个分布式系统，能够接受数据流程序并在一台或多台机器上以容错方式执行。

Flink Runtime 执行引擎可以作为 YARN（Yet Another Resource Negotiator）的应用程序在集群上运行，也可以在 Mesos 集群上运行，还可以在单机上运行（这对于调试 Flink 应用程序来说非常有用）。

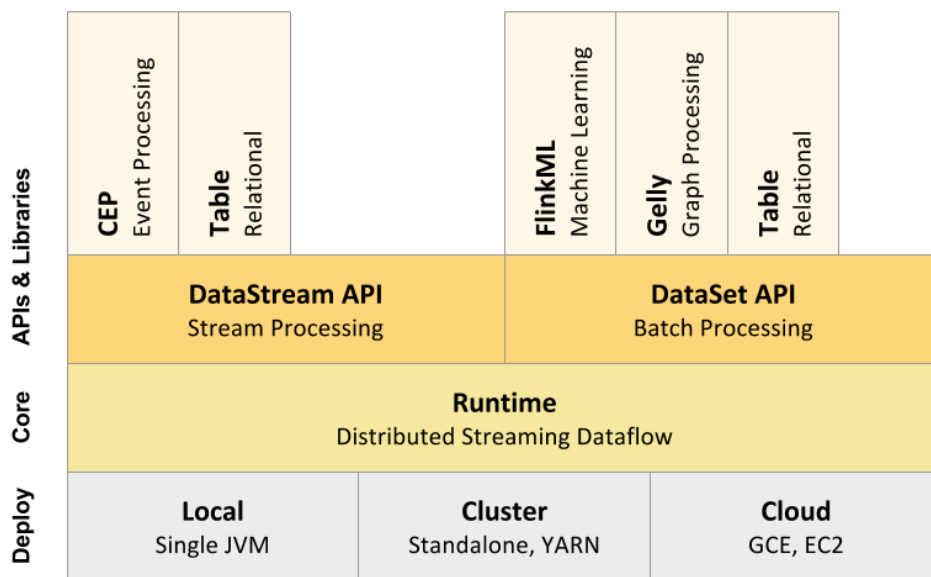


图 Flink 计算架构

上图为 Flink 技术栈的核心组成部分，值得一提的是，**Flink 分别提供了面向流式处理的接口（DataStream API）和面向批处理的接口（DataSet API）**。因此，Flink 既可以完成流处理，也可以完成批处理。Flink 支持的拓展库涉及机器学习（FlinkML）、复杂事件处理（CEP）、以及图计算（Gelly），还有分别针对流处理和批处理的 Table API。

能被 Flink Runtime 执行引擎接受的程序很强大，但是这样的程序有着冗长的代码，编写起来也很费力，基于这个原因，Flink 提供了封装在 Runtime 执行引擎之上的 API，以帮助用户方便地生成流式计算程序。**Flink 提供了用于流处理的 DataStream API 和用于批处理的 DataSet API**。值得注意的是，尽管 Flink Runtime 执行引擎是基于流处理的，但是 DataSet API 先于 DataStream API 被开发出来，这是因为工业界对无限流处理的需求在 Flink 诞生之初并不大。

DataStream API 可以流畅地分析无限数据流，并且可以用 Java 或者 Scala 来实现。开发人员需要基于一个叫 DataStream 的数据结构来开发，这个数据结构用于表示永不停止的分布式数据流。

Flink 的分布式特点体现在它能够在成百上千台机器上运行，它将大型的计算任务分成许多小的部分，每个机器执行一部分。Flink 能够自动地确保发生机器故障或者其他错误时计算能够持续进行，或者在修复 bug 或进行版本升级后有计划地再执行一次。这种能力使得开发人员不需要担心运行失败。Flink 本质上使用容错性数据流，这使得开发人员可以分析持续生成且永远不结束的数据（即流处理）。

【更多 Java、HTML5、Android、python、大数据 资料下载，可访问尚硅谷（中国）官网下载区】

第二章 Flink 基本架构

2.5 JobManager 与 TaskManager

Flink 运行时包含了两种类型的处理器：

JobManager 处理器：也称之为 Master，用于协调分布式执行，它们用来调度 task，协调检查点，协调失败时恢复等。Flink 运行时至少存在一个 master 处理器，如果配置高可用模式则会存在多个 master 处理器，它们其中有一个是 leader，而其他的都是 standby。

TaskManager 处理器：也称之为 Worker，用于执行一个 dataflow 的 task(或者特殊的 subtask)、数据缓冲和 data stream 的交换，Flink 运行时至少会存在一个 worker 处理器。

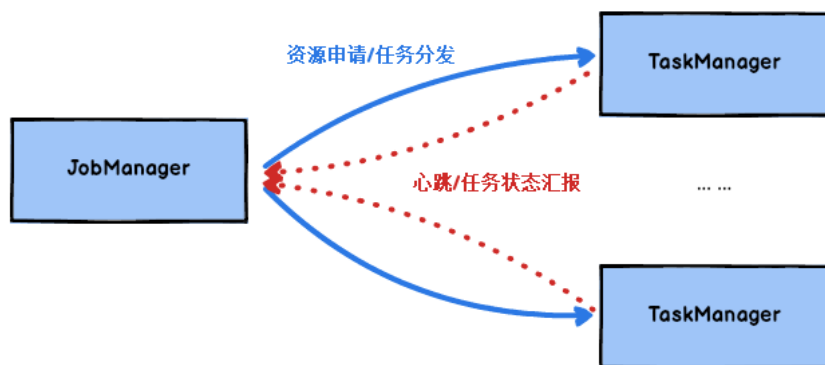


图 JobManager 与 TaskManager

Master 和 Worker 处理器可以直接在物理机上启动，或者通过像 YARN 这样的资源调度框架。

Worker 连接到 Master，告知自身的可用性进而获得任务分配。

2.1 无界数据流与有界数据流

Flink 用于处理有界和无界数据：

无界数据流：无界数据流有一个开始但是没有结束，它们不会在生成时终止并提供数据，必须连续处理无界流，也就是说必须在获取后立即处理 event。对于无界数据流我们无法等待所有数据都到达，因为输入是无界的，并且在任何时间点都不会完成。处理无界数据通常要求以特定顺序（例如事件发生的顺序）获取 event，以便能够推断结果完整性，无界流的处理称为流处理。

有界数据流：有界数据流有明确定义的开始和结束，可以在执行任何计算之前通过获取所有数据来处理有界流，处理有界流不需要有序获取，因为可以始终对有界数据集进行排序，有界流的处理也称为批处理。

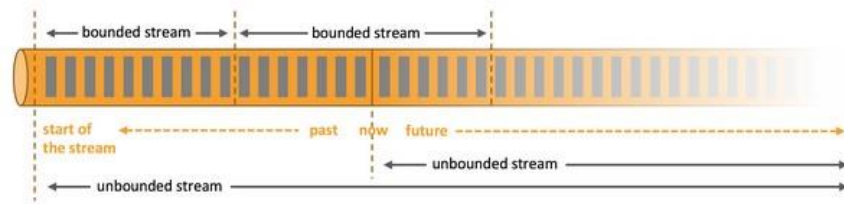


图 无界数据流与有界数据流

在无界数据流和有界数据流中我们提到了批处理和流处理，这是大数据处理系统中常见的两种数据处理方式。

批处理的特点是有界、持久、大量，批处理非常适合需要访问全套记录才能完成的计算工作，一般用于离线统计。流处理的特点是无界、实时，流处理方式无需针对整个数据集执行操作，而是对通过系统传输的每个数据项执行操作，一般用于实时统计。

在 Spark 生态体系中，对于批处理和流处理采用了不同的技术框架，批处理由 SparkSQL 实现，流处理由 Spark Streaming 实现，这也是大部分框架采用的策略，使用独立的处理器实现批处理和流处理，而 Flink 可以同时实现批处理和流处理。

Flink 是如何同时实现批处理与流处理的呢？答案是，**Flink 将批处理（即处理有限的静态数据）视作一种特殊的流处理。**

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，因为它们要实现的目标是完全不相同的：**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。

2.2 数据流编程模型

Flink 提供了不同级别的抽象，以开发流或批处理作业，如下图所示：

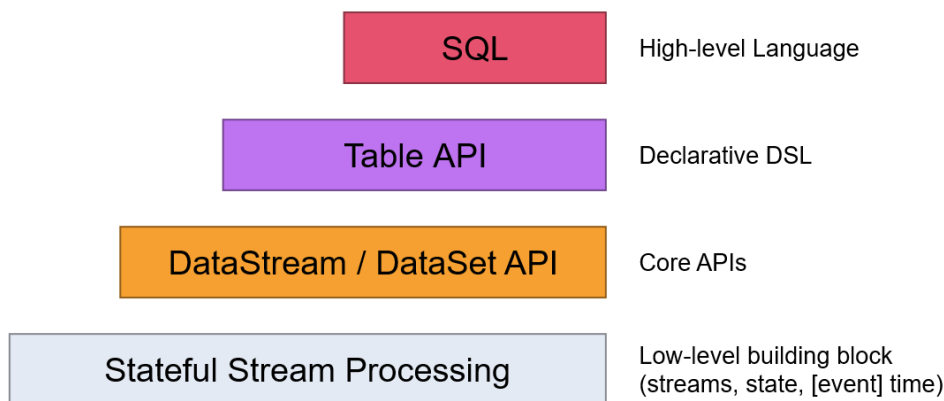


图 Flink 抽象级别

最底层级的抽象仅仅提供了有状态流，它将通过过程函数（Process Function）被嵌入到 DataStream API 中。底层过程函数（Process Function）与 DataStream API 相集成，使其可以对某些特定的操作进行底层的抽象，它允许用户可以自由地处理来自一个或多个数据流的事件，并使用一致的容错的状态。除此之外，用户可以注册事件时间并处理时间回调，从而使程序可以处理复杂的计算。

实际上，**大多数应用并不需要上述的底层抽象，而是针对核心 API（Core APIs）进行编程，比如 DataStream API（有界或无界流数据）以及 DataSet API（有界数据集）**。这些 API 为数据处理提供了通用的构建模块，比如由用户定义的多种形式的转换（transformations），连接（joins），聚合（aggregations），窗口操作（windows）等等。DataSet API 为有界数据集提供了额外的支持，例如循环与迭代。这些 API 处理的数据类型以类（classes）的形式由各自的编程语言所表示。

Table API 以表为中心，其中表可能会动态变化（在表达流数据时）。Table API 遵循（扩展的）关系模型：表有二维数据结构（schema）（类似于关系数据库中的表），同时 API 提供可比较的操作，例如 select、project、join、group-by、aggregate 等。Table API 程序声明式地定义了什么逻辑操作应该执行，而不是准确地确定这些操作代码的看上去如何。尽管 Table API 可以通过多种类型的用户自定义函数（UDF）进行扩展，其仍不如核心 API 更具表达能力，但是使用起来却更加简洁（代码量更少）。除此之外，Table API 程序在执行之前会经过内置优化器进行优化。

你可以在表与 DataStream/DataSet 之间无缝切换，以允许程序将 Table API 与 DataStream 以及 DataSet 混合使用。

Flink 提供的最高层级的抽象是 SQL。这一层抽象在语法与表达能力上与 Table API 类似，但是是以 SQL 查询表达式的形式表现程序。SQL 抽象与 Table API 交互密切，同时 SQL 查询可以直接在 Table API 定义的表上执行。

