# Learning Apache Flink

## 1. Introduction to Apache Flink

随着分布式技术的不断发展，工程师们正试图将这些技术推向极限。早期，人们寻找更快、更廉价的方式处理数据。因此Hadoop被提出来了。人们使用Hadoop，并使用Hadoop生态系统作为他们的ETL工具。现在许多公司已经开始使用Hadoop，另外一些人需要以流式方式处理数据，因此刀子一些像Spark和Flink这些技术出现。因其有更快的处理数据的能力，快速的扩展性，支持机器学习和图计算的特点，在开发社区已经非常受欢迎。

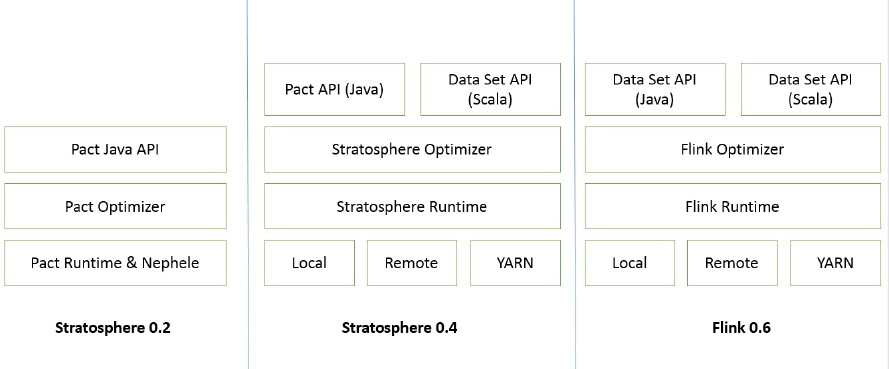
有些人在他们的日常生活中已经使用Spark了，或许他们想知道为什么我们还需要Flink？这种想法人们很自然的被人提出来。让我来用简洁的话来回到这个问题。第一点是Flink是真正的流处理引擎，它不是像Spark流处理引擎那样以更快的速度处理微批量数据。它把微批量数据的处理作为流式处理的一种特殊情况。同样通过本书我们会发现许多的不同。

这本书是关于更先进的技术中的Flink。在本章，我们将讨论一下主题：

* 历史
* 结构
* 分布式执行
* 特点
* 快速开始配置
* 集群配置
* 运行一个简单的应用程序

### History

Flink起源于柏林地区的大学研究的一个名叫Stratosphere项目，该项目的主要目标是建立下一代大数据分析平台。在2016.4.16日作为Apache旗下一个孵化项目。Stratosphere初始版本主要依据于Nephele的<http://stratosphere.eu/assets/papers/Nephele_09.pdf>论文。以下图展示了Stratosphere随着时间是怎么进化的。

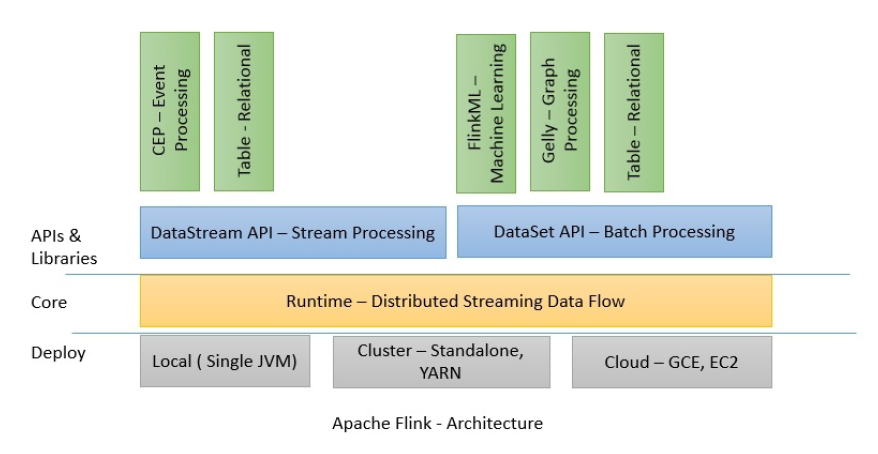


Stratosphere 最开始的版本主要包括运行时环境，优化器以及Java API。之后，随着该平台的不断成熟，它支持运行在本地和yarn上，从0.6版本以后，Stratosphere 被重新命名为Flink。之后的Flink版本不断聚焦于批量处理、流式处理、图计算、机器学习等。

Flink在0.7版本，一个非常重要的特点-streaming API被启动。Flink以前的版本仅仅支持Java API，在之后的版本也支持Scala API。让我们在下一节看看现在Flink的结构。

### Architecture

Flink 1.x 之后主要的结构由部署、主要的处理引擎以及APIs组成。我们可以很轻松的比较最后的版本和Stratosphere的结构来发现他的进化。以下图展示了组件，APIs和相关的库：



Flink具有层次结构，每一个组件都是特殊层中的一部分。每一层都是简历在其它层之上的抽象。Flink可以运行在本机、Yarn集群或者云上。运行时环境是Flink主要的数据处理引擎。它将通过API别写的程序转换成JobGraph。JobGraph是一个并行化的数据流，包含一些生产和消费数据的流任务

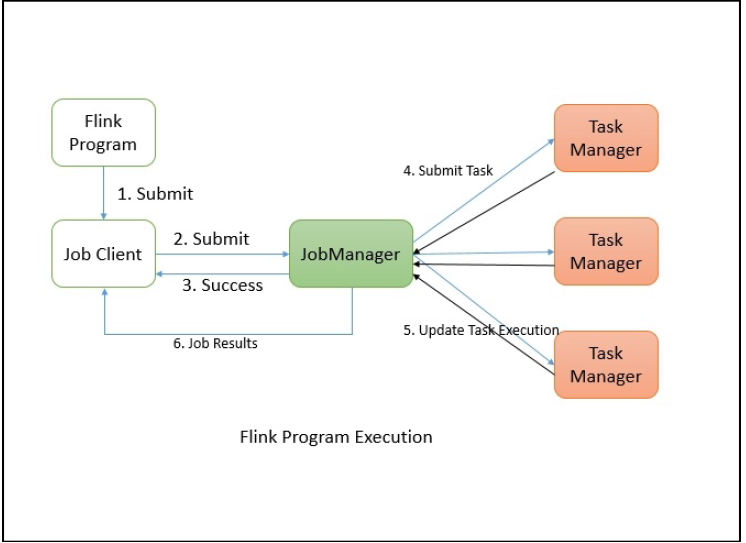
编程人员利用DataStream和DataSet APIs可以定义Job。JobGraph是在编写完成的程序中生成的。一旦JogGraph生成，DataSet API通过优化器生成优化的执行计划与此同时DataStream API通过流建立一个有效的执行计划。

这些优化的JobGraph然后会被提交到部署模式中的执行者。你可以选择部署在本地、远程或者Yarn集群。如果你已经拥有一个运行的Hadoop集群，最好使用Yarn来进行部署。

#### Distributed execution

Flink分布式执行主要由两个非常重要的进程组成：master和worker。当一个Flink程序在执行的时候，许多的进程参与执行，它们是Job Manager，Task Manager，Job Client。

