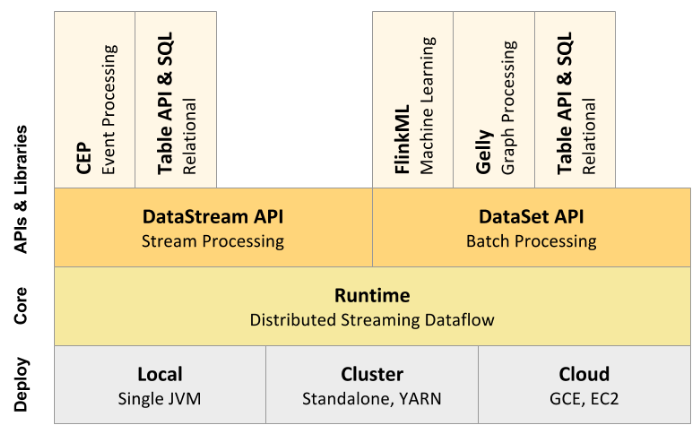
# 1. Introduction

Flink是开源的流处理和批处理的平台。核心是创建针对流数据的分布式数据处理和失效恢复的流数据引擎；在流引擎的基础上，增加了批处理的引擎，同时也支持内存管理、程序优化等。（Streaming is the core.）

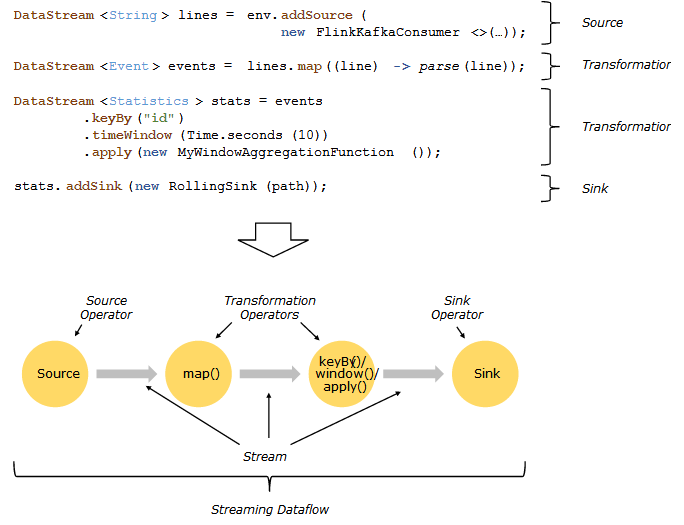
Flink的技能栈：



# 2. Concepts

## 2.1 Programs and Dataflows

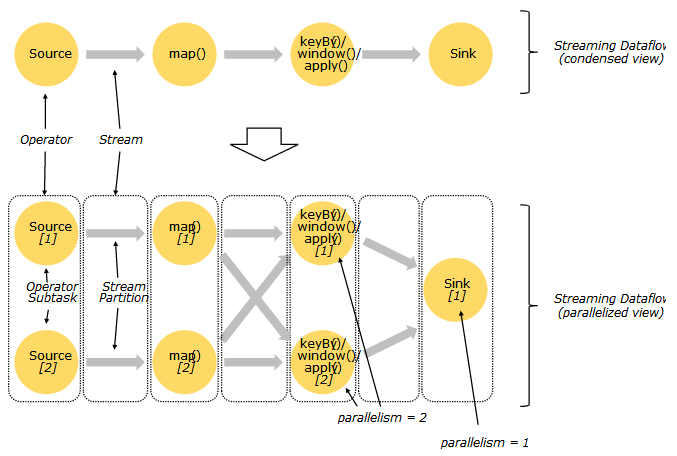
Flink的程序是由**Streams**和**transformations**组成的（注意到DataSet内在的也是一个流而已）。Stream是中间结果，transformation是一种操作，这个操作以一个或多个stream作为输入，计算得到一个或多个stream作为输出。Flink的程序被映射成流动的数据流。该数据流由streams和transformation operators组成。每个数据流开始于一个或多个数据源，结束于一个或多个**sink**操作。这有点像DAG图，但Flink也支持回路操作。一个DataFlow的流动过程如下图所示：



### 2.1.1 并行的流（Parallel Dataflows）

Flink的程序是天然并行和分布式的。streams被分割成**stream partitions**，operators被分割成**operator subtasks**.操作子集（opeartor subtasks）的执行彼此之间都是独立的，在不同的机器或容器上的不同的线程中。

操作子集的数量称之为这个操作的**parallelism**。一个流的**parallelism**通常是他的生产操作所决定的。一个程序中的不同操作可能有不同的并行度（parallelism）。如下图所示：



partition 数目是如何确定的。举例：Hadoop根据分片的数量。

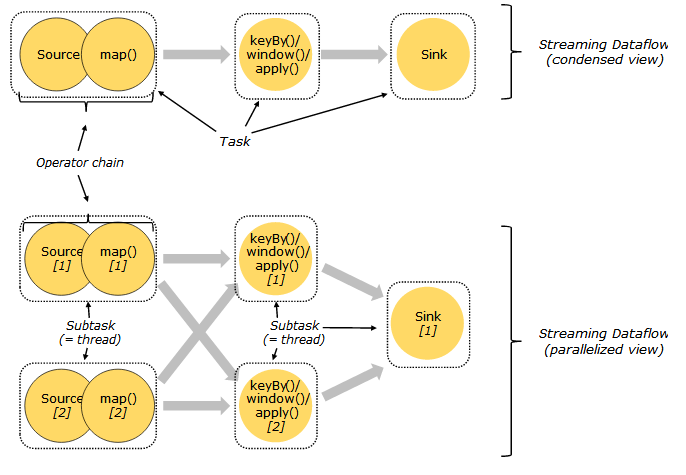
由上图可以看出节点时操作，连接是数据。stream连接不同的operator。如上所示，流可以和operator是一一对应的关系，也可以是从新分配的关系。

**One-to-one**:例如图中的source的subtask[1]所产生的分片中的数据顺序和map的subtask[1]所接收的分片中的数据顺序是完全一致的，他们两个通过一个stream partition连接起来，且是一一对应的关系；

**Redistributing**:例如图中的map和keyBy/Window以及keyBy/Window和sink之间的映射不是一一对应的，前一个subtask可能将数据发送到不同的target subtask,这依据选择转换规则（Selected Transformation）,通常这些操作有keyBy(通过hashcode重新分片)，broadcast(),rebalance(随机的重新分配)，在这种模式下，一对subtask之间是可以保证数据有序的，例如map的subtask[1]和keyBy的subtask[1]之间的数据流是有序的。

### 2.1.2 任务和操作链（Tasks & Operator Chains）

对于分布式执行的任务，Flink Chains能够将多个subtasks集成一个tasks.每一个task被一个线程执行。操作链放在一个task中是一种不错的优化方式：他可以减少线程的数量，也可以降低输出的流的数量，chain方式可以通过API来配置。如下的流可以通过chain方式合并成5个task.



## 2.2 Distributed Execution

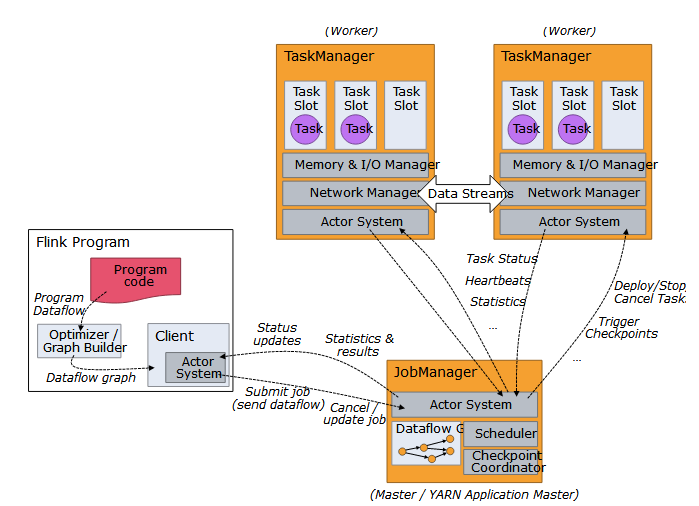
### 2.2.1 Master, Worker, Client

Flink的执行逻辑包括两种：

**主节点**（**master/JobManager**）:协调分布式执行计划，他们安排任务，设置检查点以及进行失效恢复等；通常情况下至少有一个主节点，也可以通过建立多个主节点来提高可用性，当有多个主节点时，只有一个是**leader**,其他的是**standby**.