第三章 Flink 集群搭建

Flink 可以选择的部署方式有：

Local、Standalone（资源利用率低）、Yarn、Mesos、Docker、Kubernetes、AWS。

我们主要对 Standalone 模式和 Yarn 模式下的 Flink 集群部署进行分析。

3.1 Standalone 模式安装

我们对 standalone 模式的 Flink 集群进行安装，准备三台虚拟机，其中一台作为 JobManager（hadoop-senior01），另外两台作为 TaskManager（hadoop-senior02、hadoop-senior03）。

1. 在官网下载 1.6.1 版本 Flink（<https://archive.apache.org/dist/flink/flink-1.6.1/>）。
2. 将安装包上传到要按照 JobManager 的节点（hadoop-senior01）。
3. 进入 Linux 系统对安装包进行解压：

```
[crazyjack@hadoop-senior01 ~]$ cd /opt/software/
[crazyjack@hadoop-senior01 software]$ tar -xzf flink-1.6.1-bin-hadoop27-scala_2.11.tgz -C /opt/modules/
```

4. 修改安装目录下 conf 文件夹内的 flink-conf.yaml 配置文件，指定 JobManager：

```
[crazyjack@hadoop-senior01 ~]$ cd /opt/modules/flink-1.6.1/
[crazyjack@hadoop-senior01 flink-1.6.1]$ cd conf/
[crazyjack@hadoop-senior01 conf]$ vim flink-conf.yaml
```

```
#####
# Common
#####

# The external address of the host on which the JobManager runs and can be
# reached by the TaskManagers and any clients which want to connect. This setting
# is only used in Standalone mode and may be overwritten on the JobManager side
# by specifying the --host <hostname> parameter of the bin/jobmanager.sh executable.
# In high availability mode, if you use the bin/start-cluster.sh script and setup
# the conf/masters file, this will be taken care of automatically. Yarn/Mesos
# automatically configure the host name based on the hostname of the node where the
# JobManager runs.

jobmanager.rpc.address: hadoop-senior01.itguigu.com
```

5. 修改安装目录下 conf 文件夹内的 slave 配置文件，指定 TaskManager：

```
[crazyjack@hadoop-senior01 conf]$ vim slaves
You have new mail in /var/spool/mail/crazyjack
```

```
hadoop-senior02.itguigu.com
hadoop-senior03.itguigu.com
~
~
~
```

6. 将配置好的 Flink 目录分发给其他的两台节点：

```
[crazyjack@hadoop-senior01 modules]$ xsync flink-1.6.1
```

7. 在 hadoop-senior01 节点启动集群：

```
[crazyjack@hadoop-senior01 modules]$ cd flink-1.6.1/
[crazyjack@hadoop-senior01 flink-1.6.1]$ bin/start-cluster.sh
```

8. 通过 jps 查看进程信息：

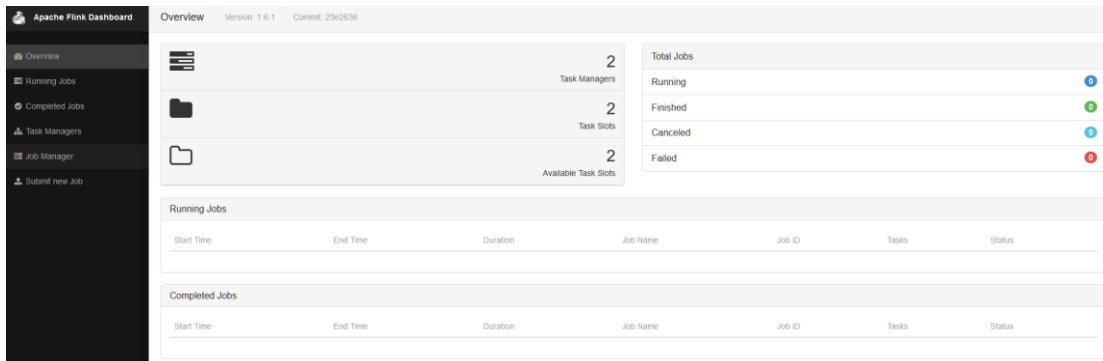
```
[crazyjack@hadoop-senior01 flink-1.6.1]$ jps
11826 StandaloneSessionClusterEntrypoint
12146 RunJar
12319 Jps
```

```
[crazyjack@hadoop-senior02 ~]$ jps
3249 TaskManagerRunner
3287 Jps
```

```
[crazyjack@hadoop-senior03 ~]$ jps
3366 Jps
3322 TaskManagerRunner
```

【更多 Java、HTML5、Android、python、大数据 资料下载，可访问尚硅谷（中国）官网下载区】

9. 访问集群 web 界面（8081 端口）：



| Overview | | Version: 1.6.1 | Commit: 23c2636 |
|----------------|----------|-------------------------|-----------------|
| | 2 | Total Jobs | |
| | 2 | Running 0 | |
| | 2 | Finished 0 | |
| | | Canceled 0 | |
| | | Failed 0 | |
| Running Jobs | | | |
| Start Time | End Time | Duration | Job Name |
| | | | |
| Completed Jobs | | | |
| Start Time | End Time | Duration | Job Name |
| | | | |

3.2Yarn 模式安装

- 1 在官网下载 1.6.1 版本 Flink（<https://archive.apache.org/dist/flink/flink-1.6.1/>）。
- 2 将安装包上传到要按照 JobManager 的节点（hadoop-senior01）。
- 3 进入 Linux 系统对安装包进行解压：

```
[crazyjack@hadoop-senior01 ~]$ cd /opt/software/
[crazyjack@hadoop-senior01 software]$ tar -xzf flink-1.6.1-bin-hadoop27-scala_2.11.tgz -C /opt/modules/
```

- 4 修改安装目录下 conf 文件夹内的 flink-conf.yaml 配置文件，指定 JobManager：

```
[crazyjack@hadoop-senior01 ~]$ cd /opt/modules/flink-1.6.1/
[crazyjack@hadoop-senior01 flink-1.6.1]$ cd conf/
[crazyjack@hadoop-senior01 conf]$ vim flink-conf.yaml
```

```
#=====
# Common
#=====

# The external address of the host on which the JobManager runs and can be
# reached by the TaskManagers and any clients which want to connect. This setting
# is only used in Standalone mode and may be overwritten on the JobManager side
# by specifying the --host <hostname> parameter of the bin/jobmanager.sh executable.
# In high availability mode, if you use the bin/start-cluster.sh script and setup
# the conf/masters file, this will be taken care of automatically. Yarn/Mesos
# automatically configure the host name based on the hostname of the node where the
# JobManager runs.

jobmanager.rpc.address: hadoop-senior01.itguigu.com
```

- 5 修改安装目录下 conf 文件夹内的 slave 配置文件，指定 TaskManager：

```
[crazyjack@hadoop-senior01 conf]$ vim slaves
You have new mail in /var/spool/mail/crazyjack
```

```
hadoop-senior02.itguigu.com
hadoop-senior03.itguigu.com
~
~
~
```

- 6 将配置好的 Flink 目录分发给其他的两台节点：

```
[crazyjack@hadoop-senior01 modules]$ xsync flink-1.6.1
```

- 7 明确虚拟机中已经设置好了环境变量 HADOOP_HOME。
- 8 启动 Hadoop 集群（HDFS 和 Yarn）。
- 9 在 hadoop-senior01 节点提交 Yarn-Session，使用安装目录下 bin 目录中的

【更多 Java、HTML5、Android、python、大数据 资料下载，可访问尚硅谷（中国）官网下载区】

yarn-session.sh 脚本进行提交：

```
/opt/modules/flink-1.6.1/bin/yarn-session.sh -n 2 -s 6 -jm 1024 -tm 1024 -nm test -d
```

其中：

-n(--container)：TaskManager 的数量。

-s(--slots)：每个 TaskManager 的 slot 数量，默认一个 slot 一个 core，默认每个 taskmanager 的 slot 的个数为 1。

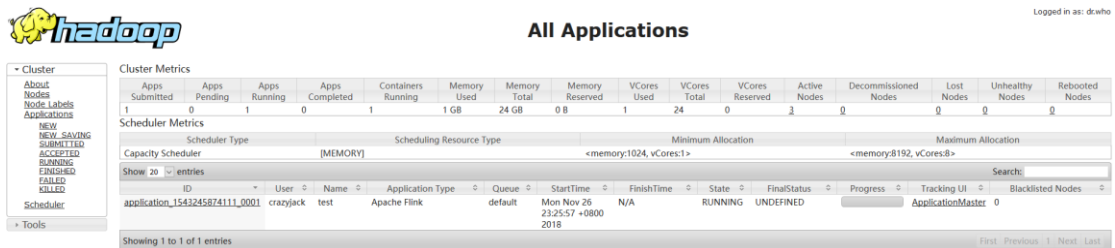
-jm：JobManager 的内存（单位 MB）。

-tm：每个 taskmanager 的内存（单位 MB）。

-nm：yarn 的 appName(现在 yarn 的 ui 上的名字)。

-d：后台执行。

10. 启动后查看 Yarn 的 Web 页面，可以看到刚才提交的会话：



The screenshot shows the Hadoop All Applications page. On the left is a sidebar with navigation links like Cluster, About, Nodes, Node Labels, Applications, NEW, NEW SAVING, SUBMITTED, ACCEPTED, RUNNING, FINISHED, FAILED, KILLED, Scheduler, and Tools. The main content area displays Cluster Metrics and Scheduler Metrics. The Cluster Metrics table shows various statistics including Apps Submitted, Pending, Running, Completed, Containers Running, Memory Used, Memory Total, Memory Reserved, VCoers Used, VCoers Total, VCoers Reserved, Active Nodes, Decommissioned Nodes, Lost Nodes, Unhealthy Nodes, and Rebooted Nodes. The Scheduler Metrics section shows the Scheduler Type (Capacity Scheduler), Scheduling Resource Type (MEMORY), and Minimum/Maximum Allocation. Below this is a table of applications with columns for ID, User, Name, Application Type, Queue, StartTime, FinishTime, State, FinalStatus, Progress, Tracking UI, and Blacklisted Nodes. One application is listed: application_1543245874111_0001, user crazyjack, name test, Application Type Apache Flink, Queue default, StartTime Mon Nov 26 23:25:57 +0800 2018, FinishTime N/A, State RUNNING, FinalStatus UNDEFINED, Progress ApplicationMaster 0, Tracking UI ApplicationMaster 0, and Blacklisted Nodes 0.