以下图展示了一个Flink程序的执行：



Flink程序需要被提交打Job Client。Job Clinet将该作业提交给Job Manager，Job Manager的主要责任是负责资源的分配和作业的执行，最主要的事情是分配需要的资源，一旦资源被分配，task将被分配到各自的Task Manager。当接收到task，Task Manager 初始化一个进程去执行。在进程执行的过程中，Task Manager 不断汇报状态（启动，执行，完成）给Job Manager，一旦job执行完成，最后的结果将会返回给client。

#### Job Manager

master进程又名Job Manager，协调和管理程序的执行。它主要的职责包括作业的调度，检查点的管理，容错回复等等。

并行化环境下有许多master，他们共享他们的职责，以此来实现高可用。只有一个master是主，如果主master宕机了，会从其他的standby master节点选取一个作为leader master。

Job Manager由以下几个主要的组件组成：

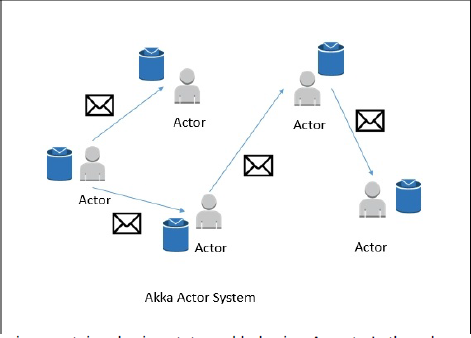
* Actor System
* Scheduler
* Check pointing

Flink内部使用Akka actor system 在job manager和task managers之间进行通讯。

##### Actor System

一个Actor System 是一个由不同角色的actors组成的容器。它提供例如调度，配置，日志等服务。它主要包含一个线程池，所有的actors都是从该线程池中进行初始化的。所有的actors都处于同一个层级结构。每一个新的actor都会给其分配一个parent。所有的actors通过消息系统进行交流。每一个actor都有自己的邮箱，它们通过该邮箱来读取信息。如果actors是local模式，那么消息将会在内存中进行共享，如果是远程模式，那么消息将会通过RPC进行调用。

每一个parent都需要对其所有的孩子进行监督。如果其中一个孩子出现问题，parent将会知道。如果孩子可以自己处理，那么parent将会重启该孩子，如果该孩子不能处理它的问题，那么它将会把该问题提升给自己的parent。



在Flink内部，一个actor是一个容器，它拥有自己的状态和行为。如果邮箱收到消息，他们他们将会不断的处理消息。它们的状态和行为是由他们所收到的消息决定的。

##### Scheduler

Flink中的执行者被定义为task slots。每一个task Manager需要管理一个或者多个task slots。在内部，Flink通过SlotSharingGroup和ColocationGroup决定哪些任务需要共享他们的task slot，哪些任务需要放置在特定的task slots。

##### Check point

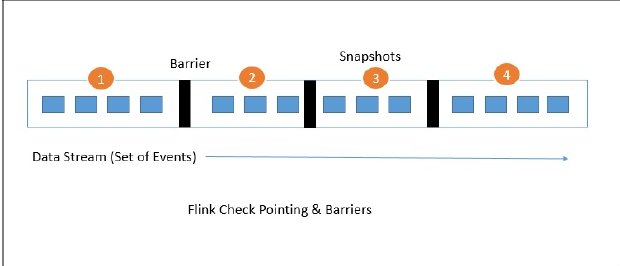
Flink通过check point来主要实现容错。它始终为分布式流数据和执行状态提供一致性快照。它受Chandy-Lamport算法的启发，但是已经根据Flink的需求进行了修改。更多关旭Chandy-Lamport算法间：<http://research.microsoft.com/enus/um/people/lamport/pubs/chandy.pdf>。

更精确的关于快照的实现基于以下文章：轻量级分布式数据流异步快照（<http://arxiv.org/abs/1506.08603>）。

容错机制不断地为数据流创造轻量级的快照。因此他们能够不断工作而不会过度负载。一般数据流的状态被保存在配置指明的地方，例如HDFS等地方。

在发生任何错误的情况下，Flink停止executors并重新启动它们，它们会从最后的检查点进行开始执行。

Stream barriers 是Flink 快照的核心元素，它们被摄入数据但不影响流浪。Barriers重来不会超越记录，它们将一组记录合并成一个快照。每一个barrier都有一个唯一的ID。以下图展示了barriers怎么被嵌入到数据流中进行快照的：

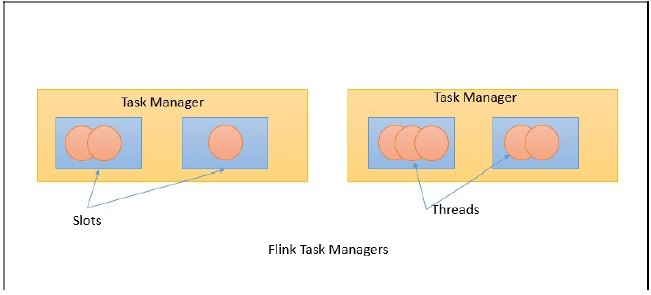


每一个快照状态都会向job Manager的检查点协调器进行汇报。当绘制快照时，Flink都会进行记录的校验以防止由于任何错误导致的数据重复处理。即使毫秒级的延迟是不可接受的，我们也可以选择在单个记录处理中选择低延迟。默认情况下，Flink只处理一次记录。 如果任何应用程序需要低延迟并且至少有一次处理就可以，我们可以关闭该触发器。 这将跳过校验并提高延迟。

#### Task Manager

Task managers是一些工作节点，它们在JVM中使用一个或者多个线程来执行任务。

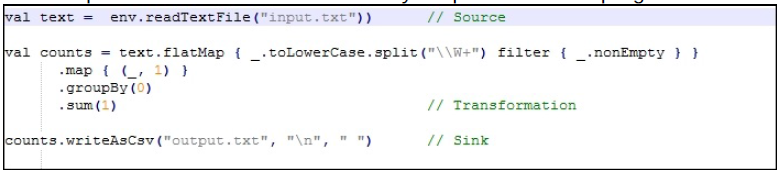
并行化的任务执行主要由每一个task manager中可用的task slots决定。每一个任务代表了被分配资源的集合到task slot上的映射。例如。如果一个task manager有4个slots，它们每一个slot将会得到task manager 内存的25%。每一个task slot可能有一个或者多个线程。在同一个slot中的线程之间共享他们的JVM。在同一个JVM中的task共享他们的TCP链接和心跳。



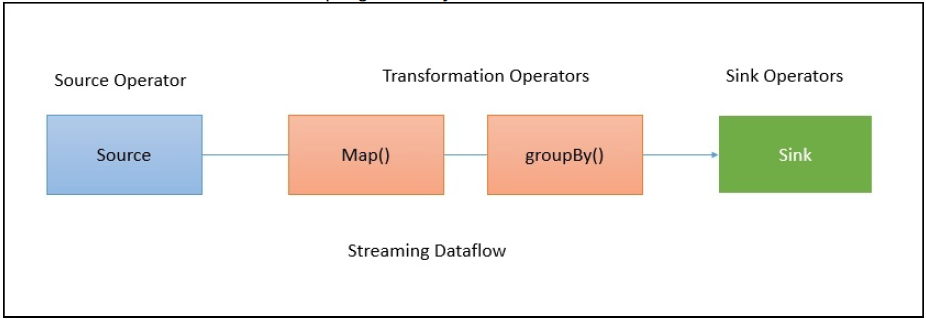
#### Job Client

Job Client不是Flink程序执行的一部分，但是他们是程序执行的开始点。Job Client主要负责接收用户提交的程序并创建数据流，然后提交该数据流到job manager去执行。当执行完成以后，job client将处理的结果返回给用户。