**工作节点**（**worker/TaskManager**）:真正执行数据流中任务（具体来说，应该是subtask）的节点，至少有一个工作节点。工作节点连接到主节点，报备可用性，获取任务的分配。

**客户端**（**client**）：客户端并不是程序执行中的一部分。但它可以向主节点发送数据流。发送完成后客户端可以断开连接，或者保持连接来接收运行结果。客户端可以以Java/Scala的语言形式来触发执行器，也可以通过命令行（例如./bin/flink run）的形式来触发。



### 2.2.2 Workers, Slots, Resources

一个工作节点（**Worker**）就是一个单独的**JVM**进程，可能以多线程的方式同时执行多个子任务（**subtasks**），为了控制一个工作节点所能接收的任务的数量，这里引进了**task slots**概念。

每一个task slot代表一个TaskManager的固定资源的集合。例如一个TaskManager有3个task slots,那么每个slot可以分配该TaskManager拥有的1/3的内存。划分资源的目的是为了防止不同的子任务之间对内存进行竞争。注意这里并没有对CPU资源进行隔离。因此slots只是对内存资源进行了隔离。

调整task slots的数量以允许用户能够定义多少个子任务（subtask）被隔离。如果一个TaskManager只有一个slot,这意味着每个任务组都在一个单独的JVM里运行。如果一个TaskManager有多个slot,这意味着更多的子任务（subtasks）可以分享同一个JVM.同一个JVM里的任务分享TCP连接和心跳包，他们也会分享数据结构和数据集，因此减少了每个任务的开销。

默认情况下，Flink允许相同的Job中不同的subtask共享同一个slot.这有可能使得一个slot管理了整个job的流程。允许共享slot有如下两点好处：

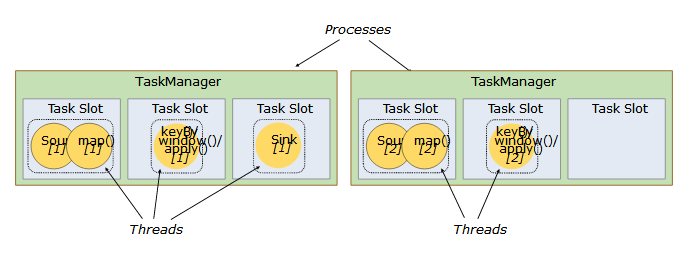
一个Flink集群可能需要非常多的tasks slots高度并行执行。并不需要计算出一个程序总共需要多少个tasks。

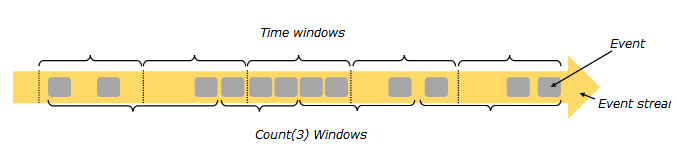
这能够更好的利用资源。如果slot不能被共享，如source/map()这样非密集的subtasks锁住资源数目和那些诸如window这样密集的subtasks一样多。通过共享slot,将并行度从2提高到了6，从而充分利用了切割的资源，然而还能够确保每个TaskManager针对繁重的子任务还能够公平共享。

共享slot的操作可以通过API来设置，一般说来，当task slots的数量和CPU核数相同时最佳。

## 2.3 Time and Windows

聚集函数(**Aggregating events**)（例如counts,sums）在流数据和批数据上的含义有所不同。准确的说，你不能统计一个流数据中的数据量，因为流数据是无限的。然而如果在流数据中引进窗口（**Window**）概念，统计某个窗口中的数据量，这样的操作是有意义的。例如统计“最近5分钟的数据量”，“求最近100个元素的总和”等。

窗口可以是基于时间（**time driven**例如每30s一个窗口），也可以事基于数据的（**data driven**例如每100个元素一个窗口）;还可以按照是否重叠来对窗口进行划分：**tumbing windows**(窗口之间没有交集)，**sliding windows**(窗口之间可以有交集)，**session windows**(基于每次活动的间隔)如下图所示：



更多关于**windows**的介绍：<https://flink.apache.org/news/2015/12/04/Introducing-windows.html>

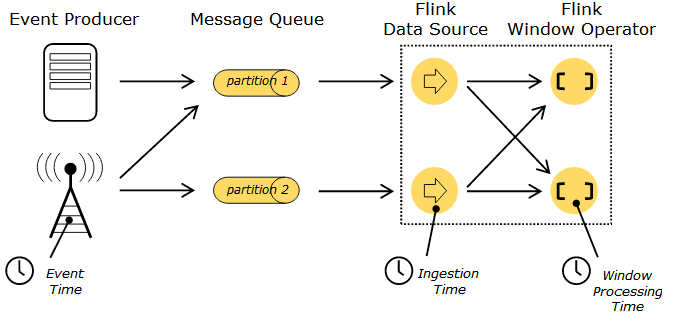
### 2.3.1 Time

流处理程序中的时间概念有很多：

**Event Time**: 事件被创建的时间。它通常是一个事件的时间戳属性，例如是由生产者赋予的成产时间，Flink能够获取事件的时间戳通过timestamp assigners.

**Ingestion Time**: 是事件作为数据源操作进入Flink的时间。

**Processing Time**: 是每个操作的执行时刻的时间。



更多关于**time**的处理细节见：<https://ci.apache.org/projects/flink/flink-docs-release-1.2/dev/event_time.html>

## 2.4 State and Fault Tolerance状态保持和错误容忍

在一个dataflow中，多数的操作似乎一次只处理一个事件（例如事件分析器），有些操作需要记录多个事件的信息（例如窗口操作）。这些操作，被称为有状态的（**stateful**）。

有状态的操作的状态是被一个称之为嵌入式的key/value store维持的。这些状态被分散在流中，并且被有状态的操作读取。

# 3. Quick Start

## 3.1 Setup

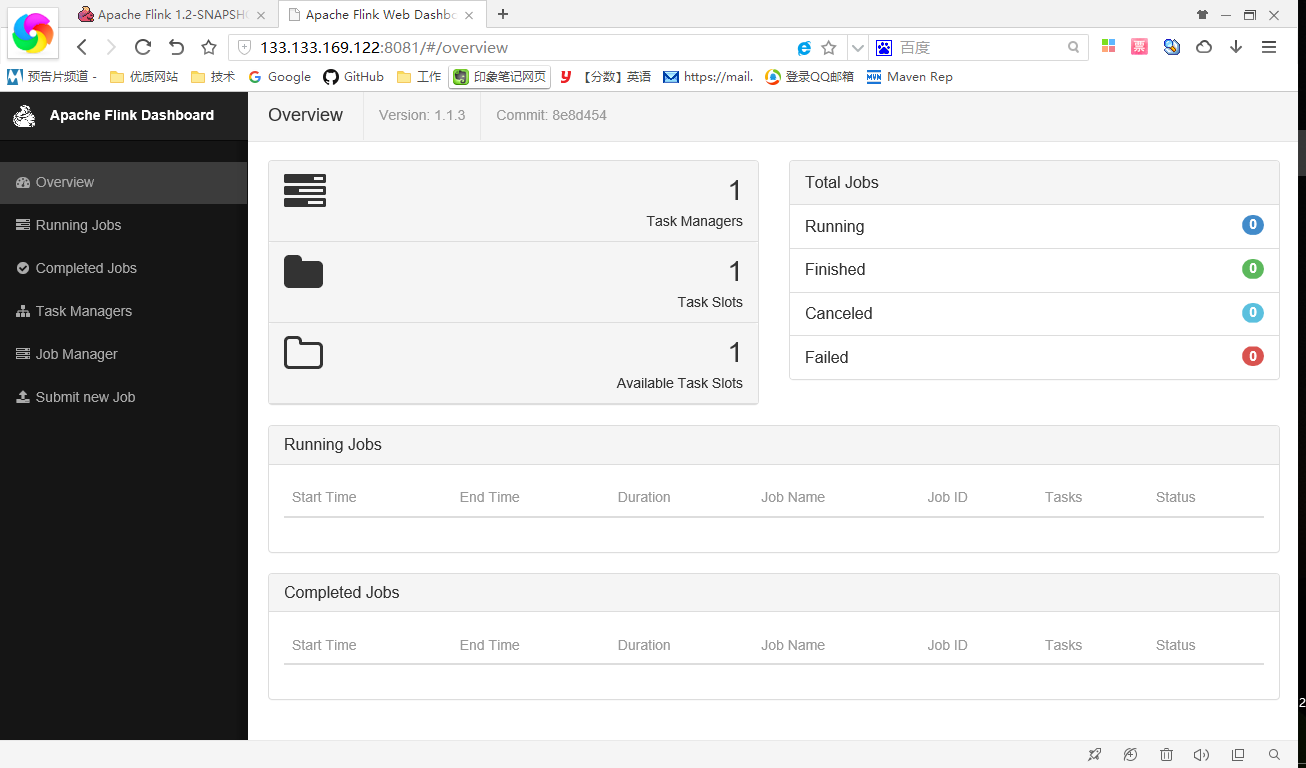
### 3.1.1 Basic Setup

(1) 下载二进制包：<http://flink.apache.org/downloads.html>

(2) 解压并进入包中，运行：bin/start-local.sh

(3) 输入网址：<http://localhost:8081>

得到如下图所示：



### 3.1.2 Run Example

以运行SocketWindowWordCount 例子为例：

（1）$ nc -l 9000 #启动监听服务

（2）$ bin/start-local.sh #开启一个Flink Server

（3）$ bin/flink run examples/streaming/SocketWindowWordCount.jar --port 9000 #运行实例