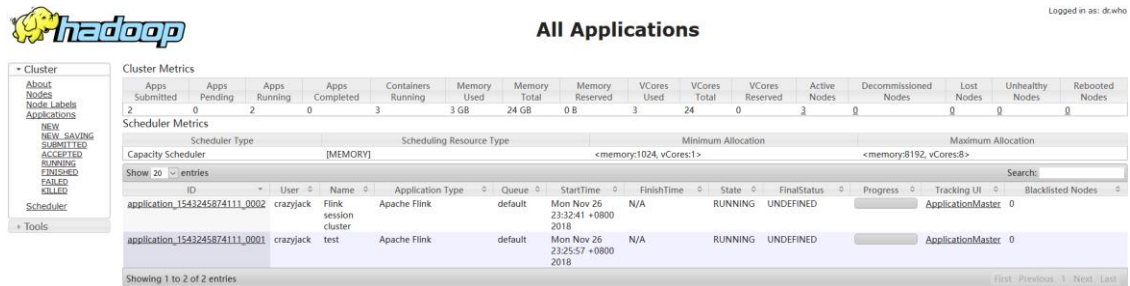
11. 在提交 Session 的节点查看进程：

```
[crazyjack@hadoop-senior01 flink-1.6.1]$ jps
4101 NodeManager
3832 NameNode
3976 DataNode
5354 Jps
5227 YarnSessionClusterEntrypoint
```

12. 提交 Jar 到集群运行：

```
/opt/modules/flink-1.6.1/bin/flink run -m yarn-cluster examples/batch/WordCount.jar
```

13. 提交后在 Yarn 的 Web 页面查看任务运行情况：



The screenshot shows the Hadoop All Applications page. On the left is a sidebar with navigation links like Cluster, About, Nodes, Node Labels, Applications, NEW, NEW SAVING, SUBMITTED, ACCEPTED, RUNNING, FINISHED, FAILED, KILLED, Scheduler, and Tools. The main content area displays Cluster Metrics and Scheduler Metrics. The Cluster Metrics table shows various statistics including Apps Submitted, Pending, Running, Completed, Containers Running, Memory Used, Memory Total, Memory Reserved, VCoers Used, VCoers Total, VCoers Reserved, Active Nodes, Decommissioned Nodes, Lost Nodes, Unhealthy Nodes, and Rebooted Nodes. The Scheduler Metrics section shows the Scheduler Type (Capacity Scheduler), Scheduling Resource Type (MEMORY), and Minimum/Maximum Allocation. Below this is a table of applications with columns for ID, User, Name, Application Type, Queue, StartTime, FinishTime, State, FinalStatus, Progress, Tracking UI, and Blacklisted Nodes. Two applications are listed: application_1543245874111_0002 and application_1543245874111_0001. The first application is in a pending state, and the second is in a running state.

14. 任务运行结束后在控制台打印如下输出：

```
(a,5)
(action,1)
(after,1)
(against,1)
(all,2)
(and,12)
(arms,1)
(arrows,1)
(awry,1)
(ay,1)
(bare,1)
(be,4)
(bear,3)
(bodkin,1)
(bourn,1)
(but,1)
(by,2)
(calamity,1)
```

第四章 Flink 运行架构

4.1 任务提交流程

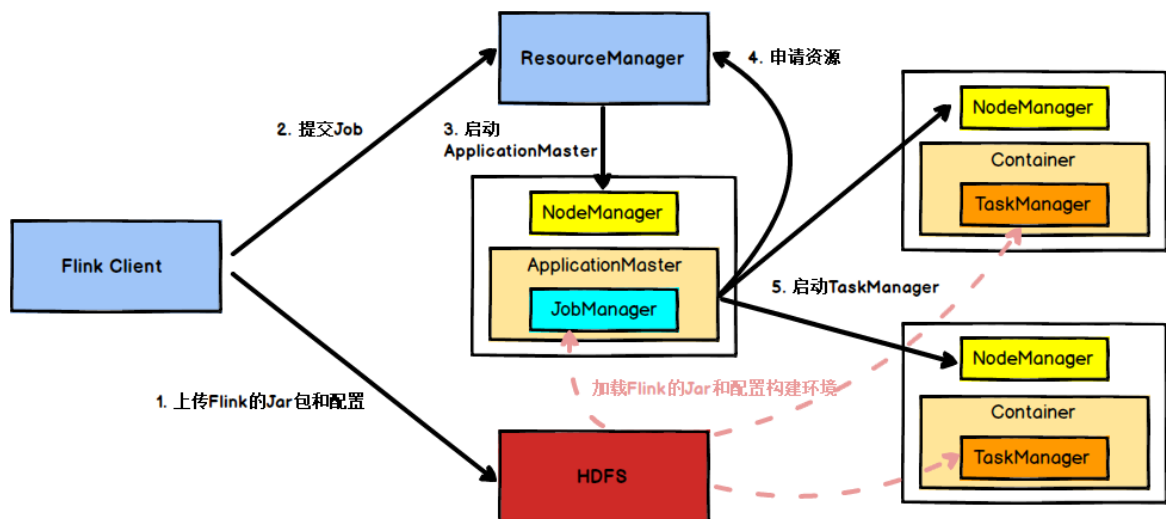


图 Yarn 模式任务提交流程

Flink 任务提交后，Client 向 HDFS 上传 Flink 的 Jar 包和配置，之后向 Yarn Resource Manager 提交任务，Resource Manager 分配 Container 资源并通知对应的 NodeManager 启动 ApplicationMaster，ApplicationMaster 启动后加载 Flink 的 Jar 包和配置构建环境，然后启动 JobManager，之后 ApplicationMaster 向 Resource Manager 申请资源启动 TaskManager，Resource Manager 分配 Container 资源后，由 ApplicationMaster 通知资源所在节点的 NodeManager 启动 TaskManager，NodeManager 加载 Flink 的 Jar 包和配置构建环境并启动 TaskManager，TaskManager 启动后向 JobManager 发送心跳包，并等待 JobManager 向其分配任务。

4.2 TaskManager 与 Slots

每一个 **TaskManager** 是一个 **JVM 进程**，它可能会在独立的线程上执行一个或多个 **subtask**。为了控制一个 worker 能接收多少个 task，worker 通过 task slot 来进行控制（一个 worker 至少有一个 task slot）。•

每个 task slot 表示 TaskManager 拥有资源的一个固定大小的子集。假如一个 TaskManager 有三个 slot，那么它会将其管理的内存分成三份给各个 slot。**资源 slot 化**意味着一个 **subtask** 将不需要跟来自其他 **job** 的 **subtask** 竞争被管理的内存，取而代之的是它将拥有一定数量的内存储备。需要注意的是，这里不会涉及到 CPU 的隔离，slot 目前仅仅用来隔离 task 的受管理的内存。

通过调整 **task slot** 的数量，允许用户定义 **subtask** 之间如何互相隔离。如果一个 TaskManager 一个 slot，那将意味着每个 task group 运行在独立的 JVM 中（该 JVM 可能是通过一个特定的容器启动的），而一个 TaskManager 多个 slot 意味着更多的 subtask 可以共享同一个 JVM。而在同一个 JVM 进程中的 task 将共享 TCP 连接（基于多路复用）和心跳消息。它们也可能共享数据集和数据结构，因此这减少了每个 task 的负载。

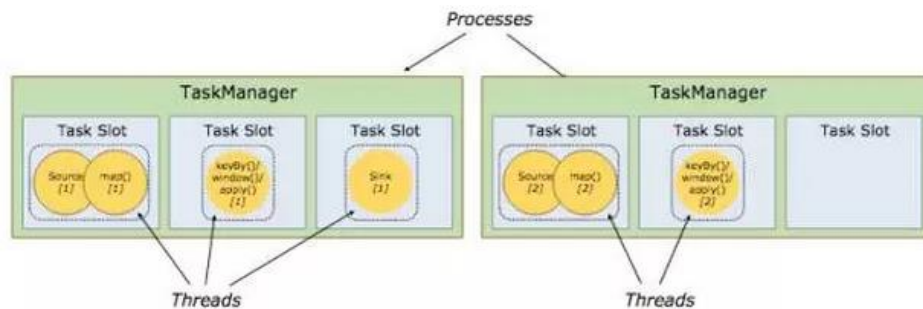


图 TaskManager 与 Slot

Task Slot 是静态的概念，是指 **TaskManager** 具有的并发执行能力，可以通过参数 `taskmanager.numberOfTaskSlots` 进行配置，而 **并行度 parallelism** 是动态概念，即 **TaskManager** 运行程序时实际使用的并发能力，可以通过参数 `parallelism.default` 进行配置。

也就是说，假设一共有 3 个 TaskManager，每一个 TaskManager 中的分配 3 个 TaskSlot，也就是每个 TaskManager 可以接收 3 个 task，一共 9 个 TaskSlot，如果我们设置 `parallelism.default=1`，即运行程序默认的并行度为 1，9 个 TaskSlot 只用了 1 个，有 8 个空闲，因此，设置合适的并行度才能提高效率。

4.3 Dataflow

Flink 程序由 Source、Transformation、Sink 这三个核心组件组成，Source 主要负责数据的读取，Transformation 主要负责对数据的转换操作，Sink 负责最终数据的输出，在各个组件之间流转的数据称为流（streams）。

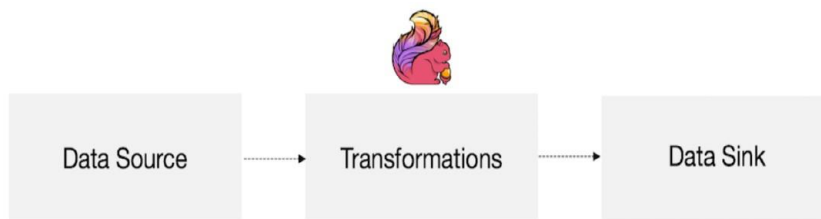


图 Flink 程序模型

Flink 程序的基础构建模块是 **流**（streams）与 **转换**（transformations）（需要注意的是，Flink 的 DataSet API 所使用的 DataSets 其内部也是 stream）。一个 stream 可以看成是一个中间结果，而一个 transformations 是以一个或多个 stream 作为输入的某种 operation，该 operation 利用这些 stream 进行计算从而产生一个或多个 result stream。

在运行时，Flink 上运行的程序会被映射成 **streaming dataflows**，它包含了 **streams 和 transformations operators**。每一个 dataflow 以一个或多个 sources 开始以一个或多个 sinks 结束，dataflow 类似于任意的有向无环图（DAG）。

```
DataStream<String> lines = env.addSource(  
    new FlinkKafkaConsumer<> {...});  
DataStream<Event> events = lines.map((line) -> parse(line));  
DataStream<Statistics> stats = events  
    .keyBy("id")  
    .timeWindow(Time.seconds(10))  
    .apply(new MyWindowAggregationFunction());  
stats.addSink(new RollingSink(path));
```

Source
Transformation
Transformation
Sink

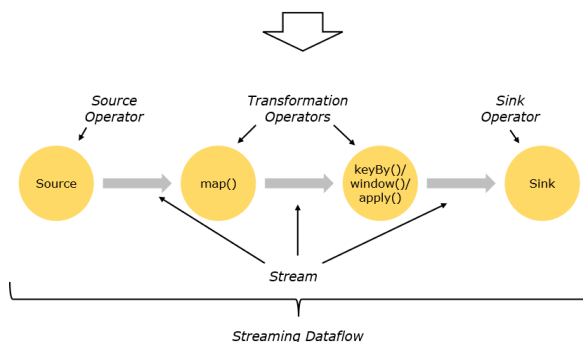


图 程序与数据流

4.4 并行数据流

Flink 程序的执行具有并行、分布式的特性。在执行过程中，一个 stream 包含一个或多个 stream partition，而每一个 operator 包含一个或多个 operator subtask，这些 operator subtasks 在不同的线程、不同的物理机或不同的容器中彼此互不依赖得执行。

一个特定 operator 的 subtask 的个数被称之为其 **parallelism(并行度)**。一个 stream 的并行度总是等同于其 producing operator 的并行度。一个程序中，不同的 operator 可能具有不同的并行度。