一个数据流是一个执行计划，考虑一个最简单的单词统计程序：

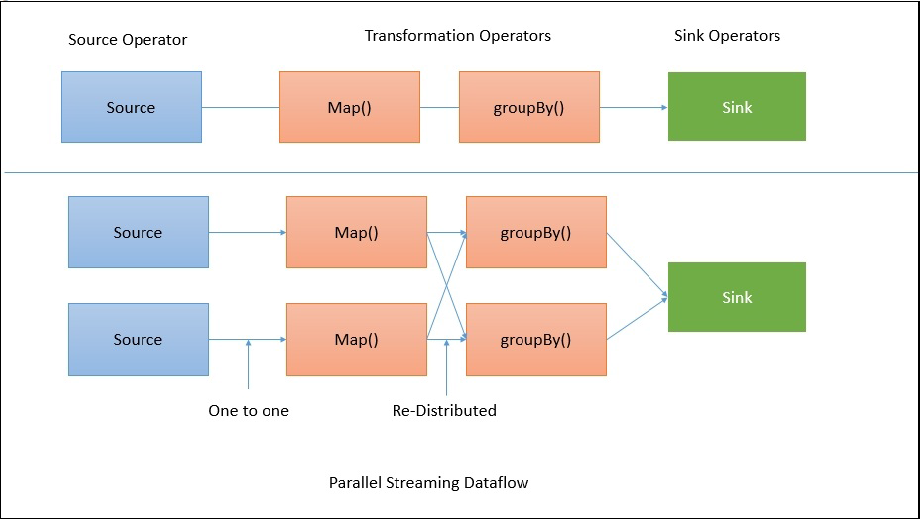


当client 接收到用户的程序，他们将其转换成数据流。上述程序的数据流图例如这个：



上图展示了一个程序一个程序怎么被转化成一个数据流的。模式数据流是分布式，并行化的。对于并行化的数据处理，Flink分解这些算子和流，算子被分解成子任务，流可以进行一对一分配或者重新分配。

数据流通过source到map算子由于不需要shuffle所以是直接分配，但是groupby()算子由于需要收集结果导致流重新分配。



### Features

#### Hight performance

Flink 可以实现高可用和低延迟，不像其他流式处理框架，例如spark，你不需要配置多种配置就可以达到更好的效果，flink相比于其他流处理框架，他的管道流处理给了他更好的性能。

#### Exactly-once stateful computation

在上一个部分我们已经讨论了，flink分布式的检查点机制保证饿了每一个记录精准一次处理。在高吞吐量应用的情况下，Flink为我们提供了一个开关，允许至少进行一次处理。

#### Flexible streaming windows

Flink支持数据驱动的windows，意味着我们可以根据时间，计数，会话来设计一个窗口，一个window也可以被我们自定义来检测时间流中特定的模式。

#### Fault tolerance

Flink的分布式轻量级快照机制有助于实现高度的容错性。 它允许Flink提供高吞吐量性能并保证交付。

#### Memory managerment

Flink在JVM内部提供自己的内存管理，使其独立于Java的默认垃圾收集器。 它通过使用散列，索引，缓存和排序有效地进行内存管理。

#### Optimizer

Flink的批量数据处理API经过优化，以避免诸如洗牌，分类等耗费内存的操作。 它还确保使用缓存以避免繁重的磁盘IO操作。

#### Stream and batch in one platform

Flink为批处理和流数据处理提供API。 因此，一旦您设置了Flink环境，它可以轻松托管流和批处理应用程序。 事实上，Flink的工作原理是Streaming，并将批处理视为流媒体的特例。

#### Libraries

Flink拥有丰富的库，可用于机器学习，图形处理，关系数据处理等。 由于其架构，执行复杂的事件处理和警报非常容易。 我们将在随后的章节中更多地了解这些库。

#### Event time semantics

Flink支持事件时间语义。 这有助于处理事件无序到达的流。 有时事件可能会延迟。 Flink的架构允许我们根据时间，计数和会话来定义窗口，这有助于处理这种情况

### Quick start setup

现在让我们来理解flink的结构方面的细节和其编程模型，让我们通过快速的配置来探索这些。Flink可以工作在windows和linux机器上。

首先我们需要下载flink二进制安装包，可以通过<http://flink.apache.org/downloads.html>进行下载。

在下载页面，你可以看到多种选项：



为了安装flink，你并不需要安装hadoop，但是当你需要连接hadoop的时候，为了兼容性，你需要下载对应的二进制版本。

#### Pre-requisite

Flink 需要java的支持，所以在你启动flink之前，需要保证java已经安装。

#### Installing on windows

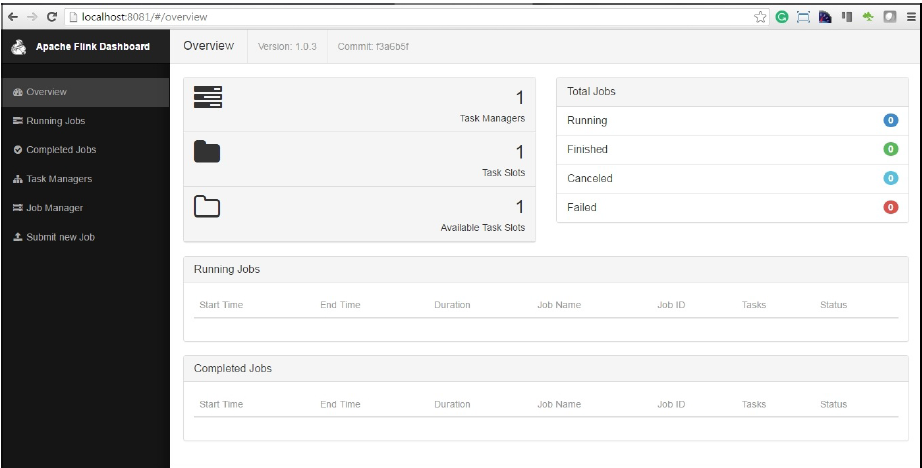
Flink安装是非常简单，仅仅需要解压文件，然后进入文件夹执行start-local.bat:

>cd flink-1.1.4

>bin\start-local.bat

然后你将会看到本地flink已经启动了。

你也可以通过web ui进行查看：<http://localhost:8081/>



你可以通过cltr+C进行停止flink。

#### Installing on Linux

与Windows安装类似，在linux上安装flink非常简单，我们需要下载二进制安装包，然后解压：

$sudo tar -xzf flink-1.1.4-bin-hadoop27-scala\_2.11.tgz

$cd flink-1.1.4

$bin/start-local.sh

与windows类似，你需要保证java已经在本机安装了，现在我们可以提交flink作业了，在linux上停止flink。我们可以执行写一下命令：

$bin/stop-local.sh

### Cluster setup

#### 待完成

最后来写，由于机器不足，不能实践。

### Running sample application

Flink二进制安装包中包含了一个简单应用的例子，让我们从这个简单的应用wordcount开始，现在我们运行该流应用，他通过netcat服务器特定端口读取数据。

因此我们首先需要通过以下命令在netcat服务器9000端口启动服务：

nc –l 9000

现在netcat服务器开始监听9000端口，当你键入的东西都会被发送到flink程序。

现在我们启动该流程序来监听netcat服务器，以下是启动命令：

bin/flink run examples/streaming/SocketTextStreamWordCount.jar --

hostname localhost --port 9000

08/06/2016 10:32:40 Job execution switched to status RUNNING.