（4）$ nc -l 9000 #在这个命令下输入单词

lorem ipsum

ipsum ipsum ipsum

bye

（5）$ tail -f log/flink-\*-jobmanager-\*.out #得到统计结果

**(**lorem,1**)**

**(**ipsum,1**)**

**(**ipsum,2**)**

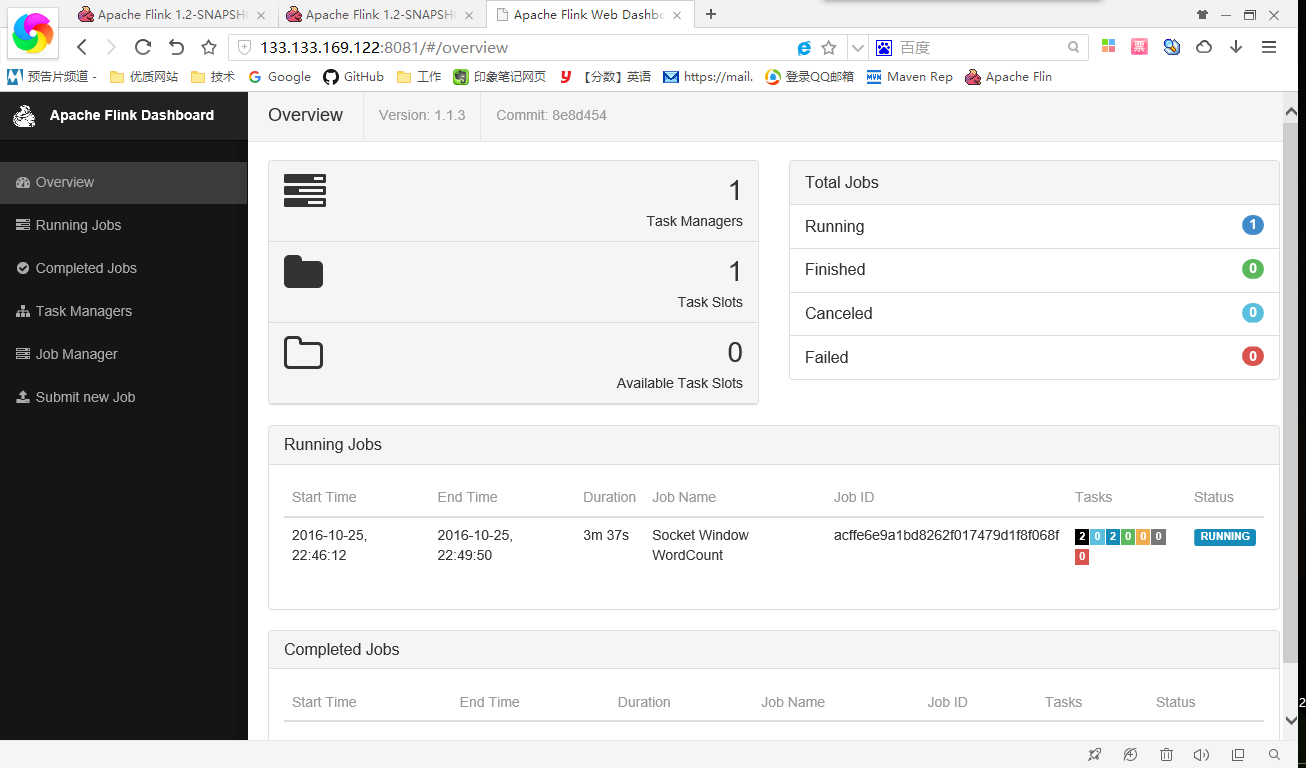
**(**ipsum,3**)**

**(**ipsum,4**)**

**(**bye,1**)**

$ bin/stop-local.sh #停止服务

运行结果如下：



### 3.1.3 Cluster Setup

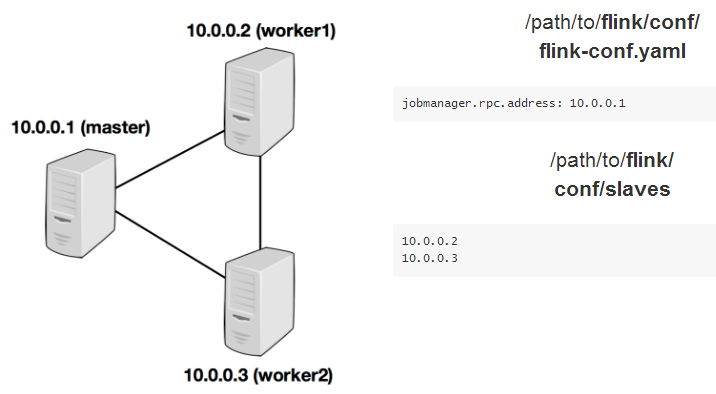
1. 配置主从节点

（1）在conf/flink-conf.yaml文件中jobmanager.rpc.address设置主节点（master）的IP地址；

（2）在conf/slaves文件中设置从节点(worker)的IP地址；

2. 将整个文件夹拷贝到多台主机上；

3. master节点中启动bin/start-cluster.sh



## 3.2 Execution

### 3.2.1 Local Execution

Flink可以在一个单独的JVM进程中执行，这样方便用户在本地进行测试和调试。Flink支持两种不同的本地执行方式：**LocalExecutionEnvironment**和**CollectionEnvironment**。

**LocalExecutionEnvironment**：启动了全部的Flink Runtime,包括了一个JobManager和一个TaskManager。他们像集群模式一样拥有内存管理和所有内置的算法。

**CollectionEnvironment**：在Java集合上执行Flink的程序。这个模式不会全部启动Flink Runtime,因此他们的执行是低开销的，轻量的。例如DataSet.map()中的map()操作可以在Java list集合上执行。

**（1）LocalExecutionEnvironment**

有两种方式可以启动LocalExecutionEnvironment：

a. **ExecutionEnvironment.createLocalEnvironment()**：默认情况下，它会使用和CPU核数一样多的线程来执行任务，当然也可以自己指定并行度。local environment可以通过**enableLogging()** / **disableLogging()**来配置输出。

具体设置的方法为：

ExecutionEnvironment env = ExecutionEnvironment.*createLocalEnvironment*();  
env.getConfig().disableSysoutLogging();

b. **ExecutionEnvironment.getExecutionEnvironment()：**多数情况下，这是一个更好的启动方式。它将会返回一个LocalEnvironment。当以命令行的形式，他将会返回一个配置好的集群执行环境。

集群启动时，也可以配置参数：

Configuration conf **=** **new** **Configuration();**

conf**.**setFloat**(**ConfigConstants**.**TASK\_MANAGER\_MEMORY\_FRACTION\_KEY**,** 0.5f**);**

**final** ExecutionEnvironment env **=** ExecutionEnvironment**.**createLocalEnvironment**(**conf**);**

|  |
| --- |
| Local Environment 示例： |
| public static void main(String[] args) throws Exception {  ExecutionEnvironment env = ExecutionEnvironment.createLocalEnvironment();  DataSet<String> data = env.readTextFile("file:///path/to/file");  data  .filter(new FilterFunction<String>() {  public boolean filter(String value) {  return value.startsWith("http://");  }  })  .writeAsText("file:///path/to/result");  JobExecutionResult res = env.execute();  } |