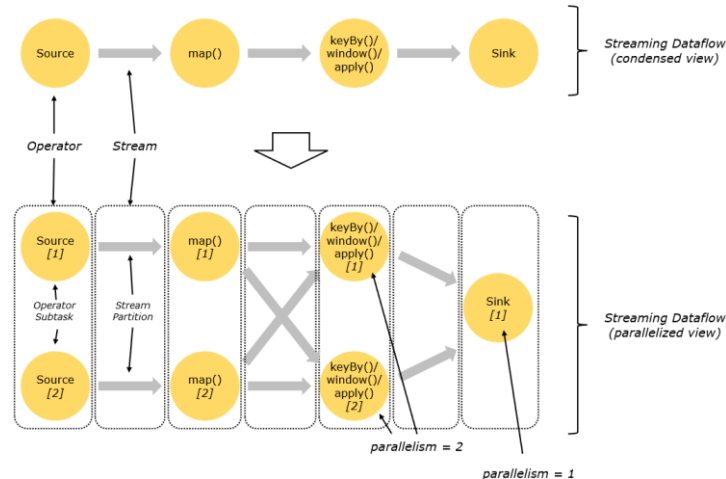


图 并行数据流

Stream 在 operator 之间传输数据的形式可以是 one-to-one(forwarding)的模式也可以是 redistributing 的模式，具体是哪一种形式，取决于 operator 的种类。

One-to-one: stream(比如在 source 和 map operator 之间)维护着分区以及元素的顺序。那意味着 map operator 的 subtask 看到的元素的个数以及顺序跟 source operator 的 subtask 生产的元素的个数、顺序相同，map、filter、flatMap 等算子都是 one-to-one 的对应关系。

Redistributing: 这种操作会改变数据的分区个数。每一个 operator subtask 依据所选择的 transformation 发送数据到不同的目标 subtask。例如，keyBy() 基于 hashCode 重分区、broadcast 和 rebalance 会随机重新分区，这些算子都会引起 redistribute 过程，而 redistribute 过程就类似于 Spark 中的 shuffle 过程。

4.5 task 与 operator chains

出于分布式执行的目的，Flink 将 operator 的 subtask 链接在一起形成 task，每个 task 在一个线程中执行。将 operators 链接成 task 是非常有效的优化：它能减少线程之间的切换和基于缓存区的数据交换，在减少时延的同时提升吞吐量。链接的行为可以在编程 API 中进行指定。

下面这幅图，展示了 5 个 subtask 以 5 个并行的线程来执行：

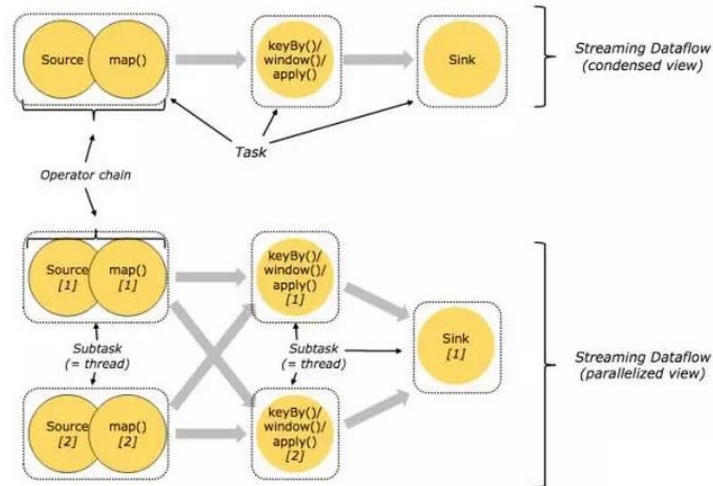


图 task 与 operator chains

4.6 任务调度流程

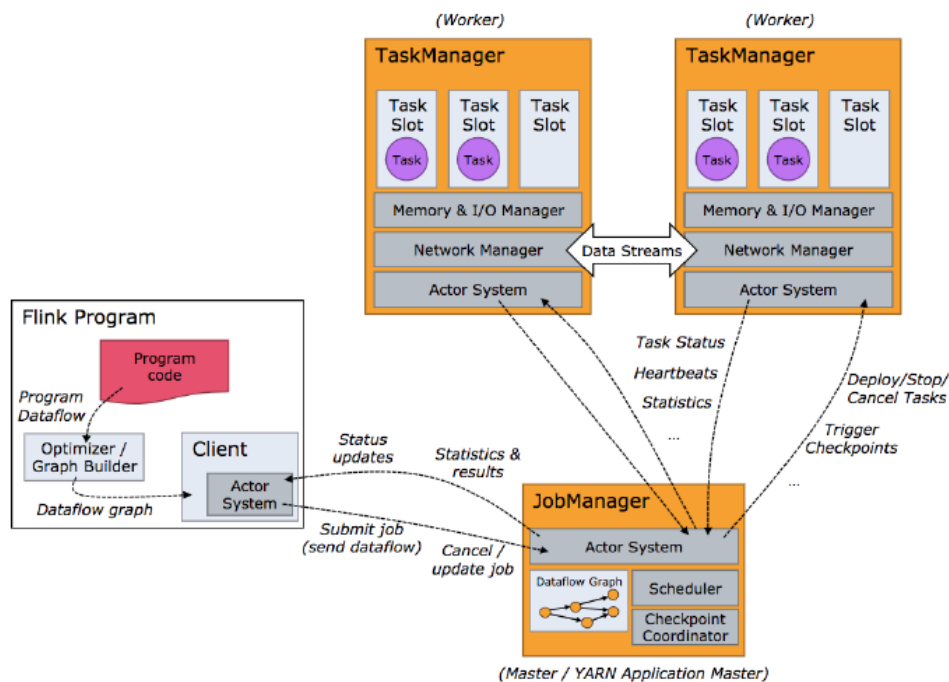


图 任务调度原理

客户端不是运行时和程序执行的一部分，但它用于准备并发送 dataflow 给 Master，然后，客户端断开连接或者维持连接以等待接收计算结果，客户端可以以两种方式运行：要么作为 Java/Scala 程序的一部分被程序触发执行，要么以命令 `./bin/flink run` 的方式执行。

第四章 Flink DataStream API

5.1 Flink 运行模型

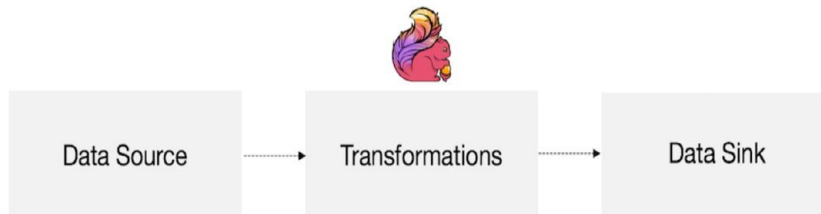


图 Flink 程序模型

以上为 Flink 的运行模型，Flink 的程序主要由三部分构成，分别为 Source、Transformation、Sink。DataSource 主要负责数据的读取，Transformation 主要负责对属于的转换操作，Sink 负责最终数据的输出。

5.2 Flink 程序架构

每个 Flink 程序都包含以下的若干流程：

- 获得一个执行环境；（Execution Environment）
- 加载/创建初始数据；（Source）
- 指定转换这些数据；（Transformation）
- 指定放置计算结果的位置；（Sink）
- 触发程序执行。

5.3 Environment

执行环境 `StreamExecutionEnvironment` 是所有 Flink 程序的基础。

创建执行环境有三种方式，分别为：

```
StreamExecutionEnvironment.getExecutionEnvironment  
StreamExecutionEnvironment.createLocalEnvironment  
StreamExecutionEnvironment.createRemoteEnvironment
```

5.3.1 `StreamExecutionEnvironment.getExecutionEnvironment`

创建一个执行环境，表示当前执行程序的上下文。如果程序是独立调用的，则此方法返回本地执行环境；如果从命令行客户端调用程序以提交到集群，则此方法返回此集群的执行环境，也就是说，`getExecutionEnvironment` 会根据查询运行的方式决定返回什么样的运行环境，是最常用的一种创建执行环境的方式。

```
val env = StreamExecutionEnvironment.getExecutionEnvironment
```

【更多 Java、HTML5、Android、python、大数据 资料下载，可访问尚硅谷（中国）官网下载区】

5.3.2 StreamExecutionEnvironment.createLocalEnvironment

返回本地执行环境，需要在调用时指定默认的并行度。

```
val env = StreamExecutionEnvironment.createLocalEnvironment(1)
```

5.3.3 StreamExecutionEnvironment.createRemoteEnvironment

返回集群执行环境，将 Jar 提交到远程服务器。需要在调用时指定 JobManager 的 IP 和端口号，并指定要在集群中运行的 Jar 包。

```
val env = StreamExecutionEnvironment.createRemoteEnvironment(1)
```

5.4 Source

5.4.1 基于 File 的数据源

1. readTextFile(path)

一列一列的读取遵循 TextInputFormat 规范的文本文件，并将结果作为 String 返回。

```
val env = StreamExecutionEnvironment.getExecutionEnvironment
val stream = env.readTextFile("/opt/modules/test.txt")
stream.print()
env.execute("FirstJob")
```

注意：stream.print(): 每一行前面的数字代表这一行是哪一个并行线程输出的。

2. readFile(fileInputFormat, path)

按照指定的文件格式读取文件。

```
val env = StreamExecutionEnvironment.getExecutionEnvironment
val path = new Path("/opt/modules/test.txt")
val stream = env.readFile(new TextInputFormat(path), "/opt/modules/test.txt")
stream.print()
env.execute("FirstJob")
```

5.4.2 基于 Socket 的数据源

1. socketTextStream

从 Socket 中读取信息，元素可以用分隔符分开。

```
val env = StreamExecutionEnvironment.getExecutionEnvironment
val stream = env.socketTextStream("localhost", 11111)
stream.print()
env.execute("FirstJob")
```

5.4.3 基于集合（Collection）的数据源

1. fromCollection(seq)

从集合中创建一个数据流，集合中所有元素的类型是一致的。

```
val env = StreamExecutionEnvironment.getExecutionEnvironment
val list = List(1,2,3,4)
val stream = env.fromCollection(list)
stream.print()
env.execute("FirstJob")
```

2. fromCollection(Iterator)

从迭代(Iterator)中创建一个数据流，指定元素数据类型的类由 iterator 返回。

```
val env = StreamExecutionEnvironment.getExecutionEnvironment
val iterator = Iterator(1,2,3,4)
val stream = env.fromCollection(iterator)
stream.print()
env.execute("FirstJob")
```

3. fromElements(elements:_*)

从一个给定的对象序列中创建一个数据流，所有的对象必须是相同类型的。

```
val env = StreamExecutionEnvironment.getExecutionEnvironment
val list = List(1,2,3,4)
val stream = env.fromElement(list)
stream.print()
env.execute("FirstJob")
```

4. generateSequence(from, to)

从给定的间隔中并行地产生一个数字序列。

```
val env = StreamExecutionEnvironment.getExecutionEnvironment
val stream = env.generateSequence(1,10)
stream.print()
env.execute("FirstJob")
```