08/06/2016 10:32:40 Source: Socket Stream -> Flat Map(1/1)

switched to SCHEDULED

08/06/2016 10:32:40 Source: Socket Stream -> Flat Map(1/1)

switched to DEPLOYING

08/06/2016 10:32:40 Keyed Aggregation -> Sink: Unnamed(1/1)

switched to SCHEDULED

08/06/2016 10:32:40 Keyed Aggregation -> Sink: Unnamed(1/1)

switched to DEPLOYING

08/06/2016 10:32:40 Source: Socket Stream -> Flat Map(1/1)

switched to RUNNING

08/06/2016 10:32:40 Keyed Aggregation -> Sink: Unnamed(1/1)

switched to RUNNING

这将会启动flink 作业的执行，现在我们在netcat控制台键入一些东西，flink程序将会处理它。

例如，在netcat server服务器上输入一下内容：

$nc -l 9000

hi Hello

Hello World

This distribution includes cryptographic software. The country in

which you currently reside may have restrictions on the import,

possession, use, and/or re-export to another country, of

encryption software. BEFORE using any encryption software, please

check your country's laws, regulations and policies concerning the

import, possession, or use, and re-export of encryption software,

to

see if this is permitted. See <http://www.wassenaar.org/> for

more

information.

你可以通过日志验证输出：

$ tail -f flink-\*-taskmanager-\*-flink-instance-\*.out

==> flink-root-taskmanager-0-flink-instance-1.out <==

(see,2)

(http,1)

(www,1)

(wassenaar,1)

(org,1)

(for,1)

(more,1)

(information,1)

(hellow,1)

(world,1)

==> flink-root-taskmanager-1-flink-instance-1.out <==

(is,1)

(permitted,1)

(see,2)

(http,1)

(www,1)

(wassenaar,1)

(org,1)

(for,1)

(more,1)

(information,1)

==> flink-root-taskmanager-2-flink-instance-1.out <==

(hello,1)

(worlds,1)

(hi,1)

(how,1)

(are,1)

(you,1)

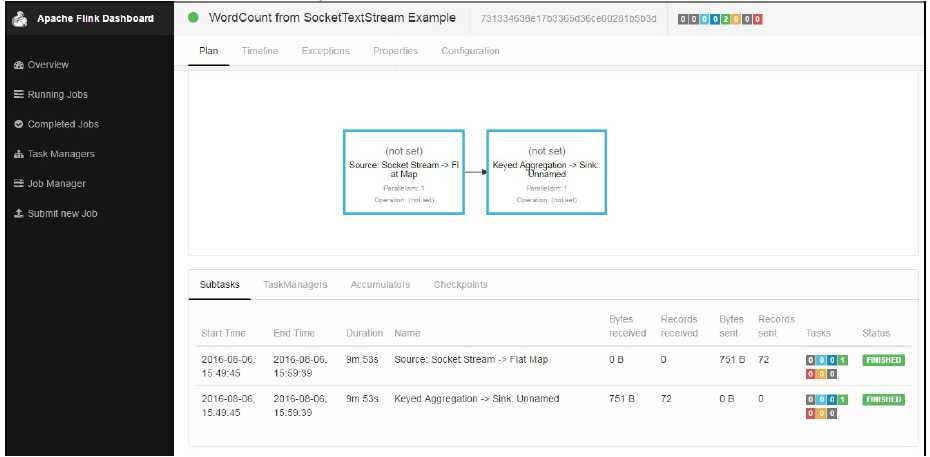
(how,2)

(is,1)

(it,1)

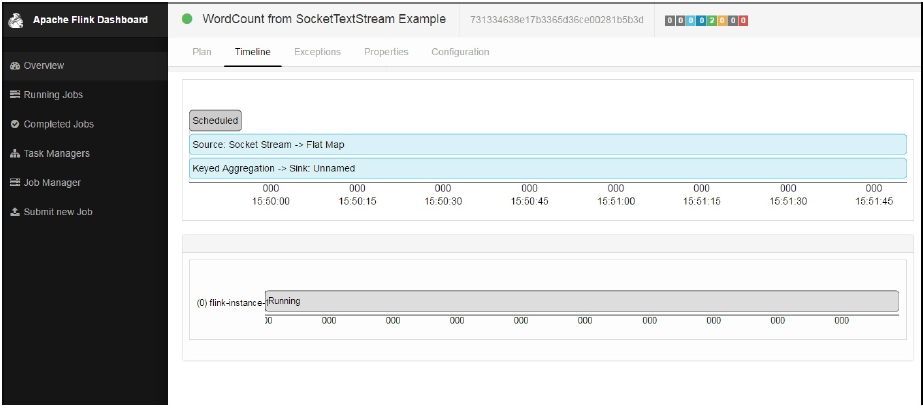
(going,1)

你也可以检查flink的web ui去观察该作业执行的性能。一下快照展示了该执行的数据流计划：



当前的作业，flink有两个算法，第一个算子是source算在，他通过socket流读取数据，第二个算在是转化算子，它聚集单词的计数。

我们也可以看到作业执行的时间线：



### Summary

在本章中，我们讨论了Flink如何开始作为一个大学项目，然后成为一个完整的企业级数据处理平台。 我们研究了Flink架构的细节以及它的流程模型是如何工作的。 我们还学习了如何在本地和集群模式下运行Flink。

在下一章中，我们将学习Flink的Streaming API并查看其细节，以及如何使用该API来解决我们的数据流处理问题。

## 2. Data Processing Using the DataStream API

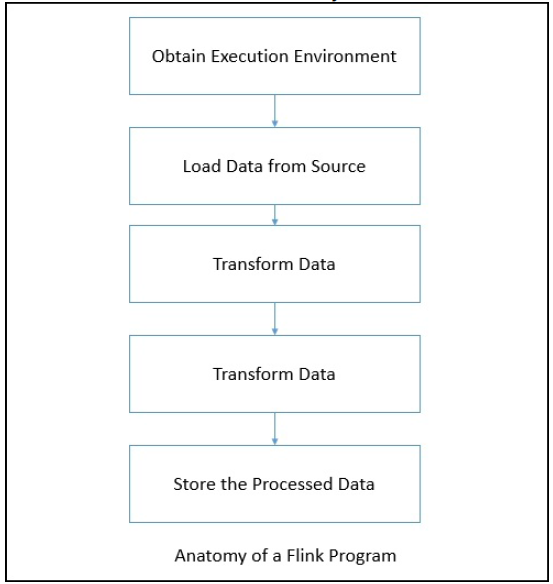
### DataStream API

实时在线分析目前是是一个很重要的问题。许多不同的领域需要实时的处理数据。因此现在有许多的技术尝试去提供这种能力，例如Strom和Spark已经在市场上出现了很长的时间。来自物联网的应用程序需要实时或近实时地存储，处理和分析数据。为了去迎合这些需求，Flink提供了一个流式数据处理API，叫左 DataStream API。