**（2）Collection Environment**

在Java集合上使用**CollectionEnvironment**是一个低开销的轻量级的执行Flink程序的方式。通常是在自动化测试，调试，代码重用的情景下使用。请注意这种情景只适合在小数量级上使用，而且不能超过JVM堆的大小，这种在Collections上执行方式不是多线程的，只有一个线程被使用。

|  |
| --- |
| Collection Environment示例： |
| public static void main(String[] args) throws Exception {  *// initialize a new Collection-based execution environment*  final ExecutionEnvironment env = new CollectionEnvironment();  DataSet<User> users = env.fromCollection( */\* get elements from a Java Collection \*/*);  */\* Data Set transformations ... \*/*  *// retrieve the resulting Tuple2 elements into a ArrayList.*  Collection<...> result = new ArrayList<...>();  resultDataSet.output(new LocalCollectionOutputFormat<...>(result));  *// kick off execution.*  env.execute();  *// Do some work with the resulting ArrayList (=Collection).*  for(... t : result) {  System.err.println("Result = "+t);  }  } |

### 3.2.2 Cluster Execution

有两种方式可以将我们编写的Flink程序送到Flink集群中执行。当然不管是单机执行还是集群执行，我们都必须在程序中包含flink的客户端，MAVEN依赖如下：

<dependency>

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

<artifactId>flink-clients\_2.10</artifactId>

<version>1.2-SNAPSHOT</version>

</dependency>

**（1）Command Line Interface**

我们可以将打包好的（JARS）程序提交到一个集群或单机中执行。我们可以在flink的binary解压包的**/bin/flink** 来执行命令。

例如：./bin/flink run ./examples/batch/WordCount.jar

也可以用来触发SavePoints

例如：./bin/flink savepoint <jobID> [savepointDirectory]

更具体的shell见：<https://ci.apache.org/projects/flink/flink-docs-release-1.2/setup/cli.html>

**（2）Remote Environment**

我们也可以通过API来获取一个远程执行环境（Remote Environment）。如下所示：

|  |
| --- |
| 获取远程执行环境示例： |
| public static void main(String[] args) throws Exception {  ExecutionEnvironment env = ExecutionEnvironment  .createRemoteEnvironment("flink-master", 6123, "/home/user/udfs.jar");  DataSet<String> data = env.readTextFile("hdfs://path/to/file");  data  .filter(new FilterFunction<String>() {  public boolean filter(String value) {  return value.startsWith("http://");  }  })  .writeAsText("hdfs://path/to/result");  env.execute();  } |

注意到在上面的代码示例中，客户端使用到了自己私有的JAR文件，因此需要在创建远程环境时，指定JAR包的路径。

**（3）Linking with modules not contained in the binary distribution**

一般情况下，二进制包的lib文件夹下包含了分布式运行环境所有的所有jar包，并且这些jar包会自动加载到分布式程序的类路径中。但总有一些jar包没有被包含进去。例如streaming connectors和Flink新引进的模块都没有被包含进去，为了使得一些依赖于这些模块的包的程序能够被运行，你需要让这些模块能够在运行时刻被分布式的运行环境所感知和捕获。对此我们有两种方式：

a. 拷贝这些模块的JAR包到TaskManager的lib文件夹下，并且重启TaskManager.

b. 把应用程序所需要的这些模块的jar包打包到应用程序中。

如果提交给集群中的任务，包含一些原生集群中不存在的类（例如这些类是用户自定义的）则可能导致任务无法执行，并报错：

java.lang.RuntimeException: The initialization of the DataSource's outputs caused an error: Could not read the user code wrapper: com.duansky.learn.flink.\_DataSet$LineSplitter

# 4. APIs

在这一章节主要介绍了Flink API的使用规则，偏向于函数式编程，所以需要预先补习**函数式编程**思想。再看这一章节时，就比较轻松。

## 4.1 Basic API Concepts

### 4.1.1 Anatomy of a Flink Program

一个标准的Flink程序，有如下五个步骤：

1. Obtain an **execution environment**,
2. Load/create the **initial data**,
3. Specify **transformations** on this data,
4. Specify where to put the **results** of your computations,
5. **Trigger** the program execution

## 4.2 Streaming(DataStream API)

#### 4.1.1.4 Filter

|  |
| --- |
| Filter  针对某个数据集，只返回数据集中元素带入函数中返回结果为true的元素。即过滤掉数据集中不符合条件的元素。注意，该函数只验证真假，不能修改元素的值。 |
| public interface FilterFunction<T> extends Function, Serializable {    */\*\*  \* The filter function that evaluates the predicate.  \* <p>  \* <strong>IMPORTANT:</strong> The system assumes that the function does not  \* modify the elements on which the predicate is applied. Violating this assumption  \* can lead to incorrect results.  \*   \* @param value The value to be filtered.  \* @return True for values that should be retained, false for values to be filtered out.  \*   \* @throws Exception This method may throw exceptions. Throwing an exception will cause the operation  \* to fail and may trigger recovery.  \*/* boolean filter(T value) throws Exception; } |

示例：过滤掉集合中小于0的元素。

public static class NaturalNumberFilter implements FilterFunction<Integer>{  
 public boolean filter(Integer value) throws Exception {  
 return value >= 0;  
 }  
}

## 4.3 Batch(DataSet API)

原文:

<https://ci.apache.org/projects/flink/flink-docs-release-1.2/dev/batch/dataset_transformations.html>

Flink程序中的DataSet是常规的数据集，它包括实现在这些数据集上的转换（transformation），例如 filtering, mapping, joining ,grouping等。数据集是由特定的数据源（data source）初始化，这些数据源可能是读取某个文件或者某个集合；经过若干次的转换（transformation），最后通过sink来返回结果。sink包括将数据写到特定的文件，或者输出到屏幕等。

**流程**：data source -> transformations -> data sink

### 4.3.1 DataSet Transformations

最好的方式就是直接看DataSet API.

原文：

<https://ci.apache.org/projects/flink/flink-docs-release-1.2/dev/batch/dataset_transformations.html#reduce-on-grouped-dataset>

#### 4.3.1.1 Map

|  |
| --- |
| MapFunction  实现元素中一对一的映射。将原来集合中的每一个元素，映射成另外一个元素。 |
| public interface MapFunction<T, O> extends Function, Serializable {   */\*\*  \* The mapping method. Takes an element from the input data set and transforms  \* it into exactly one element.  \*  \* @param value The input value.  \* @return The transformed value  \*  \* @throws Exception This method may throw exceptions. Throwing an exception will cause the operation  \* to fail and may trigger recovery.  \*/* O map(T value) throws Exception; } |

示例：如下代码实现了将一个Tuple2<Integer,Integer>元素映射成一个Integer。

public static class MyMapFunction implements MapFunction<Tuple2<Integer,Integer>,Integer>{  
 public Integer map(Tuple2<Integer, Integer> value) throws Exception {  
 return value.f0 + value.f1;  
 }  
}

#### 4.3.1.2 FlatMap

|  |
| --- |
| FlatMap  实现元素中的一对多映射。能够将集合中的一个元素映射成任意数量（0个，1个或多个）的元素。 |
| public interface FlatMapFunction<T, O> extends Function, Serializable {   */\*\*  \* The core method of the FlatMapFunction. Takes an element from the input data set and transforms  \* it into zero, one, or more elements.  \*  \* @param value The input value.  \* @param out The collector for returning result values.  \*  \* @throws Exception This method may throw exceptions. Throwing an exception will cause the operation  \* to fail and may trigger recovery.  \*/* void flatMap(T value, Collector<O> out) throws Exception; } |

示例：将一个字符串映射成0个、1个或者多个字符串。

public void flatMap(String value, Collector<String> out) throws Exception {  
 for(String single : value.split("\\W")) {  
 out.collect(single);  
 }  
}

#### 4.1.1.3 MapPartition

|  |
| --- |
| MapPartition  每次传入一个partition中的所有数据，然后提供针对该partition数据的处理逻辑，并返回0个，1个或多个结果。如下所示中values迭代器是运行在该Partition中所有数据上的迭代器，可以拿到该Partition中的所有数据。Parition的数目跟算子的并行度有关。 |
| public interface MapPartitionFunction<T, O> extends Function, Serializable {    */\*\*  \* A user-implemented function that modifies or transforms an incoming object.  \*  \* @param values All records for the mapper  \* @param out The collector to hand results to.  \* @throws Exception  \*/* void mapPartition(Iterable<T> values, Collector<O> out) throws Exception; } |

示例：如下类用来统计每个Partition中的数据量。

public static class PartitionCounter implements MapPartitionFunction<String,Long>{  
  
 public void mapPartition(Iterable<String> values, Collector<Long> out) throws Exception {  
 long res = 0;  
 for(String value : values){  
 System.*out*.println(value);  
 res++;  
 }  
 out.collect(res);  
 }  
}