5.5 Sink

Data Sink 消费 DataStream 中的数据，并将它们转发到文件、套接字、外部系统或者打印出。

Flink 有许多封装在 DataStream 操作里的内置输出格式。

5.6.1 writeAsText

将元素以字符串形式逐行写入（TextOutputFormat），这些字符串通过调用每个元素的 toString() 方法来获取。

5.6.2 WriteAsCsv

将元组以逗号分隔写入文件中（CsvOutputFormat），行及字段之间的分隔是可配置的。每个字段的值来自对象的 toString() 方法。

5.6.3 print/printToErr

打印每个元素的 toString() 方法的值到标准输出或者标准错误输出流中。或者也可以在输出流中添加一个前缀，这个可以帮助区分不同的打印调用，如果并行度大于 1，那么输出也会有一个标识由哪个任务产生的标志。

5.6.4 writeUsingOutputFormat

自定义文件输出的方法和基类（FileOutputFormat），支持自定义对象到字节的转换。

5.6.5 writeToSocket

根据 SerializationSchema 将元素写入到 socket 中。

5.6 Transformation

5.6.1 Map

DataStream → DataStream: 输入一个参数产生一个参数。

```
val env = StreamExecutionEnvironment.getExecutionEnvironment

val stream = env.generateSequence(1,10)
val streamMap = stream.map { x => x * 2 }
streamFilter.print()

env.execute("FirstJob")
```

5.6.2 FlatMap

DataStream → DataStream: 输入一个参数，产生 0 个、1 个或者多个输出。

```
val env = StreamExecutionEnvironment.getExecutionEnvironment

val stream = env.readTextFile("test.txt")
val streamFlatMap = stream.flatMap{
    x => x.split(" ")
}
```

【更多 Java、HTML5、Android、python、大数据 资料下载，可访问尚硅谷（中国）官网下载区】

```
streamFilter.print()

env.execute("FirstJob")
```

5.6.3 Filter

DataStream → DataStream: 结算每个元素的布尔值，并返回布尔值为 true 的元素。下面这个例子是过滤出非 0 的元素：

```
val env = StreamExecutionEnvironment.getExecutionEnvironment

val stream = env.generateSequence(1,10)
val streamFilter = stream.filter{
    x => x == 1
}
streamFilter.print()

env.execute("FirstJob")
```

5.6.4 Connect

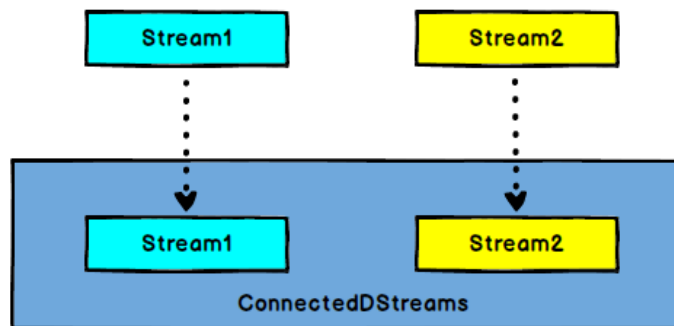


图 Connect 算子

DataStream,DataStream → ConnectedStreams: 连接两个保持他们类型的数据流，两个数据流被 Connect 之后，只是被放在了一个同一个流中，内部依然保持各自的数据和形式不发生变化，两个流相互独立。

```
val env = StreamExecutionEnvironment.getExecutionEnvironment

val stream = env.readTextFile("test.txt")

val streamMap = stream.flatMap(item => item.split(" ")).filter(item =>
item.equals("hadoop"))
val streamCollect = env.fromCollection(List(1,2,3,4))

val streamConnect = streamMap.connect(streamCollect)

streamConnect.map(item=>println(item), item=>println(item))

env.execute("FirstJob")
```

5.6.5 CoMap, CoFlatMap

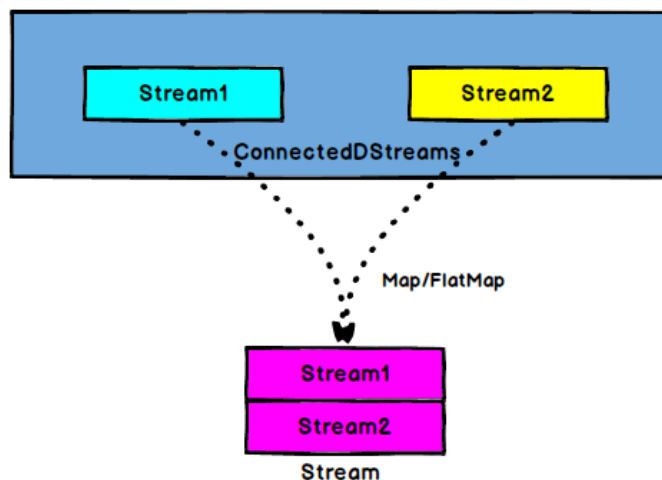


图 CoMap/CoFlatMap

ConnectedStreams → DataStream: 作用于 ConnectedStreams 上，功能与 map 和 flatMap 一样，对 ConnectedStreams 中的每一个 Stream 分别进行 map 和 flatMap 处理。

```
val env = StreamExecutionEnvironment.getExecutionEnvironment

val stream1 = env.readTextFile("test.txt")
val streamFlatMap = stream1.flatMap(x => x.split(" "))
val stream2 = env.fromCollection(List(1,2,3,4))
val streamConnect = streamFlatMap.connect(stream2)
val streamCoMap = streamConnect.map(
    (str) => str + "connect",
    (in) => in + 100
)

env.execute("FirstJob")
```

```
val env = StreamExecutionEnvironment.getExecutionEnvironment

val stream1 = env.readTextFile("test.txt")
val stream2 = env.readTextFile("test1.txt")
val streamConnect = stream1.connect(stream2)
val streamCoMap = streamConnect.flatMap(
    (str1) => str1.split(" "),
    (str2) => str2.split(" ")
)
streamConnect.map(item=>println(item), item=>println(item))

env.execute("FirstJob")
```


5.6.6 Split

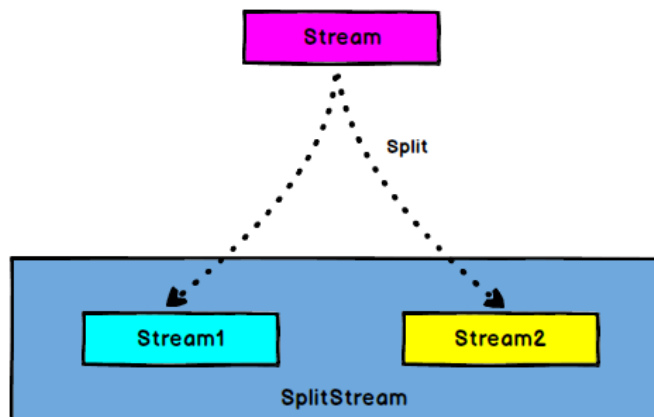


图 Split

DataStream → SplitStream: 根据某些特征把一个 DataStream 拆分成两个或者多个 DataStream。

```
val env = StreamExecutionEnvironment.getExecutionEnvironment

val stream = env.readTextFile("test.txt")
val streamFlatMap = stream.flatMap(x => x.split(" "))
val streamSplit = streamFlatMap.split(
  num =>
    # 字符串内容为 hadoop 的组成一个 DataStream, 其余的组成一个 DataStream
    (num.equals("hadoop")) match{
      case true => List("hadoop")
      case false => List("other")
    }
)

env.execute("FirstJob")
```

5.6.7 Select

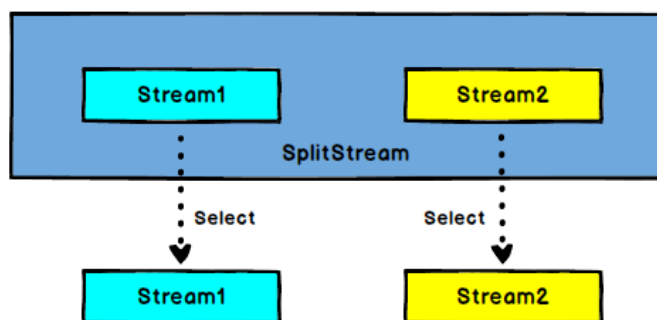


图 Select

SplitStream → DataStream: 从一个 SplitStream 中获取一个或者多个 DataStream。

```
val env = StreamExecutionEnvironment.getExecutionEnvironment

val stream = env.readTextFile("test.txt")
val streamFlatMap = stream.flatMap(x => x.split(" "))
val streamSplit = streamFlatMap.split(
```

【更多 Java、HTML5、Android、python、大数据 资料下载，可访问尚硅谷（中国）官网下载区】

```
num =>
  (num.equals("hadoop")) match{
    case true => List("hadoop")
    case false => List("other")
  }
)

val hadoop = streamSplit.select("hadoop")
val other = streamSplit.select("other")
hadoop.print()

env.execute("FirstJob")
```

5.6.8 Union

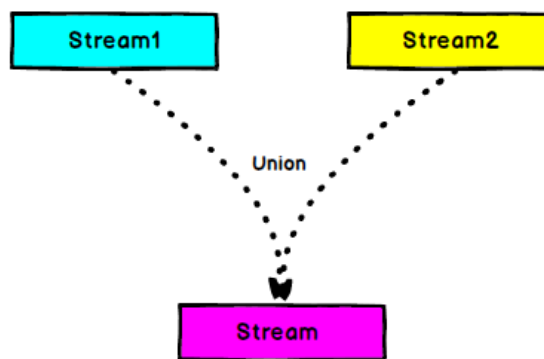


图 Union

DataStream → DataStream: 对两个或者两个以上的 DataStream 进行 union 操作，产生一个包含所有 DataStream 元素的新 DataStream。注意:如果你将一个 DataStream 跟它自己做 union 操作，在新的 DataStream 中，你将看到每一个元素都出现两次。

```
val env = StreamExecutionEnvironment.getExecutionEnvironment

val stream1 = env.readTextFile("test.txt")
val streamFlatMap1 = stream1.flatMap(x => x.split(" "))
val stream2 = env.readTextFile("test1.txt")
val streamFlatMap2 = stream2.flatMap(x => x.split(" "))
val streamConnect = streamFlatMap1.union(streamFlatMap2)

env.execute("FirstJob")
```

5.6.9 KeyBy

DataStream → KeyedStream: 输入必须是 Tuple 类型，逻辑地将一个流拆分成不相交的分区，每个分区包含具有相同 key 的元素，在内部以 hash 的形式实现的。

```
val env = StreamExecutionEnvironment.getExecutionEnvironment
val stream = env.readTextFile("test.txt")
val streamFlatMap = stream.flatMap{
  x => x.split(" ")
}
val streamMap = streamFlatMap.map{
  x => (x,1)
```

【更多 Java、HTML5、Android、python、大数据 资料下载，可访问尚硅谷（中国）官网下载区】

```
}  
val streamKeyBy = streamMap.keyBy(0)  
env.execute("FirstJob")
```

5.6.10 Reduce

KeyedStream → DataStream: 一个分组数据流的聚合操作，合并当前的元素和上次聚合的结果，产生一个新的值，返回的流中包含每一次聚合的结果，而不是只返回最后一次聚合的最终结果。

```
val env = StreamExecutionEnvironment.getExecutionEnvironment  
  
val stream = env.readTextFile("test.txt").flatMap(item => item.split(" ")).map(item =>  
    (item, 1)).keyBy(0)  
  
val streamReduce = stream.reduce(  
    (item1, item2) => (item1._1, item1._2 + item2._2)  
)  
  
streamReduce.print()  
  
env.execute("FirstJob")
```