在本章，我们将看到有关DataStream API的详细信息，覆盖一下主题：

* Execution environment
* Data sources
* Transformations
* Data Sinks
* Connectors
* Use case – sensor data analytics

任何一个Flink程序工作的时候包含一下结构：



我们将会看到每一步并且看到我们采用该结构怎样使用DataStream API

#### Execution environment

为了开始写一个flink程序，我们首先需要或者创建一个执行环境。这依赖你你尝试做什么，flink提供：

获取一个存在的flink环境

创建一个本地环境

创建一个远程环境

一般，你只需要使用getExecutionEnvironment()。他就会根据你的上下文创建一个正确的环境。如果你在本地idea运行你的flink程序，他就会自动创建一个本地执行环境。否则，如果你在运行一个jar，flink集群管理者将会使用分布式环境执行该程序。

如果你想创建一个本地或者远程的执行环境，你可以选择使用createLocalEnvironment()和createRemoteEnviroment(String host,int port,String,and .jar files)方法。

### Data sources

Source是flink程序期望获取数据的地方。他是flink程序的第二步。Flink支持一些已经实现的数据源函数，但是也支持用户自定义的。首先让我们看看内置的数据源函数。

#### Socket-based

DataStream API支持从socket中读取数据。你仅仅需要制定主机名和端口他就可以自动从里面读取数据并工作：

socketTextStream(hostname,port);

有也可以选择一个特定的分隔符：

socketTextStream(hostname,port,delimiter);

您还可以指定API尝试获取数据的最大次数：

socketTextStream(hostname,port,delimiter,maxRetry);

#### File-based

你也可以施工用file-based函数从文件中创建一个流，你可以使用readTextFile(String path)从指定的路径中创建一个流，默认情况读取的是TextInputFormat格式的模本并且一行一行的读取成一个字符串。

如果读取的格式不是text格式的，你可以指定使用一下函数：

readFile(FileInputFormat<out> inputFormat,String path)

flink还支持在生成文件时读取文件流，使用readFileStream()函数：

readFileStream(String filePath,long intervalMillins,FileMonitoringFunction.WatchType watch Type)

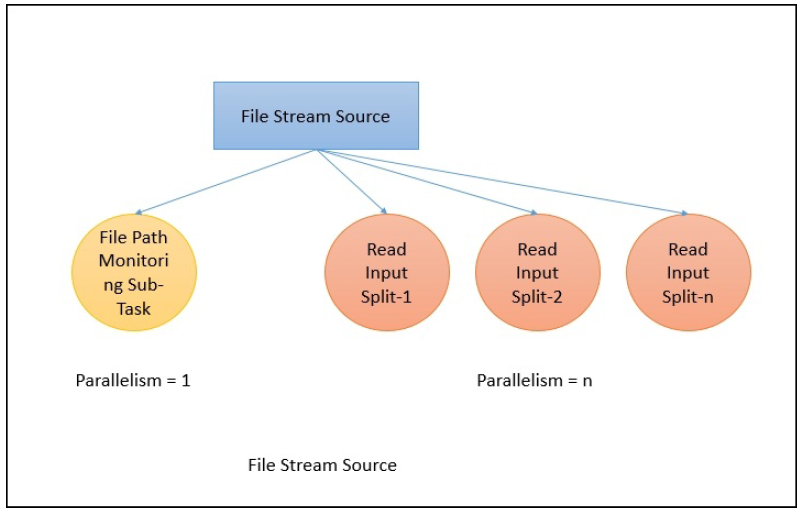
你仅仅需要指定文件的路径，文件的拉取间隔和监控类型，监控类型包含3中类型：

* FileMonitoringFunction.WatchType.ONLY\_NEW\_FILES表示系统处理新的文件。
* FileMonitoringFunction.WatchType.PROCESS\_ONLY\_APPENDED表示系统处理文件追加了的内容。
* FileMonitoringFunction.WatchType.PEPROCESS\_WITH\_APPENDED表示系统不仅处理文件追加的内容，也要处理文件以前的内容。

如果文件不是text文件，你必须使用一下函数，它会让我们定义输入的文件格式：

ReadFile(fileINputFormat,path,watchType,interval,pathFilter,typeInfo)

内部，读取文件被分成两个子任务，一个子任务基于给定的watchType监控指定的文件路径，另外一个子任务执行实际的文件读取。第一个子任务不需要并行化，它仅仅是一给定的时间间隔不断的扫描给定的文件路径并汇报结果给处理程序，拆分文件并将拆分分配给各个下游线程：



### Transformations

数据转化操作是讲一个数据流转换成另外一个，输入的流可以是一个或者多个，输出流也可以是零个、一个或者多个。现在让我们一个一个的理解每一个transformations。

##### Map

这是最简单的transormations中的一个，他的输入是一个数据流，输出也是一个数据流。

In Java：

inputStream.map(new MapFunction<Integer, Integer>() {

@Override

public Integer map(Integer value) throws Exception {

return 5 \* value;

}

});

In Scala：

InputStream.map{x => x \* 5}

##### FlatMap

FlatMap将一条记录转换成零个、一个或者多个记录。

In Java:

inputStream.flatMap(new FlatMapFunction<String, String>() {

@Override

public void flatMap(String value, Collector<String> out)

throws Exception {

for(String word: value.split(" ")){

out.collect(word);

}

}

});

In Scala:

InputStream.flatMap{str => str.split(“ ”)}

##### Filter

Filter函数根据条件，如果满足条件，结果就为true，记录被保留。 Filter函数可以输出零条记录。

In Java：

inputStream.filter(new FilterFunction<Integer>() {

@Override

public boolean filter(Integer value) throws Exception {

return value != 1;

}

});

In Scala：

InputStream.filter( \_! = 1)

##### KeyBy

KeyBy根据key进行将数据流进行逻辑分区。内部使用哈希函数进行分区，返回KeyedDataStream。

In Java：

InputStream.keyBy(“someKey”)

In scala:

inputStream.keyBy(“someKey”)

##### Reduce

Reduce rolls out the KeyedDataStream by reducing the last reduced value with the current value.

The following code does the sum reduce of a KeyedDataStream.

In Java:

keyedInputStream. reduce(new ReduceFunction<Integer>() {

@Override

public Integer reduce(Integer value1, Integer value2)

throws Exception {

return value1 + value2;

}

});

In Scala:

keyedInputStream. reduce { \_ + \_ }

##### Fold

Fold rolls out the KeyedDataStream by combining the last folder stream with the current record.

It emits a data stream back.