#### 4.1.1.5 Projection

|  |
| --- |
| Projection  在Tuple 类型的数据集上的投影操作。该操作不需要传入用户自定义的类。实现的功能是在原来的Tuple上选择某几维作为新的数据集。 |
| */\*\*  \* Applies a Project transformation on a {@link Tuple} {@link DataSet}.<br>  \* <b>Note: Only Tuple DataSets can be projected using field indexes.</b><br>  \* The transformation projects each Tuple of the DataSet onto a (sub)set of fields.<br>  \* Additional fields can be added to the projection by calling {@link ProjectOperator#project(int[])}.  \*  \* <b>Note: With the current implementation, the Project transformation looses type information.</b>  \*  \* @param fieldIndexes The field indexes of the input tuple that are retained.  \* The order of fields in the output tuple corresponds to the order of field indexes.  \* @return A ProjectOperator that represents the projected DataSet.  \*  \* @see Tuple  \* @see DataSet  \* @see ProjectOperator  \*/* public <OUT extends Tuple> ProjectOperator<?, OUT> project(int... fieldIndexes) {  return new Projection<>(this, fieldIndexes).projectTupleX(); } |

示例：将选取原来三维数据中的第2维和第0维 形成新的数据集。

DataSet**<**Tuple3**<**Integer**,** Double**,** String**>>** in **=** *// [...]*

*// converts Tuple3<Integer, Double, String> into Tuple2<String, Integer>*

DataSet**<**Tuple2**<**String**,** Integer**>>** out **=** in**.**project**(**2**,**0**);**

#### 4.1.1.6 GroupBy

能够在分组数据集上进行聚合操作。定义数据的分组方式有如下几种：

* key expressions（key表达式）
* a key-selector function（key选择器）
* one or more field position keys (Tuple DataSet only)
* Case Class fields (Case Classes only)

#### 4.1.1.7 Reduce [Grouped/Full DataSet]

通过引入用户自定义的聚合函数（reduce function），能够将分组数据集中的每组数据聚合成一个单一元素，也可以将整个数据集聚合成一个单一元素。对于每组数据，聚合函数能够将两个元素聚合成一个元素，重复这样的操作过程直到每组只剩下一个元素为止；对于整个数据集，聚合函数则重复两两聚合操作，直至整个数据集只剩下一个元素。

**注意**

该方法既可以运用在分组数据集上=>每组数据集产生一个结果；

也可以运用在整个数据集上=>整个数据集产生一个结果。

**（1）Reduce on DataSet Grouped by Key Expression**

通过挑选数据集的元素中的一个或多个属性作为分组标准key来分组。每一个key expression 都是一个公有的属性或者一个getter方法。也可以使用“.”来继续引用对象中的对象（例如user.address）。而“\*”表示选择所有的属性作为key选择器。示例如下：

public class \_Reduce {  
  
 */\*\*  
 \* word and count POJO  
 \*/* public static class WC{  
 public String word;  
 public int count;  
 public WC(String word,int count){  
 this.word = word;  
 this.count = count;  
 }  
 }  
  
 */\*\*  
 \* ReduceFunction that sums Integer attributes of a POJO  
 \*/* public static class WordCounter implements ReduceFunction<WC>{  
 public WC reduce(WC value1, WC value2) throws Exception {  
 return new WC(value1.word, value1.count + value2.count);  
 }  
 }  
  
 public static void main(String args[]) throws Exception {  
 ExecutionEnvironment env = ExecutionEnvironment.*getExecutionEnvironment*();  
 List<WC> list = new ArrayList<WC>(10);  
 list.add(new WC("a",1)); list.add(new WC("b",1));  
 list.add(new WC("b",1)); list.add(new WC("b",1));  
 DataSet<WC> words = env.fromCollection(list);  
  
 words  
 .groupBy("word")// DataSet grouping on field "word"  
 .reduce(new WordCounter())// apply ReduceFunction on grouped DataSet  
 .print()  
 ;  
 }  
}

结果抛出异常：

Exception in thread "main" org.apache.flink.api.common.InvalidProgramException: This type (GenericType<com.duansky.benchmark.flink.test.dataset.transformations.\_Reduce.WC>) cannot be used as key.

根据**groupBy(String... fields)**函数，能够被挑选成key的，有三种情况：

a. if (type instanceof CompositeType)

b. if (!type.isKeyType()) TypeInformation类自动完成判断，一般用户自定义的类是GenericTypeInfo.而它的实现方式是判断该类是否实现了Comparable。如果没有实现直接报错。如果实现了转入c.

c. if (!(*SELECT\_ALL\_CHAR*.equals(keyExpr) || *SELECT\_ALL\_CHAR\_SCALA*.equals(keyExpr)))

判断该字符是否是\*或者\_，如果不是则报错。

也就是说如果希望在自定义的类的DataSet上实现groupBy功能，则该类必须是CompositeType类型，即简单的POJO对象。Flink中对POJO对象的要求如下：

*/\*\*  
 \* TypeInformation for "Java Beans"-style types. Flink refers to them as POJOs,  
 \* since the conditions are slightly different from Java Beans.  
 \* A type is considered a FLink POJO type, if it fulfills the conditions below.  
 \* <ul>  
 \* <li>It is a public class, and standalone (not a non-static inner class)</li>  
 \* <li>It has a public no-argument constructor.</li>  
 \* <li>All fields are either public, or have public getters and setters.</li>  
 \* </ul>  
 \*   
 \** ***@param*** <*T*> *The type represented by this type information.*

因此上面的WC类，不能是static，也必须要有无参构造函数。

如下：

public class WC {  
  
 private String word;  
 private int count;  
  
 public WC(){}  
  
 public WC(String word,int count){  
 this.word = word;  
 this.count = count;  
 }  
  
 public int getCount() {  
 return count;  
 }  
  
 public void setCount(int count) {  
 this.count = count;  
 }  
  
 public String getWord() {  
 return word;  
 }  
  
 public void setWord(String word) {  
 this.word = word;  
 }  
  
 public String toString(){  
 return word + ":" + count;  
 }  
}

此时输出结果为：

a:1

b:3

**（2）Reduce on DataSet Grouped by KeySelector Function**

|  |
| --- |
| KeySelector  Key选择器可以从一个数据集中的每个元素中提取一个key值，这个key值作为分组的标准。 |
| public interface KeySelector<IN, KEY> extends Function, Serializable {    */\*\*  \* User-defined function that extracts the key from an arbitrary object.  \*   \* For example for a class:  \* <pre>  \* public class Word {  \* String word;  \* int count;  \* }  \* </pre>  \* The key extractor could return the word as  \* a key to group all Word objects by the String they contain.  \*   \* The code would look like this  \* <pre>  \* public String getKey(Word w) {  \* return w.word;  \* }  \* </pre>  \*   \* @param value The object to get the key from.  \* @return The extracted key.  \*   \* @throws Exception Throwing an exception will cause the execution of the respective task to fail,  \* and trigger recovery or cancellation of the program.   \*/* KEY getKey(IN value) throws Exception; } |

示例：如下所示，针对每个输入的WC对象，选择该对象的word属性作为key来对DataSet数据集进行分组。

public static class SelectWord implements KeySelector<WC,String>{  
 @Override  
 public String getKey(WC value) throws Exception {  
 return value.getWord();  
 }  
}

**（3）Reduce on DataSet Grouped by Field Position Keys (Tuple DataSets only)**

Field Position keys 选择Tuple类型数据集的一个或多个属性作为分组的key。示例如下所示：

DataSet**<**Tuple3**<**String**,** Integer**,** Double**>>** tuples **=** *// [...]*

DataSet**<**Tuple3**<**String**,** Integer**,** Double**>>** reducedTuples **=**

tuples

*// group DataSet on first and second field of Tuple*

**.**groupBy**(**0**,** 1**)**

*// apply ReduceFunction on grouped DataSet*

**.**reduce**(new** **MyTupleReducer());**

**（4）Reduce on DataSet grouped by Case Class Fields**

当使用Case Classes时，你也可以通过指定属性的名字来作为分组的key. Scala代码如下：

**case** **class** **MyClass(val** a**:** **String,** b**:** **Int,** c**:** **Double)**

**val** tuples **=** **DataSet[MyClass]** **=** *// [...]*

*// group on the first and second field*

**val** reducedTuples **=** tuples**.**groupBy**(**"a"**,** "b"**).**reduce **{** **...** **}**

#### 4.1.1.8 GroupReduce [Grouped/Full DataSet]

GroupReduce和Reduce很像，区别在于GroupReduce一次接收和处理一个分组数据，并且产生0个、1个或多个结果。而Reduce是每次接收分组中的两个数据，并且将这两个数据合并成一个数据，然后重复这样的操作直至分组只剩下一个数据。同样，GroupReduce既可以运用在分组数据集上，也可以运用在整个数据集上；运用在整个数据集上可以看做整个数据集只有一个分组。

|  |
| --- |
| GroupReduce  类似于MapPartition，GroupReduce一次接收一个分组的数据，并且能够产生0个，1个或多个结果。 |
| public interface GroupReduceFunction<T, O> extends Function, Serializable {    */\*\*  \* The reduce method. The function receives one call per group of elements.  \*   \* @param values All records that belong to the given input key.  \* @param out The collector to hand results to.  \*  \* @throws Exception This method may throw exceptions. Throwing an exception will cause the operation  \* to fail and may trigger recovery.  \*/* void reduce(Iterable<T> values, Collector<O> out) throws Exception; } |