5.6.11 Fold

KeyedStream → DataStream: 一个有初始值的分组数据流的滚动折叠操作，合并当前元素和前一次折叠操作的结果，并产生一个新的值，返回的流中包含每一次折叠的结果，而不是只返回最后一次折叠的最终结果。

```
val env = StreamExecutionEnvironment.getExecutionEnvironment  
  
val stream = env.readTextFile("test.txt").flatMap(item => item.split(" ")).map(item =>  
    (item, 1)).keyBy(0)  
  
val streamReduce = stream.fold(100)(  
    (begin, item) => (begin + item._2)  
)  
  
streamReduce.print()  
  
env.execute("FirstJob")
```

5.6.12 Aggregations

KeyedStream → DataStream: 分组数据流上的滚动聚合操作。`min` 和 `minBy` 的区别是 `min` 返回的是一个最小值，而 `minBy` 返回的是其字段中包含最小值的元素(同样原理适用于 `max` 和 `maxBy`)，返回的流中包含每一次聚合的结果，而不是只返回最后一次聚合的最终结果。

```
keyedStream.sum(0)  
keyedStream.sum("key")  
keyedStream.min(0)  
keyedStream.min("key")
```

【更多 Java、HTML5、Android、python、大数据 资料下载，可访问尚硅谷（中国）官网下载区】

```
keyedStream.max(0)
keyedStream.max("key")
keyedStream.minBy(0)
keyedStream.minBy("key")
keyedStream.maxBy(0)
keyedStream.maxBy("key")
```

```
val env = StreamExecutionEnvironment.getExecutionEnvironment

val stream = env.readTextFile("test02.txt").map(item => (item.split(" ")(0), item.split(" ")
(1).toLong)).keyBy(0)

val streamReduce = stream.sum(1)

streamReduce.print()

env.execute("FirstJob")
```

在 2.3.10 之前的算子都是可以直接作用在 Stream 上的,因为他们不是聚合类型的操作,但是到 2.3.10 后你会发现,我们虽然可以对一个无边界的流数据直接应用聚合算子,但是它会记录下每一次的聚合结果,这往往不是我们想要的,其实,reduce、fold、aggregation 这些聚合算子都是和 Window 配合使用的,只有配合 Window,才能得到想要的结果。

第六章 Time 与 Window

6.1 Time

在 Flink 的流式处理中,会涉及到时间的不同概念,如下图所示:

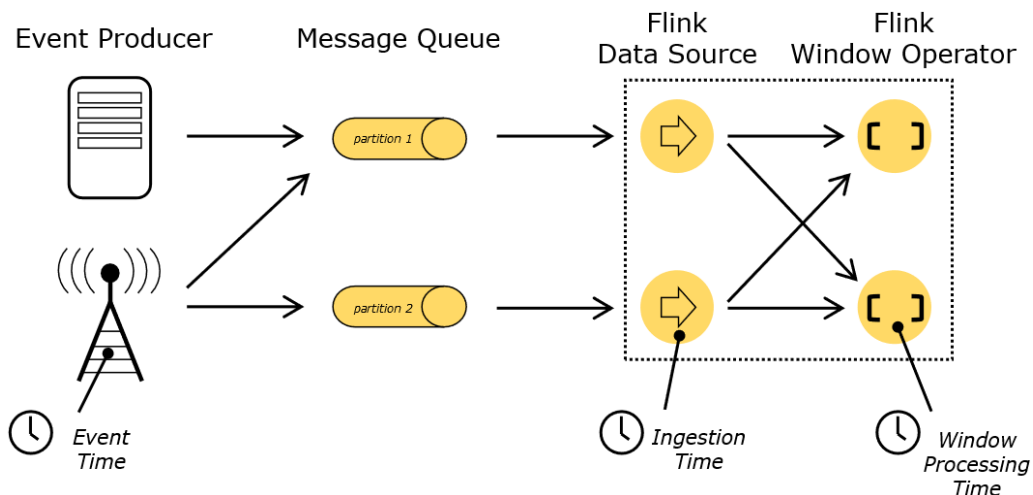


图 Flink 时间概念

Event Time: 是事件创建的时间。它通常由事件中的时间戳描述,例如采集的日志数据中,每一条日志都会记录自己的生成时间,Flink 通过时间戳分配器访问事件时间戳。

【更多 Java、HTML5、Android、python、大数据 资料下载,可访问尚硅谷(中国)官网下载区】

Ingestion Time: 是数据进入 Flink 的时间。

Processing Time: 是每一个执行基于时间操作的算子的本地系统时间，与机器相关，默认的时间属性就是 Processing Time。

例如，一条日志进入 Flink 的时间为 2017-11-12 10:00:00.123，到达 Window 的系统时间为 2017-11-12 10:00:01.234，日志的内容如下：

```
2017-11-02 18:37:15.624 INFO Fail over to rm2
```

对于业务来说，要统计 1min 内的故障日志个数，哪个时间是最有意义的？——eventTime，因为我们要根据日志的生成时间进行统计。

6.2 Window

6.2.1 Window 概述

streaming 流式计算是一种被设计用于处理无限数据集的数据处理引擎，而无限数据集是指一种不断增长的本质的无限的数据集，而 window 是一种切割无限数据为有限块进行处理的手段。

Window 是无限数据流处理的核心，Window 将一个无限的 stream 拆分成有限大小的”buckets”桶，我们可以在这些桶上做计算操作。

6.2.2 Window 类型

Window 可以分成两类：

- CountWindow: 按照指定的数据条数生成一个 Window，与时间无关。
- TimeWindow: 按照时间生成 Window。

对于 TimeWindow，可以根据窗口实现原理的不同分成三类：滚动窗口（Tumbling Window）、滑动窗口（Sliding Window）和会话窗口（Session Window）。

1. 滚动窗口（Tumbling Windows）

将数据依据固定的窗口长度对数据进行切片。

特点：时间对齐，窗口长度固定，没有重叠。

滚动窗口分配器将每个元素分配到一个指定窗口大小的窗口中，滚动窗口有一个固定的大小，并且不会出现重叠。例如：如果你指定了一个 5 分钟大小的滚动窗口，窗口的创建如下图所示：

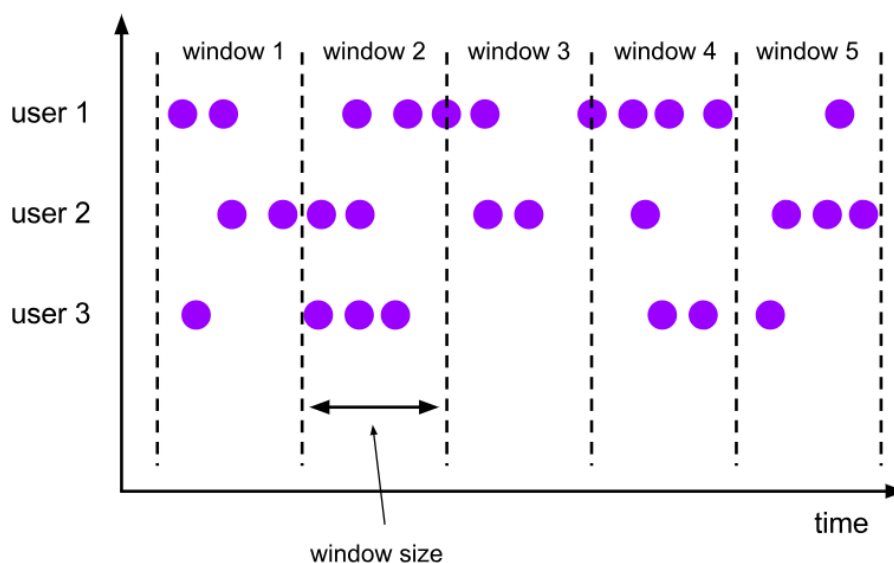


图 滚动窗口

适用场景：适合做 BI 统计等（做每个时间段的聚合计算）。

2. 滑动窗口 (Sliding Windows)

滑动窗口是固定窗口的更广义的一种形式，滑动窗口由固定的窗口长度和滑动间隔组成。

特点：时间对齐，窗口长度固定，有重叠。

滑动窗口分配器将元素分配到固定长度的窗口中，与滚动窗口类似，窗口的大小由窗口大小参数来配置，另一个窗口滑动参数控制滑动窗口开始的频率。因此，滑动窗口如果滑动参数小于窗口大小的话，窗口是可以重叠的，在这种情况下元素会被分配到多个窗口中。

例如，你有 10 分钟的窗口和 5 分钟的滑动，那么每个窗口中 5 分钟的窗口里包含着上个 10 分钟产生的数据，如下图所示：

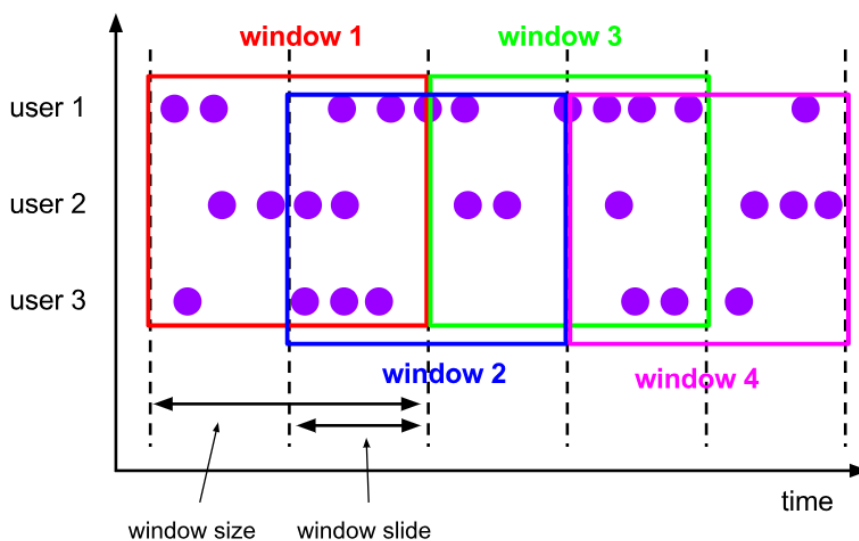


图 滑动窗口

适用场景：对最近一个时间段内的统计（求某接口最近 5min 的失败率来决定是否要报警）。

【更多 Java、HTML5、Android、python、大数据 资料下载，可访问尚硅谷（中国）官网下载区】

3. 会话窗口 (Session Windows)

由一系列事件组合一个指定时间长度的 **timeout** 间隙组成，类似于 **web** 应用的 **session**，也就是一段长时间没有接收到新数据就会生成新的窗口。

特点：时间无对齐。

session 窗口分配器通过 session 活动来对元素进行分组，session 窗口跟滚动窗口和滑动窗口相比，不会有重叠和固定的开始时间和结束时间的情况，相反，**当它在一个固定的时间周期内不再收到元素，即非活动间隔产生，那个这个窗口就会关闭**。一个 session 窗口通过一个 session 间隔来配置，这个 session 间隔定义了非活跃周期的长度，当这个非活跃周期产生，那么当前的 session 将关闭并且后续的元素将被分配到新的 session 窗口中去。