In Java:

keyedInputStream keyedStream.fold("Start", new FoldFunction<Integer,

String>() {

@Override

public String fold(String current, Integer value) {

return current + "=" + value;

}

});

In Scala:

keyedInputStream.fold("Start")((str, i) => { str + "=" + i })

The preceding given function when applied on a stream of (1,2,3,4,5) would emit a stream like

this: Start=1=2=3=4=5

##### Aggregations

DataStream API指出多种聚合操作，例如min,max,sum等。这些函数作用域KeyedDataStream。

In Java：

keyedInputStream.sum(0)

keyedInputStream.sum("key")

keyedInputStream.min(0)

keyedInputStream.min("key")

keyedInputStream.max(0)

keyedInputStream.max("key")

keyedInputStream.minBy(0)

keyedInputStream.minBy("key")

keyedInputStream.maxBy(0)

keyedInputStream.maxBy("key")

In Scala:

keyedInputStream.sum(0)

keyedInputStream.sum("key")

keyedInputStream.min(0)

keyedInputStream.min("key")

keyedInputStream.max(0)

keyedInputStream.max("key")

keyedInputStream.minBy(0)

keyedInputStream.minBy("key")

keyedInputStream.maxBy(0)

keyedInputStream.maxBy("key")

max与maxBy最大的不同是max返回流中的最大值而maxBy返回流中最大值对应的key。min和minBy同理。

##### Window

窗口函数允许按照时间或者其他条件对keyedDataStream进行分组。以下展示了以10秒钟为时间窗口对记录进行分组的转换操作：

In Java:

inputStream.keyBy(0).window(TumblingEventTimeWindows.of(Time.seconds(1

0)));

In Scala:

inputStream.keyBy(0).window(TumblingEventTimeWindows.of(Time.seconds(1

0)))

Flink定义数据划片以此来处理无限的数据流。这些划片被称为窗口。这些滑块帮助转化操作作用于块的数据。为了在流上应用窗口操作，我们需要指明窗口操作作用的key以窗口操作作用于该key上的具体操作。为了划分流成窗口，我们可以使用flink内置的窗口函数，例如tumbing windows，sliding windows，global 和 session windows。Flink也支持用户自定义窗口划分，只需继承WindowAssginer类即可。让我们尝试理解一下这些不同的窗口划分具体是怎么工作的。

###### Global windows

Global window是一个无限的窗口 ，除非我们指定一个特殊的触发。在这种情况下，一般来说在global window每一个元素被分配成一个单一的key。如果你不指定任何触发操作，将不会有任何操作被触发。

###### Tumbing windows

Tumbing window是基于一个具体的时间划分窗口的。它们是固定长度的窗口并且不重叠。Tumbing window适用于当你需要在固定时间处理数据的情况。例如tumbing window以10分钟划分窗口，那么他将会将10分钟之内数据进行分组

###### Sliding windows

Sliding window类似于tumbing window但是它有重叠，它们是固定长度的窗口，由给定窗口滑动参数重叠以前的窗口。当您想要计算某些事件发生在特定时间内的事件时，这种窗口类型非常有用

###### Session windows

Session windows是非常有用的，它的窗口的大小是由输入数据决定的。Session window允许窗口的开始和大小可伸缩，我们还可以提供会话间隙配置参数，它指示在考虑关闭会话之前等待多久。

##### WindowALL

windowAll函数允许对常规数据流进行分组。通常这是一个非并行数据转换，因为它在非分区数据流上运行。

In Java：

inputStream.windowAll(TumblingEventTimeWindows.of(Time.seconds(10)));

In Scala：

inputStream.windowAll(TumblingEventTimeWindows.of(Time.seconds(10)))

类似于常规的数据流功能，我们也拥有窗口数据流功能，唯一的区别是他们在窗口数据流上工作。所以window reduce与reduce相似，window fold与fold类似，聚合函数也类似。

##### Union

Union是的不同的两个或者多个流合并在一起，他是并行化的，如果我们讲一个流自身进行合并，那么他会将一条记录输出两次。

In Java：

inputStream. union(inputStream1, inputStream2, ...);

In Scala：

inputStream. union(inputStream1, inputStream2, ...);

##### Window join

我们也可以使用window来通过某些keys来连接两个数据流，以下例子展示了在5秒的窗口中连接2个数据流，连接的条件死第一个数据流的第一个属性值等于第二个数据流的第二个属性值。

In Java：

inputStream. join(inputStream1)

.where(0).equalTo(1)

.window(TumblingEventTimeWindows.of(Time.seconds(5)))

.apply (new JoinFunction () {...});

In Scala：

inputStream. join(inputStream1)

.where(0).equalTo(1)

.window(TumblingEventTimeWindows.of(Time.seconds(5)))

.apply { ... }

##### Split

Split函数将将一个流按照标准分解成两个流，当你得到一个复杂流的时候，你可能像一个一个的处理他们的时候可以使用split函数。

In Java:

SplitStream<Integer> split = inputStream.split(new

OutputSelector<Integer>() {

@Override

public Iterable<String> select(Integer value) {

List<String> output = new ArrayList<String>();

if (value % 2 == 0) {

output.add("even");

}

else {

output.add("odd");

}

return output;

}

});

In Scala:

val split = inputStream.split(

(num: Int) =>

(num % 2) match {

case 0 => List("even")

case 1 => List("odd")

}

)

##### Select

这个函数允许你从split stream中选择一个特殊的流。

In Java:

SplitStream<Integer> split;

DataStream<Integer> even = split.select("even");

DataStream<Integer> odd = split.select("odd");

DataStream<Integer> all = split.select("even","odd");

In Scala:

val even = split select "even"

val odd = split select "odd"

val all = split.select("even","odd")

##### Project

Project函数允许你从一个流中选择一个子属性集合，只有选择的子属性才可以流到下一个处理流。

In Java:

DataStream<Tuple4<Integer, Double, String, String>> in = // [...]

DataStream<Tuple2<String, String>> out = in.project(3,2);

In Scala:

val in : DataStream[(Int,Double,String)] = // [...]

val out = in.project(3,2)

上面的函数从一个给定的流中选择第二个和第三个属性，下面是简单的输入数据和输出数据：

(1,10.0, A, B )=> (B,A)

(2,20.0, C, D )=> (D,C)

### Physical Partitioning

flink允许我们对流数据进行物理分区，你也可以自定义分区。让我们看看不同的分区类型。

#### Custon partitioning

如前面提到的，您可以提供分区程序的自定义实现。

In Java：

inputStream.partitionCustom(partitioner, "someKey");

inputStream.partitionCustom(partitioner, 0);

In Java：

inputStream.partitionCustom(partitioner, "someKey");

inputStream.partitionCustom(partitioner, 0);

在编写定制分区程序时，您需要确保实现高效的散列函数

#### Random partitioning

Random partitioning将数据流平均的随机分区。

In Java:

inputStream.shuffle();

In Scala:

inputStream.shuffle();

#### Rebalancing partitioning

这种类型的分区有助于均匀分配数据，它使用循环方法进行分配。这种类型的分区在数据偏斜时很好。

In Java：

InputStream.rebalance();