示例如下：接收一个Tuple2<Integer,String>元素，去除同一个组中重复的元素。

public static class DistinctReduce implements GroupReduceFunction<Tuple2<Integer,String>,Tuple2<Integer,String>>{  
 @Override  
 public void reduce(Iterable<Tuple2<Integer, String>> values, Collector<Tuple2<Integer, String>> out) throws Exception {  
 Integer key = null;  
 Set<String> set = new HashSet<>();  
 for (Tuple2<Integer, String> value : values) {  
 key = value.f0;  
 set.add(value.f1);  
 }  
 for (String s : set) {  
 out.collect(new Tuple2<>(key,s));  
 }  
 }  
}

**sort group**

|  |
| --- |
| sortGroup(int field, Order order)  可以使用该函数来对一个组中的所有元素进行排序。 |
| */\*\*  \* Sorts {@link org.apache.flink.api.java.tuple.Tuple} elements within a group on the specified field in the specified {@link Order}.<br>  \* <b>Note: Only groups of Tuple elements and Pojos can be sorted.</b><br>  \* Groups can be sorted by multiple fields by chaining {@link #sortGroup(int, Order)} calls.  \*   \* @param field The Tuple field on which the group is sorted.  \* @param order The Order in which the specified Tuple field is sorted.  \* @return A SortedGrouping with specified order of group element.  \*   \* @see org.apache.flink.api.java.tuple.Tuple  \* @see Order  \*/* public SortedGrouping<T> sortGroup(int field, Order order) {  if (this.getKeys() instanceof Keys.SelectorFunctionKeys) {  throw new InvalidProgramException("KeySelector grouping keys and field index group-sorting keys cannot be used together.");  }   SortedGrouping<T> sg = new SortedGrouping<T>(this.inputDataSet, this.keys, field, order);  sg.customPartitioner = getCustomPartitioner();  return sg; } |

**补：Combinable GroupReduceFunctions**

#### 4.1.1.9 GroupCombine [Grouped/Full DataSet]

|  |
| --- |
| CombineGroup  该函数能够局部的合并分组数据集中的元素，并且针对每个组输出0个，1个或多个元素。该函数的功能和GroupReduce的功能很像，都是用来合并分组数据集中的元素。区别在于CombineGroup操作的代价很小，而且不会在整个数据集上进行合并操作，是一个内存方案的，在独立的分片上进行而没有数据交换的，快速的局部合并操作，一般是在进行像Reduce这样的全局合并操作之前，先通过CombineGroup进行小范围的合并操作来减少数据传输量，然后再在全局范围内进行Reduce操作，从而提高效率。  注意到CombineGroup既可以运用在整个数据集上，也可以运用在分组数据集上。 |
| public interface GroupCombineFunction<IN, OUT> extends Function, Serializable {   */\*\*  \* The combine method, called (potentially multiple timed) with subgroups of elements.  \*  \* @param values The elements to be combined.  \* @param out The collector to use to return values from the function.  \*  \* @throws Exception The function may throw Exceptions, which will cause the program to cancel,  \* and may trigger the recovery logic.  \*/* void combine(Iterable<IN> values, Collector<OUT> out) throws Exception; } |

示例：如下所示，是带有CombineGroup的word count，首先通过Combine Group函数将数据集简单的局部聚合一下，然后再用Reduce函数进行全局聚合。注意需要两次使用groupBy函数。

public class \_GroupCombine {  
 public static void main(String args[]) throws Exception {  
  
 ExecutionEnvironment env = ExecutionEnvironment.*getExecutionEnvironment*();  
 env.getConfig().disableSysoutLogging();  
  
 List<String> list = new ArrayList<>(10);  
 list.add("a"); list.add("b"); list.add("c"); list.add("d"); list.add("e");  
 list.add("a"); list.add("f"); list.add("c"); list.add("g"); list.add("e");  
  
 DataSet<String> dataSet = env.fromCollection(list);  
 dataSet  
 .groupBy((KeySelector<String, String>) value -> value)  
 .combineGroup(new GroupCombineFunction<String, Tuple2<String,Integer>>() {  
  
 @Override  
 public void combine(Iterable<String> values, Collector<Tuple2<String, Integer>> out) throws Exception {  
 String key = null; int count = 0;  
 for(String value : values){  
 key = value;  
 count ++;  
 }  
 out.collect(new Tuple2<>(key,count));  
 }  
 })  
 .groupBy(0) //pay attention! we need groupBy again here!  
 .reduceGroup(new GroupReduceFunction<Tuple2<String,Integer>, String>() {  
 @Override  
 public void reduce(Iterable<Tuple2<String, Integer>> values, Collector<String> out) throws Exception {  
 String key = null; int count = 0;  
 for (Tuple2<String, Integer> value : values) {  
 key = value.f0;  
 count += value.f1;  
 }  
 out.collect(key + ":" + count);  
 }  
 })  
 .print()  
 ;  
 }  
}

#### 4.1.1.10 Aggregate [Grouped/Full Tuple DataSet]

Flink提供了如下三种内置聚合操作：

* **Sum**
* **Min**
* **Max**

这些聚合操作只能运用在Tuple DataSet上，而且也只能支持属性位置索引的方式进行分组的数据集。同时这些聚合操作可以运用在分组数据集上，也可以运用在整个数据集上。同理，运用在整个数据集上的效果相当于将整个数据集当成一个分组。示例如下：

public static void main(String[] args) throws Exception {  
 ExecutionEnvironment env = ExecutionEnvironment.*getExecutionEnvironment*();  
 env.getConfig().disableSysoutLogging();  
  
 List<Tuple3<Integer,String,Double>> list = new ArrayList<>(10);  
 list.add(new Tuple3<>(1,"a",0.1)); list.add(new Tuple3<>(2,"a",0.2));  
 list.add(new Tuple3<>(3,"a",0.3)); list.add(new Tuple3<>(2,"b",0.2));  
 DataSet<Tuple3<Integer,String,Double>> dataSet = env.fromCollection(list);  
  
 dataSet  
 .groupBy(1)

// compute sum of the first field  
 .aggregate(Aggregations.*SUM*,0)

// compute minimum of the third field  
// .and(Aggregations.MIN,2)  
 .and(Aggregations.*MAX*,2)  
 .print()  
 ;  
 }

先对Tuple数据集根据第二维进行分组，然后以SUM方式聚合第一维度的值，再以MAX方式聚合第三纬度的值，最后的结果为(6,a,0.3) (2,b,0.2)，如果aggregate运用在一个分组数据集上，则每组返回一个聚集的结果；如果运用在整个数据集上，则只返回一个聚集的而结果。注意 如果希望多个aggregate连用时，用.**and**()函数进行连接；aggregate只能运用在Tuple类型的数据集上。

#### 4.1.1.11 MinBy / MaxBy [Grouped/Full Tuple DataSet]

连续在多个维度的聚合，直至只有一个元素。同理，该聚合函数既可以运用在整个数据集上，也可以运用在分组数据集上，如果运用在分组数据集上，则每组产生一个结果；如果运用在整个数据集上，则整个数据集只产生一个结果。使用规则如下：

*/\*\*  
 \* Applies a special case of a reduce transformation (minBy) on a grouped {****@link*** *DataSet}.<br>  
 \* The transformation consecutively calls a {****@link*** *ReduceFunction}   
 \* until only a single element remains which is the result of the transformation.  
 \* A ReduceFunction combines two elements into one new element of the same type.  
 \*   
 \** ***@param*** *fields Keys taken into account for finding the minimum.  
 \** ***@return*** *A {****@link*** *ReduceOperator} representing the minimum.  
 \*/*@SuppressWarnings({ "unchecked", "rawtypes" })  
public ReduceOperator<T> minBy(int... fields) {  
   