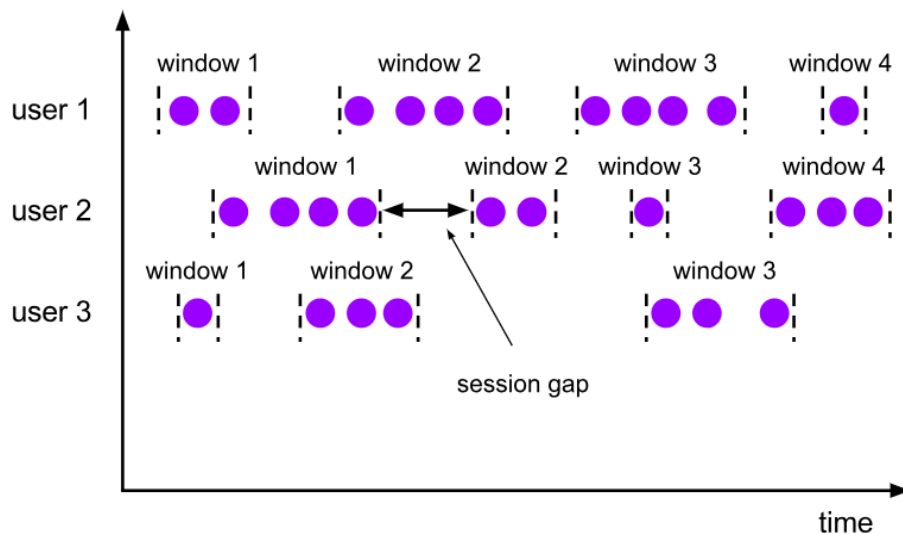


图 会话窗口

6.3 Window API

6.3.1 CountWindow

CountWindow 根据窗口中相同 **key** 元素的数量来触发执行，执行时只计算元素数量达到窗口大小的 **key** 对应的结果。

注意：CountWindow 的 **window_size** 指的是相同 **Key** 的元素的个数，不是输入的所有元素的总数。

1 滚动窗口

默认的 CountWindow 是一个滚动窗口，只需要指定窗口大小即可，当元素数量达到窗口大小时，就会触发窗口的执行。

```
// 获取执行环境
val env = StreamExecutionEnvironment.getExecutionEnvironment

// 创建 SocketSource
```

【更多 Java、HTML5、Android、python、大数据 资料下载，可访问尚硅谷（中国）官网下载区】

```
val stream = env.socketTextStream("localhost", 11111)

// 对 stream 进行处理并按 key 聚合
val streamKeyBy = stream.map(item => (item.split(" ") (0), item.split(" ") (1).toLong)).keyBy(0)

// 引入滚动窗口
// 这里的 5 指的是 5 个相同 key 的元素计算一次
val streamWindow = streamKeyBy.countWindow(5)

// 执行聚合操作
val streamReduce = streamWindow.reduce(
    (item1, item2) => (item1._1, item1._2 + item2._2)
)

// 将聚合数据写入文件
streamReduce.print()

// 执行程序
env.execute("TumblingWindow")
```

2 滑动窗口

滑动窗口和滚动窗口的函数名是完全一致的，只是在传参数时需要传入两个参数，一个是 window_size，一个是 sliding_size。

下面代码中的 sliding_size 设置为了 2，也就是说，每收到两个相同 key 的数据就计算一次，每一次计算的 window 范围是 5 个元素。

```
// 获取执行环境
val env = StreamExecutionEnvironment.getExecutionEnvironment

// 创建 SocketSource
val stream = env.socketTextStream("localhost", 11111)

// 对 stream 进行处理并按 key 聚合
val streamKeyBy = stream.map(item => (item.split(" ") (0), item.split(" ") (1).toLong)).keyBy(0)

// 引入滚动窗口
// 当相同 key 的元素个数达到 2 个时，触发窗口计算，计算的窗口范围为 5
val streamWindow = streamKeyBy.countWindow(5,2)

// 执行聚合操作
val streamReduce = streamWindow.reduce(
    (item1, item2) => (item1._1, item1._2 + item2._2)
)

// 将聚合数据写入文件
streamReduce.print()

// 执行程序
env.execute("TumblingWindow")
}
```

6.3.2 TimeWindow

TimeWindow 是将指定时间范围内的所有数据组成一个 window，一次对一个 window 里面的所有数据进行计算。

【更多 Java、HTML5、Android、python、大数据 资料下载，可访问尚硅谷（中国）官网下载区】

1. 滚动窗口

Flink 默认的时间窗口根据 Processing Time 进行窗口的划分, 将 Flink 获取到的数据根据进入 Flink 的时间划分到不同的窗口中。

```
// 获取执行环境
val env = StreamExecutionEnvironment.getExecutionEnvironment

// 创建 SocketSource
val stream = env.socketTextStream("localhost", 11111)

// 对 stream 进行处理并按 key 聚合
val streamKeyBy = stream.map(item => (item, 1)).keyBy(0)

// 引入时间窗口
val streamWindow = streamKeyBy.timeWindow(Time.seconds(5))

// 执行聚合操作
val streamReduce = streamWindow.reduce(
    (item1, item2) => (item1._1, item1._2 + item2._2)
)

// 将聚合数据写入文件
streamReduce.print()

// 执行程序
env.execute("TumblingWindow")
```

时间间隔可以通过 Time.milliseconds(x), Time.seconds(x), Time.minutes(x) 等其中的一个来指定。

2. 滑动窗口 (SlidingEventTimeWindows)

滑动窗口和滚动窗口的函数名是完全一致的, 只是在传参数时需要传入两个参数, 一个是 window_size, 一个是 sliding_size。

下面代码中的 sliding_size 设置为了 2s, 也就是说, 窗口每 2s 就计算一次, 每一次计算的 window 范围是 5s 内的所有元素。

```
// 获取执行环境
val env = StreamExecutionEnvironment.getExecutionEnvironment

// 创建 SocketSource
val stream = env.socketTextStream("localhost", 11111)

// 对 stream 进行处理并按 key 聚合
val streamKeyBy = stream.map(item => (item, 1)).keyBy(0)

// 引入滚动窗口
val streamWindow = streamKeyBy.timeWindow(Time.seconds(5), Time.seconds(2))

// 执行聚合操作
val streamReduce = streamWindow.reduce(
    (item1, item2) => (item1._1, item1._2 + item2._2)
)

// 将聚合数据写入文件
streamReduce.print()

// 执行程序
env.execute("TumblingWindow")
```

时间间隔可以通过 `Time.milliseconds(x)`, `Time.seconds(x)`, `Time.minutes(x)` 等其中的一个来指定。

6.3.3 Window Reduce

WindowedStream → DataStream: 给 window 赋一个 reduce 功能的函数，并返回一个聚合的结果。

```
// 获取执行环境
val env = StreamExecutionEnvironment.getExecutionEnvironment

// 创建 SocketSource
val stream = env.socketTextStream("localhost", 11111)

// 对 stream 进行处理并按 key 聚合
val streamKeyBy = stream.map(item => (item, 1)).keyBy(0)

// 引入时间窗口
val streamWindow = streamKeyBy.timeWindow(Time.seconds(5))

// 执行聚合操作
val streamReduce = streamWindow.reduce(
    (item1, item2) => (item1._1, item1._2 + item2._2)
)

// 将聚合数据写入文件
streamReduce.print()

// 执行程序
env.execute("TumblingWindow")
```

6.3.4 Window Fold

WindowedStream → DataStream: 给窗口赋一个 fold 功能的函数，并返回一个 fold 后的结果。

```
// 获取执行环境
val env = StreamExecutionEnvironment.getExecutionEnvironment

// 创建 SocketSource
val stream = env.socketTextStream("localhost", 11111, '\n', 3)

// 对 stream 进行处理并按 key 聚合
val streamKeyBy = stream.map(item => (item, 1)).keyBy(0)

// 引入滚动窗口
val streamWindow = streamKeyBy.timeWindow(Time.seconds(5))

// 执行 fold 操作
val streamFold = streamWindow.fold(100){
    (begin, item) =>
        begin + item._2
}

// 将聚合数据写入文件
streamFold.print()
```

【更多 Java、HTML5、Android、python、大数据 资料下载，可访问尚硅谷（中国）官网下载区】

```
// 执行程序
env.execute("TumblingWindow")
```

6.3.5 Aggregation on Window

WindowedStream → **DataStream**: 对一个 window 内的所有元素做聚合操作。
min 和 minBy 的区别是 min 返回的是最小值，而 minBy 返回的是包含最小值字段的元素(同样的原理适用于 max 和 maxBy)。

```
// 获取执行环境
val env = StreamExecutionEnvironment.getExecutionEnvironment

// 创建 SocketSource
val stream = env.socketTextStream("localhost", 11111)

// 对 stream 进行处理并按 key 聚合
val streamKeyBy = stream.map(item => (item.split(" ")(0), item.split(" ")(1))).keyBy(0)

// 引入滚动窗口
val streamWindow = streamKeyBy.timeWindow(Time.seconds(5))

// 执行聚合操作
val streamMax = streamWindow.max(1)

// 将聚合数据写入文件
streamMax.print()

// 执行程序
env.execute("TumblingWindow")
```

第七章 EventTime 与 Window

7.1 EventTime 的引入

在 Flink 的流式处理中，绝大部分的业务都会使用 **eventTime**，一般只在 **eventTime** 无法使用时，才会被迫使用 **ProcessingTime** 或者 **IngestionTime**。

如果要使用 EventTime，那么需要引入 EventTime 的时间属性，引入方式如下所示：

```
val env = StreamExecutionEnvironment.getExecutionEnvironment

// 从调用时刻开始给 env 创建的每一个 stream 追加时间特征
env.setStreamTimeCharacteristic(TimeCharacteristic.EventTime)
```

7.2 Watermark

7.2.1 基本概念

我们知道，流处理从事件产生，到流经 source，再到 operator，中间是有一个过程和时间的，虽然大部分情况下，流到 operator 的数据都是按照事件产生的时间顺序来的，但是也不排除由于网络等原因，导致乱序的产生，所谓乱序，就是指 Flink 接收到的事件的先后顺序不是严格按照事件的 Event Time 顺序排列的。

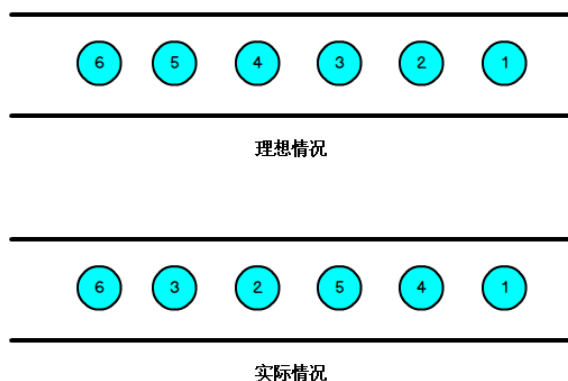


图 数据的乱序

那么此时出现一个问题，一旦出现乱序，如果只根据 eventTime 决定 window 的运行，我们不能明确数据是否全部到位，但又不能无限期的等下去，此时必须要有个机制来保证一个特定的时间后，必须触发 window 去进行计算了，这个特别的机制，就是 Watermark。

Watermark 是一种衡量 Event Time 进展的机制，它是数据本身的一个隐藏属性，数据本身携带着对应的 Watermark。

Watermark 是用于处理乱序事件的，而正确的处理乱序事件，通常用 Watermark 机制结合 window 来实现。

数据流中的 Watermark 用于表示 timestamp 小于 Watermark 的数据，都已经到达了，因此，window 的执行也是由 Watermark 触发的。