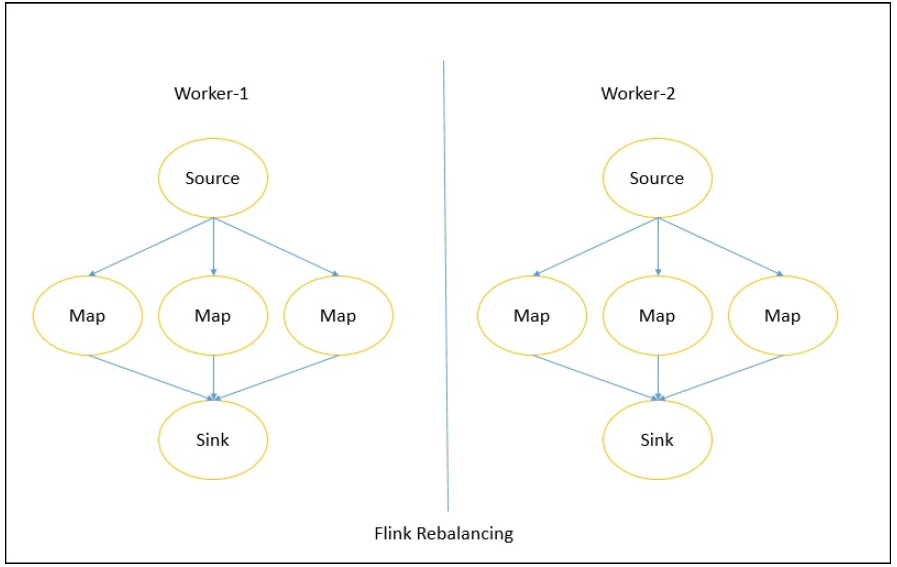
In Scala:

inputStream.rebalance()

#### Rescaling

Rescaling用于在算子之间分配数据，使得转换操作作用与数据子集上并且最后将他们聚合起来。这种情况发生在只有一个节点上，因此他不需要数据在网络上传输。

下图展示了这种分配过程：



In Java：

InputStream.rescale();

In Scala:

inputStream.rescale();

#### Broadcasting

Broadcasting 将所有记录分配给每一个分区，这将每个元素分散到所有分区。

In Java:

inputStream.broadcast();

In Scala:

inputStream.broadcast();

### Data sinks

当数据转换完毕，我们需要将结果保存在某些地方。以下是flink提供给我们保存结果的方式：

* writeAsText():一次将一行记录作为字符串写入。
* writeAsCsv():将元组以逗号分隔写入文件，行与列的分隔符你也可以自己配置。
* print()/printer():将记录写入标准输出。你也可以选择写入标准错误输出
* writeUsingOutputFormat():你可以提供一个自定义个的输出格式。你只需要继承OutputFormat类并实现其中的序列化和反序列化方法。
* writeToSocket():flink也支持将结果写入到一个特定的socket。你需要定义序列化元数据以便正确的序列化和格式化。

### Event time and watermarks

Flink Streaming API从google数据流模型中获取灵感，它支持不同的时间概念。一般来说，在流式环境中有三个地方你可以捕获时间，它们如下：

#### Event time

在生产数据的地方事件触发的时间。例如，在loT项目中，传感器捕获到数据的时间。通常这些事件时间需要在记录进入flink之前嵌入。在处理的时候，这些时间戳被提取并考虑窗口化。事件时间处理可用于无序事件。

#### Processing time

Processing time是机器在处理流数据的时候的时间。处理时间窗口只考虑处理事件的时间戳。处理时间是流处理的最简单方式，因为它不需要处理机器和生产机器之间的任何同步。在分布式异步环境中，处理时间不提供确定性，因为它取决于记录在系统中流动的速度。

#### Ingestion time

这是一个特定事件进入flink的时间。所有基于时间的操作均参考此时间戳。 摄入时间比处理更昂贵，但它会产生可预测的结果。 摄入时间程序不能处理任何乱序事件，因为它仅在事件进入Flink系统后才会分配时间戳。

以下展示了一个例子来说明event time和watermarks。在考虑摄取时间和处理时间的情况下，我们只需要时间特征和水印生成自动照顾。以下是相同的代码片段：

In Java:

final StreamExecutionEnvironment env =

StreamExecutionEnvironment.getExecutionEnvironment();

env.setStreamTimeCharacteristic(TimeCharacteristic.ProcessingTime);

//or

env.setStreamTimeCharacteristic(TimeCharacteristic.IngestionTime);

In Scala:

val env = StreamExecutionEnvironment.getExecutionEnvironment

env.setStreamTimeCharacteristic(TimeCharacteristic.ProcessingTime)

//or

env.setStreamTimeCharacteristic(TimeCharacteristic.IngestionTime)

在事件时间流程序的情况下，我们需要指定分配水印和时间戳的方式，主要有两种方式：

* 直接来自数据源属性
* 使用时间戳分配器

为了使event time工作，我们需指定时间特征：

In Java:

StreamExecutionEnvironment.getExecutionEnvironment();

env.setStreamTimeCharacteristic(TimeCharacteristic.EventTime;

In Scala:

val env = StreamExecutionEnvironment.getExecutionEnvironment

env.setStreamTimeCharacteristic(TimeCharacteristic.EventTime)

在存储记录的过程中最好将event time进行存储，flink提供了一些已经定义的时间戳提取器和水印生成器，参考：<https://ci.apache.org/projects/flink/flink-docs-release-1.2/dev/event_timestamp_extractors.html>。

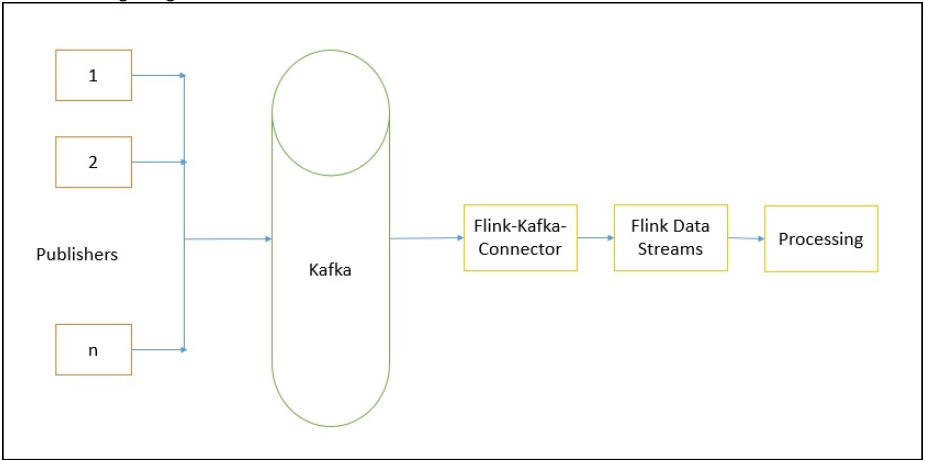
### Connectors

Flink支持多种从不同技术中读写数据的连接器。

#### Kafka connector

Kafka是一个主题订阅，分配，消息队列系统，它允许用户发布消息到一个确定的主题中，然后将该消息分发到该主题的订阅者。flink提供的选项可将Kafka消费者定义为flink流中的数据源。为了使用flink kafka连接器，我们需要使用一个特定的jar文件。

下图展示了flink的连接器怎样工作的：



我们需要使使用一下maven 依赖来使用该连接器。我已经使用了kafka0.9，所以我在pom.xml中添加如下依赖：

<dependency>

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

<artifactId>flink-connector-kafka-0.9\_2.11/artifactId>

<version>1.1.4</version>

</dependency>

现在让我们看看怎么以kafka消费者的身份使用kafka数据源：

In Java:

Properties properties = new Properties();

properties.setProperty("bootstrap.servers", "localhost:9092");

properties.setProperty("group.id", "test");

DataStream<String> input = env.addSource(new

FlinkKafkaConsumer09<String>("mytopic", new SimpleStringSchema(),

properties));