 // Check for using a tuple  
 if(!this.inputDataSet.getType().isTupleType()) {  
 throw new InvalidProgramException("Method minBy(int) only works on tuples.");  
 }  
   
 return new ReduceOperator<T>(this, new SelectByMinFunction(  
 (TupleTypeInfo) this.inputDataSet.getType(), fields), Utils.*getCallLocationName*());  
}

示例如下，先在第一维度上聚合，选出最小值；然后再在输出的结果中，进行第二维度的聚合，继续选出最小值，如果仍然有多个，则只返回第一个复合条件的元素，如下示例代码的输出结果为：(1,a,0.2,0.4)、(2,b,0.2,0.2)

public static void main(String[] args) throws Exception {  
 ExecutionEnvironment env = ExecutionEnvironment.*getExecutionEnvironment*();  
 env.getConfig().disableSysoutLogging();  
  
 List<Tuple4<Integer,String,Double,Double>> list = new ArrayList<>(10);  
 list.add(new Tuple4<>(1,"a",0.2,0.4)); list.add(new Tuple4<>(1,"a",0.2,0.3));  
 list.add(new Tuple4<>(3,"a",0.4,0.4)); list.add(new Tuple4<>(2,"b",0.2,0.2));  
 DataSet<Tuple4<Integer,String,Double,Double>> dataSet = env.fromCollection(list);  
  
 dataSet  
 .groupBy(1)  
 .minBy(0,2)  
// .aggregate(Aggregations.SUM,0)  
// .and(Aggregations.MIN,2)  
// .and(Aggregations.MAX,2)  
 .print()  
 ;  
  
 }

#### 4.1.1.17 Distinct

去掉数据集中的重复元素。该函数需要指定选取去重的key。类似于GroupBy，选取key有三种方法：

* POJO对象的属性选择器；
* Tuple对象的位置选择器；
* 继承KeySelector类实现自定义的选择器。

注意到如果使用如下多维度的去重方法，如下实例所示，则会根据这多个维度是否均相同来判断是否重复。下面的输出结果为：(1,a,0.1)

public static void main(String[] args) throws Exception {  
 ExecutionEnvironment env = ExecutionEnvironment.*getExecutionEnvironment*();  
 env.getConfig().disableSysoutLogging();  
  
 List<Tuple3<Integer,String,Double>> list = new ArrayList<>(4);  
 list.add(new Tuple3<>(1,"a",0.1)); list.add(new Tuple3<>(1,"b",0.1));  
 DataSet<Tuple3<Integer,String,Double>> dataSet = env.fromCollection(list);  
  
 dataSet  
 .distinct(0,2)  
 .print();  
}

#### 4.1.1.18 Join

#### 4.1.1.19 OuterJoin

#### 4.1.1.20 Cross

#### 4.1.1.21 CoGroup

#### 4.1.1.22 Union

#### 4.1.1.23 Rebalance

#### 4.1.1.24 Hash-Partition

#### 4.1.1.25 Range-Partition

#### 4.1.1.26 Sort Partition

#### 4.1.1.27 First-n

# 5. Libraries

## 5.1 Gelly

Gelly是Flink的Graph API。它包含了大量的方法和工具来简化在Flink上做图相关的开发。注意到Gelly目前不是二进制包中的一部分，注意如何执行二进制包中不含有jar包的程序的方法。

### 5.1.1 Graph API

本节的内容主要分为以下6个模块，是Graph API的基础。属于Lower API.



**（1）Graph Representation**

在Gelly中，一个图是由顶点（**Vertex**）和边（**Edge**）构成的集合（**DataSet**）。如下所示：

|  |
| --- |
| 图示例 |
| *// create a new vertex with a Long ID and a String value*  Vertex<Long, String> v = new Vertex<Long, String>(1L, "foo");  *// create a new vertex with a Long ID and no value*  Vertex<Long, NullValue> v = new Vertex<Long, NullValue>(1L, NullValue.getInstance());  Edge<Long, Double> e = new Edge<Long, Double>(1L, 2L, 0.5);  *// reverse the source and target of this edge*  Edge<Long, Double> reversed = e.reverse();  Double weight = e.getValue(); *// weight = 0.5*  ExecutionEnvironment env = ExecutionEnvironment.getExecutionEnvironment();  DataSet<Vertex<String, Long>> vertices = ...  DataSet<Edge<String, Double>> edges = ...  Graph<String, Long, Double> graph = Graph.fromDataSet(vertices, edges, env); |

**（2）Graph Properties**

Graph有相当多的方法来获取图相关的属性

|  |
| --- |
| 图属性示例 |
| *// get the Vertex DataSet*  DataSet<Vertex<K, VV>> getVertices()  *// get the Edge DataSet*  DataSet<Edge<K, EV>> getEdges()  *// get the IDs of the vertices as a DataSet*  DataSet<K> getVertexIds()  *// get the source-target pairs of the edge IDs as a DataSet*  DataSet<Tuple2<K, K>> getEdgeIds()  *// get a DataSet of <vertex ID, in-degree> pairs for all vertices*  DataSet<Tuple2<K, LongValue>> inDegrees()  *// get a DataSet of <vertex ID, out-degree> pairs for all vertices*  DataSet<Tuple2<K, LongValue>> outDegrees()  *// get a DataSet of <vertex ID, degree> pairs for all vertices, where degree is the sum of in- and out- degrees*  DataSet<Tuple2<K, LongValue>> getDegrees()  *// get the number of vertices*  long numberOfVertices()  *// get the number of edges*  long numberOfEdges()  *// get a DataSet of Triplets<srcVertex, trgVertex, edge>*  DataSet<Triplet<K, VV, EV>> getTriplets() |

**（3）Graph Mutations**

图也提供了很多方法来修改图。

|  |
| --- |
| 图的修改示例 |
| *// adds a Vertex to the Graph. If the Vertex already exists, it will not be added again.*  Graph<K, VV, EV> addVertex(final Vertex<K, VV> vertex)  *// adds a list of vertices to the Graph. If the vertices already exist in the graph, they will not be added once more.*  Graph<K, VV, EV> addVertices(List<Vertex<K, VV>> verticesToAdd)  *// adds an Edge to the Graph. If the source and target vertices do not exist in the graph, they will also be added.*  Graph<K, VV, EV> addEdge(Vertex<K, VV> source, Vertex<K, VV> target, EV edgeValue)  *// adds a list of edges to the Graph. When adding an edge for a non-existing set of vertices, the edge is considered invalid and ignored.*  Graph<K, VV, EV> addEdges(List<Edge<K, EV>> newEdges)  *// removes the given Vertex and its edges from the Graph.*  Graph<K, VV, EV> removeVertex(Vertex<K, VV> vertex)  *// removes the given list of vertices and their edges from the Graph*  Graph<K, VV, EV> removeVertices(List<Vertex<K, VV>> verticesToBeRemoved)  *// removes \*all\* edges that match the given Edge from the Graph.*  Graph<K, VV, EV> removeEdge(Edge<K, EV> edge)  *// removes \*all\* edges that match the edges in the given list*  Graph<K, VV, EV> removeEdges(List<Edge<K, EV>> edgesToBeRemoved) |

**（4）Graph Validation**

有时候用户的输入数据可能有误，我们可以检验图是否正确。我们可以自定义一个**GraphValidator**类，然后传入到图的**validate**（）方法中。

**（5）Graph Transformations**

图的变换，我们可以在原图上做各种变换。这些变换包括Map/Translate/Filter/Join/Reverse/Undirected/Union/Difference/Intersect等；

**（6）Neighborhood Methods**

图提供了大量的运算在邻接边或者邻接点上的聚集函数。例如：ReduceOnEdges/reduceOnNeighbors/groupReduceOnEdges/groupReduceOnNeighbors等，这些方法需要定义运算的方向：即In/Out/All(入度，出度，所有)

### 5.1.2 Iterative Graph Processing

Gelly充分利用了Flink的高效的迭代功能来支持大数量级的图相关的迭代运算。目前为止，我们提供了vertex-centric, scatter-gather, and gather-sum-apply三种模型。下面我们将阐述着三种模型，并且描述和展示如何使用它们。

#### 5.1.2.1 Vertex-Centric Iterations

**（1）模型定义**

vertex-centric模型，或者大家所熟知的“像vertex来思考”，提倡我们以顶点为运算单位来计算图。计算在每一步的同步迭代器中进行，我们称之为superstep. 在每一步的superstep中，每个顶点执行一次用户自定义的函数。顶点之间通过传输信息进行交流。一个顶点能够像其他图中的任何顶点发送消息。这种计算模型如下图所示：