Watermark 可以理解成一个延迟触发机制，我们可以设置 Watermark 的延时长 t ，每次系统会校验已经到达的数据中最大的 maxEventTime ，然后认定 eventTime 小于 $\text{maxEventTime} - t$ 的所有数据都已经到达，如果有窗口的停止时间等于 $\text{maxEventTime} - t$ ，那么这个窗口被触发执行。

有序流的 Watermarker 如下图所示：（Watermark 设置为 0）

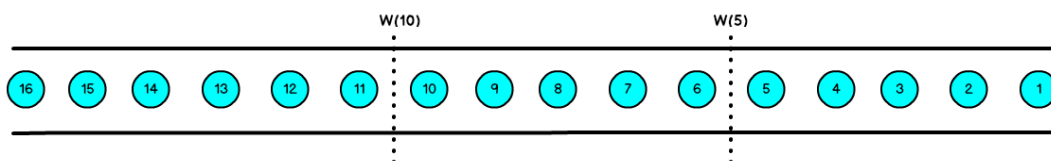


图 有序数据的 Watermark

乱序流的 Watermarker 如下图所示：（Watermark 设置为 2）

【更多 Java、HTML5、Android、python、大数据 资料下载，可访问尚硅谷（中国）官网下载区】

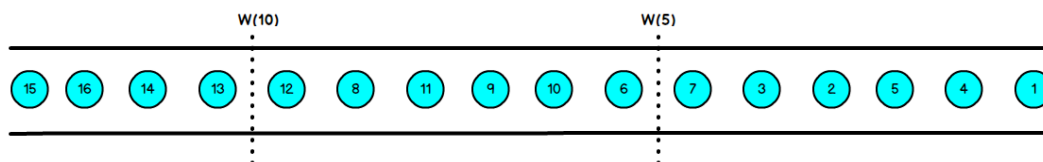


图 无序数据的 Watermark

当 Flink 接收到每一条数据时，都会产生一条 Watermark，这条 Watermark 就等于当前所有到达数据中的 `maxEventTime` - 延迟时长，也就是说，Watermark 是由数据携带的，一旦数据携带的 Watermark 比当前未触发的窗口的停止时间要晚，那么就会触发相应窗口的执行。由于 Watermark 是由数据携带的，因此，如果运行过程中无法获取新的数据，那么没有被触发的窗口将永远都不被触发。

上图中，我们设置的允许最大延迟到达时间为 2s，所以时间戳为 7s 的事件对应的 Watermark 是 5s，时间戳为 12s 的事件的 Watermark 是 10s，如果我们的窗口 1 是 1s~5s，窗口 2 是 6s~10s，那么时间戳为 7s 的事件到达时的 Watermark 恰好触发窗口 1，时间戳为 12s 的事件到达时的 Watermark 恰好触发窗口 2。

7.2.2 Watermark 的引入

```
val env = StreamExecutionEnvironment.getExecutionEnvironment

// 从调用时刻开始给 env 创建的每一个 stream 追加时间特征
env.setStreamTimeCharacteristic(TimeCharacteristic.EventTime)

val stream = env.socketTextStream("localhost", 11111).assignTimestampsAndWatermarks(
    new BoundedOutOfOrdernessTimestampExtractor[String](Time.milliseconds(200)) {
        override def extractTimestamp(t: String): Long = {
            // EventTime 是日志生成时间，我们从日志中解析 EventTime
            t.split(" ")[0].toLong
        }
    }
)
```

7.3 EvnetTimeWindow API

当使用 EventTimeWindow 时，所有的 Window 在 EventTime 的时间轴上进行划分，也就是说，在 Window 启动后，会根据初始的 EventTime 时间每隔一段时间划分一个窗口，如果 Window 大小是 3 秒，那么 1 分钟内会把 Window 划分为如下的形式：

```
[00:00:00,00:00:03)
[00:00:03,00:00:06)
...
[00:00:57,00:01:00)
```

如果 Window 大小是 10 秒，则 Window 会被分为如下的形式：

```
[00:00:00,00:00:10)
[00:00:10,00:00:20)
...
[00:00:50,00:01:00)
```

【更多 Java、HTML5、Android、python、大数据 资料下载，可访问尚硅谷（中国）官网下载区】

注意，窗口是左闭右开的，形式为： $[\text{window_start_time}, \text{window_end_time})$ 。

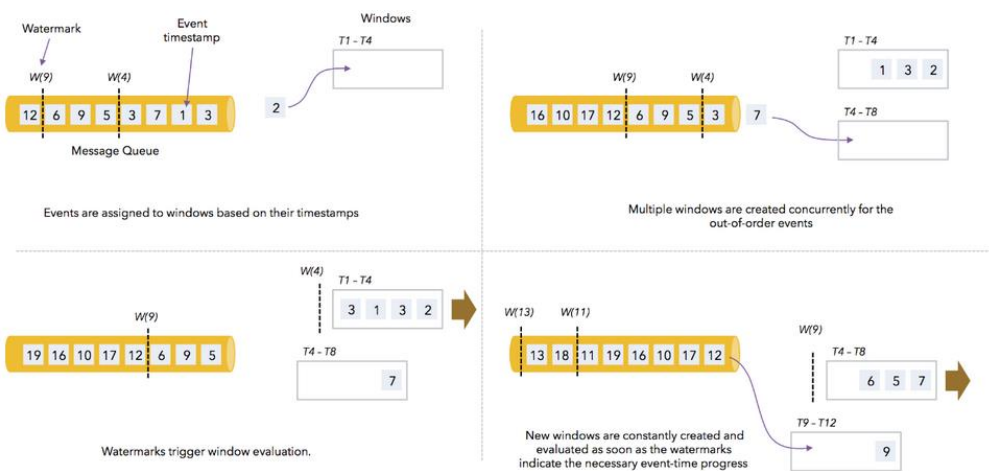
Window 的设定无关数据本身，而是系统定义好了的，也就是说，Window 会一直按照指定的时间间隔进行划分，不论这个 Window 中有没有数据，EventTime 在这个 Window 期间的数据会进入这个 Window。

Window 会不断产生，属于这个 Window 范围的数据会被不断加入到 Window 中，所有未被触发的 Window 都会等待触发，只要 Window 还没触发，属于这个 Window 范围的数据就会一直被加入到 Window 中，直到 Window 被触发才会停止数据的追加，而当 Window 触发之后才接受到的属于被触发 Window 的数据会被丢弃。

Window 会在以下的条件满足时被触发执行：

- **watermark 时间 \geq window_end_time;**
- **在 $[\text{window_start_time}, \text{window_end_time})$ 中有数据存在。**

我们通过下图来说明 Watermark、EventTime 和 Window 的关系。



7.3.1 滚动窗口（TumblingEventTimeWindows）

```
// 获取执行环境
val env = StreamExecutionEnvironment.getExecutionEnvironment
env.setStreamTimeCharacteristic(TimeCharacteristic.EventTime)

// 创建 SocketSource
val stream = env.socketTextStream("localhost", 11111)

// 对 stream 进行处理并按 key 聚合
val streamKeyBy = stream.assignTimestampsAndWatermarks(
  new BoundedOutOfOrdernessTimestampExtractor[String](Time.milliseconds(3000)) {
    override def extractTimestamp(element: String): Long = {
      val sysTime = element.split(" ")[0].toLong
      println(sysTime)
      sysTime
    }
  }).map(item => (item.split(" ")[1], 1)).keyBy(0)

// 引入滚动窗口
val streamWindow = streamKeyBy.window(TumblingEventTimeWindows.of(Time.seconds(10)))

// 执行聚合操作
val streamReduce = streamWindow.reduce(
  (item1, item2) => (item1._1, item1._2 + item2._2)
```

【更多 Java、HTML5、Android、python、大数据 资料下载，可访问尚硅谷（中国）官网下载区】

```

)

// 将聚合数据写入文件
streamReduce.print

// 执行程序
env.execute("TumblingWindow")

```

结果是按照 **Event Time** 的时间窗口计算得出的，而无关系统的时间（包括输入的快慢）。

7.3.2 滑动窗口（SlidingEventTimeWindows）

```

// 获取执行环境
val env = StreamExecutionEnvironment.getExecutionEnvironment
env.setStreamTimeCharacteristic(TimeCharacteristic.EventTime)

// 创建 SocketSource
val stream = env.socketTextStream("localhost", 11111)

// 对 stream 进行处理并按 key 聚合
val streamKeyBy = stream.assignTimestampsAndWatermarks(
    new BoundedOutOfOrdernessTimestampExtractor[String](Time.milliseconds(0)) {
        override def extractTimestamp(element: String): Long = {
            val sysTime = element.split(" ")[0].toLong
            println(sysTime)
            sysTime
        }
    }).map(item => (item.split(" ")[1], 1)).keyBy(0)

// 引入滚动窗口
val streamWindow = streamKeyBy.window(SlidingEventTimeWindows.of(Time.seconds(10),
    Time.seconds(5)))

// 执行聚合操作
val streamReduce = streamWindow.reduce(
    (item1, item2) => (item1._1, item1._2 + item2._2)
)

// 将聚合数据写入文件
streamReduce.print

// 执行程序
env.execute("TumblingWindow")

```

7.3.3 会话窗口（EventTimeSessionWindows）

相邻两次数据的 **EventTime** 的时间差超过指定的时间间隔就会触发执行。如果加入 Watermark，那么当触发执行时，所有满足时间间隔而还没有触发的 Window 会同时触发执行。

```

// 获取执行环境
val env = StreamExecutionEnvironment.getExecutionEnvironment
env.setStreamTimeCharacteristic(TimeCharacteristic.EventTime)

// 创建 SocketSource
val stream = env.socketTextStream("localhost", 11111)

```

【更多 Java、HTML5、Android、python、大数据 资料下载，可访问尚硅谷（中国）官网下载区】

```
// 对 stream 进行处理并按 key 聚合
val streamKeyBy = stream.assignTimestampsAndWatermarks(
  new BoundedOutOfOrdernessTimestampExtractor[String](Time.milliseconds(0)) {
    override def extractTimestamp(element: String): Long = {
      val sysTime = element.split(" ")[0].toLong
      println(sysTime)
      sysTime
    }
  }).map(item => (item.split(" ")[1], 1)).keyBy(0)

// 引入滚动窗口
val streamWindow = streamKeyBy.window(EventTimeSessionWindows.withGap(Time.seconds(5)))

// 执行聚合操作
val streamReduce = streamWindow.reduce(
  (item1, item2) => (item1._1, item1._2 + item2._2)
)

// 将聚合数据写入文件
streamReduce.print

// 执行程序
env.execute("TumblingWindow")
```

第八章 总结

Flink 是一个真正意义上的流计算引擎，在满足低延迟和低容错开销的基础之上，完美的解决了 exactly-once 的目标，真是由于 Flink 具有诸多优点，越来越多的企业开始使用 Flink 作为流处理框架，逐步替换掉了原本的 Storm 和 Spark 技术框架。

本课程对 Flink 的基本概念进行了讲解，并对 Flink 的流处理进行了较为详细的解析，希望能够帮助同学们入门 Flink 技术框架，让同学们能够使用 Flink 完成更为完美的流式处理任务。