In Scala:

properties.setProperty("bootstrap.servers", "localhost:9092");

// only required for Kafka 0.8

properties.setProperty("zookeeper.connect", "localhost:2181");

properties.setProperty("group.id", "test");

stream = env.addSource(new FlinkKafkaConsumer09[String]("mytopic", new

SimpleStringSchema(), properties))

.print

在上述代码中，我们首先设置了kafka的主机名和zookeeper的主机名和端口。然后，我们制定一个主题名称，当前我们使用的是mytopic。所以，如果有任何的消息推动到mytopic主题中，我们就可以通过flink来进行处理。如果你收到不同的数据格式，你可以指定你自己的序列化元数据。默认，flink支持字符串和josn序列化。为了使容错生效，我们需要打开checkpoint。Flink不断的定期不断的将状态快照保存起来，如果有故障，flink可以从最后checkpoint的地方回复并重新启动处理流。

我们也可以定义kafka生产者定义为flink 的sink。这样我们就可以将数据写入到kafka的主题中，以下方式展示了写入数据到kafka主题的过程：

In Java:

stream.addSink(new FlinkKafkaProducer09<String>("localhost:9092",

"mytopic", new SimpleStringSchema()));

In Scala:

stream.addSink(new FlinkKafkaProducer09<String>("localhost:9092",

"mytopic", new SimpleStringSchema()));

#### Twitter connector（没学过）

#### RabbitMQ connector（没学过）

#### ElasticSearch connector（没学过）

##### Embedded node mode（没学过）

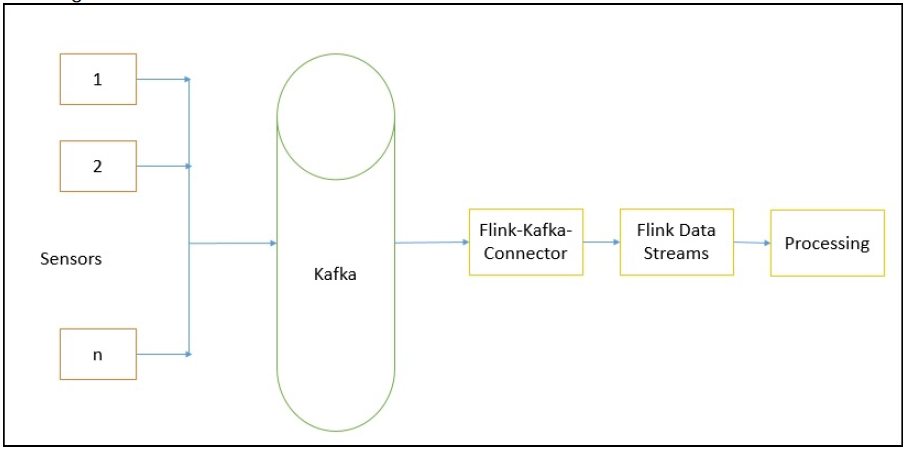
##### Transport client mode（没学过）

#### Cassandra connector（没学过）

### Use case – sensor data analytics

现在我们已经看到了多种方面的DataStream API。让我们使用这些概念来解决一个真实的用例，考虑现在有一个机器上面有一个传感器，我们需要收集这些传感器采集到的数据信息，然后统计每一个传感器每5分钟的平均温度。

以下使我们的架构：



在这种情况下。我们假象这些传感器发送给kafka主题的数据格式为：(timestamp，temperature，sensor-ID)。现在我们需要写代码来从kafka主题中读取数据并使用flink转换算子来处理它。

这里需要考虑的重要因素是我们已经有来自传感器的时间戳值，我们可以使用事件时间计算时间因素。这意味着我们即使在事件达到失序时也能够处理事件。

我们简单的流执行环境首先从kafka中读取数据开始，因为我们在事件中有时间戳，所以我们需要写一个自定义的时间戳和水印提取器以此来地区时间戳值，我们的窗口处理基于此时间戳。以下是一个代码片段：

// set up the streaming execution environment

final StreamExecutionEnvironment env =

StreamExecutionEnvironment.getExecutionEnvironment();

// env.enableCheckpointing(5000);

nv.setStreamTimeCharacteristic(TimeCharacteristic.EventTime);

Properties properties = new Properties();

properties.setProperty("bootstrap.servers", "localhost:9092");

properties.setProperty("zookeeper.connect", "localhost:2181");

properties.setProperty("group.id", "test");

FlinkKafkaConsumer09<String> myConsumer = new FlinkKafkaConsumer09<>

("temp", new SimpleStringSchema(),

properties);

myConsumer.assignTimestampsAndWatermarks(new

CustomWatermarkEmitter());

现在我们假象我们从kakfa中受到的字符串格式如下：

Timestamp,Temperature,Sensor-Id

以下代码展示了从一条记录中提取时间戳：

AssignerWithPunctuatedWatermarks<String> {

private static final long serialVersionUID = 1L;

@Override

public long extractTimestamp(String arg0, long arg1) {

if (null != arg0 && arg0.contains(",")) {

String parts[] = arg0.split(",");

return Long.parseLong(parts[0]);

}

return 0;

}

@Override

public Watermark checkAndGetNextWatermark(String arg0, long arg1)

{

if (null != arg0 && arg0.contains(",")) {

String parts[] = arg0.split(",");

return new Watermark(Long.parseLong(parts[0]));

}

return null;

}

}

现在我们简单的创建一个包含key的数据然后计算均值。代码如下：

DataStream<Tuple2<String, Double>> keyedStream =

env.addSource(myConsumer).flatMap(new Splitter()).keyBy(0)

.timeWindow(Time.seconds(300))

.apply(new WindowFunction<Tuple2<String, Double>, Tuple2<String,

Double>, Tuple, TimeWindow>() {

@Override

public void apply(Tuple key, TimeWindow window,

Iterable<Tuple2<String, Double>> input,

Collector<Tuple2<String, Double>> out) throws Exception {

double sum = 0L;

int count = 0;

for (Tuple2<String, Double> record : input) {

sum += record.f1;

count++;

}

Tuple2<String, Double> result = input.iterator().next();

result.f1 = (sum/count);

out.collect(result);

}

});

现在我们执行以上代码，如果传感器正确的将数据发送到kafka中，我们将会得到每个传感器每5分钟的平均温度。完整的代码：<https://github.com/deshpandetanmay/mastering-flink/tree/master/chapter02/flink-streaming>.

### Summary

本章，我们开始讲解了许多重要的DataStream API。我们看到了数据源，转换算子以及sink是怎么一起工作的，然后我们看到了各种连接器例如elastricsearch,Cassandra,Kafka,RabbitMQ等等最后，我们使用flink解决了一个真实事件传感器数据的分析。

下一章，我们将要学习另外一种重要flink生态系统的一个DataSet API。

## 3. Data Processing Using the Batch Processing API

### DataSources

### Transformations

### Broadcast variables

### Data sinks

### Connectors

### Iterations

### Use case – Athletes data insights using Flink batch API

### summary

## 4. Data Processing Using the Table API

## 5. Complex Event Processing

## 6. Maching Learning Using FlinkML

## 7. Flink Graph API – Gelly

## 8. Distributed Data Processing with Flink and Hadoop

## 9. Deploying Flink on Cloud

## 10. Best Practices