图中的虚线框表示一个并行执行单元。在每一步的superstep中，所有活动的顶点（active vertices）都并行执行相同的用户自定义的逻辑。superstep之间是同步执行的（如图所示的一个superstep 一个superstep串行执行），因此在一个superstep执行期间，发送到该节点的消息能够确定在下个superstep开始执行之前确定被送达。（即一个superstep执行期间，不会存在该消息还没有处理完就到了下个superstep了的情况）

**要点：**

**a. 以顶点为单位进行并行执行；**

**b. 同一时刻的多个顶点执行相同的计算逻辑，这些顶点共同构成了一个superstep;**

**c. superstep之间是串行执行的（保证了同步性）。**

如果想使用Gelly中的vertex-centric模型，用户唯一需要做的，就是定义每个顶点的计算函数：**ComputeFunction**. 这个函数和最大迭代次数一起将作为Gelly的**runVertexCentricIteration**函数的参数。该函数将在输入的图中执行vertex-centric迭代并且返回一个顶点值被更改的新图。一个可选的参数**MessageCombiner**（合并消息的函数）用来减少节点之间的通信代价。

让我们使用vertex-centric模型来实现**Single-Source-Shortest-Paths**（单源点最短路径）。初始情况下，除源点外每个顶点都有一个无穷大值，而源点有一个0值（可以看做源点到源点的距离为0）。在第一个superstep中，源点传递它与邻接点的距离值，在接下来的superstep中，每一个顶点都接收它自己所受到的消息（即距离值），并且在这些值中选择一个最小的值，如果这个最小的值比它当前的最小的值小，则更新它的状态并且向它的邻居进行传播。如果一个顶点在superstep中并没有更新自己的值，那么在下一次的superstep中，它不会发送任何消息给邻居。这个算法当没有值被更新或者达到了最大的superstep步数时停止。在这个算法中，message combiner 可以用来减少发送到目标顶点的消息的数量。

**（2）模型使用**

|  |
| --- |
| 代码示例： |
| *// read the input graph*  Graph<Long, Double, Double> graph = ...  *// define the maximum number of iterations*  int maxIterations = 10;  *// Execute the vertex-centric iteration*  Graph<Long, Double, Double> result = graph.runVertexCentricIteration(  new SSSPComputeFunction(), new SSSPCombiner(), maxIterations);  *// Extract the vertices as the result*  DataSet<Vertex<Long, Double>> singleSourceShortestPaths = result.getVertices();  *// - - - UDFs - - - //*  public static final class SSSPComputeFunction extends ComputeFunction<Long, Double, Double, Double> {  public void compute(Vertex<Long, Double> vertex, MessageIterator<Double> messages) {  double minDistance = (vertex.getId().equals(srcId)) ? 0d : Double.POSITIVE\_INFINITY;  for (Double msg : messages) {  minDistance = Math.min(minDistance, msg);  }  if (minDistance < vertex.getValue()) {  setNewVertexValue(minDistance);  for (Edge<Long, Double> e: getEdges()) {  sendMessageTo(e.getTarget(), minDistance + e.getValue());  }  }  }  *// message combiner*  public static final class SSSPCombiner extends MessageCombiner<Long, Double> {  public void combineMessages(MessageIterator<Double> messages) {  double minMessage = Double.POSITIVE\_INFINITY;  for (Double msg: messages) {  minMessage = Math.min(minMessage, msg);  }  sendCombinedMessage(minMessage);  }  } |

**（2）Scatter-Gather Iterations**

**（3）Gather-Sum-Apply Iterations**

### 5.1.2 Library Methods

### 5.1.3 Graph Algorithms

**org.apache.flink.graph.asm**包提供了很多封装好的更高一级的图算法API提供给用户使用。这些算法提供了很多优化方法和配置参数，允许用户在处理相同的输入和相似的参数时能够重用。

回顾这些算法，无非就是以顶点为中心，或者以边为中心进行计算。其实是在图算法API上做了一层封装，提供了很多静态函数，方便用户统计或者修改图的某些属性。

|  |  |  |
| --- | --- | --- |
| Type | Algorithm | Description |
| 有向图  顶点视角 | **VertexInDegree** | 计算有向图所有顶点的入度。  DataSet**<**Vertex**<**K**,** LongValue**>>** inDegree **=** graph  **.**run**(new** **VertexInDegree()**  **.**setIncludeZeroDegreeVertices**(true));**  可选参数：  **setIncludeZeroDegreeVertices**:默认情况下，只有边才纳入计算，当设置这个参数时，会使用连接的方式将入度为0的顶点也加入到输出结果集中。  **setParallelism**:覆盖这个操作的并行度。 |
| **VertexOutDegree** | 计算有向图所有顶点的出度。同上。 |
| **VertexDegrees** | 计算有向图所有顶点的入度和出度。同上。 |
| 有向图  边视角 | **EdgeSourceDegrees** | 计算有向图所有边的起点的入度和出度。 |
| **EdgeTargetDegrees** | 计算有向图所有边的终点的入度和出度。 |
| **EdgeDegreesPair** | 计算有向图所有边的起点和终点的入度和出度。 |
| 无向图  顶点视角 | **VertexDegree** | 计算无向图的所有顶点的度。 |
| 无向图  边视角 | **EdgeSourceDegree** | 计算无向图的所有边的起点的度。 |
| **EdgeTargetDegree** | 计算无向图的所有边的终点的度。 |
| **EdgeDegreePair** | 计算无向图的所有边的起点和终点的度。 |
| 图的修改 | **MaximumDegree** | 过滤一个无向图，使得图中的所有顶点的度不得超过指定的值。 |
| **Simplify** | 针对有向图来说，是用来移除自己指向自己的边和重复的边；  针对无向图来说，是用来移除自己指向自己的边和重复的边，同时添加对称的边（例如a->b,则添加对应的b->a） |
| **TranslateGraphIds** | 根据给定的转换函数，转换图中的顶点的编号。  graph**.**run**(new** **TranslateGraphIds(new** **LongValueToStringValue()));** |
| **TranslateVertexValues** | 根据给定的转换函数，转换图中的顶点的值。  graph**.**run**(new** **TranslateVertexValues(new** **LongValueAddOffset(**vertexCount**)));** |
| **TranslateEdgeValues** | 根据给定的转换函数，转换图中的边的值。  graph**.**run**(new** **TranslateEdgeValues(new** **Nullify()));** |

### 5.1.4 Graph Generators

**org.apache.flink.graph.generator**包中提供了很多图生成相关的API。这些图生成算法有如下特点：

* 是并行的，以便用来创建大数据集；
* 是可以自由扩展的，用户可以自己设置并行度；
* 是精简的，尽量少的使用操作符。

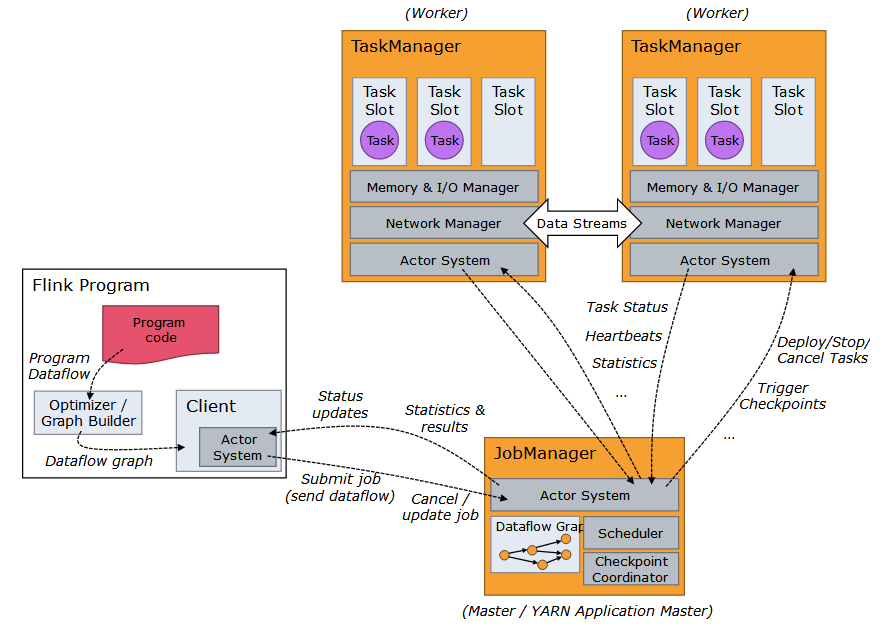
在图生成的时候，算子的并行度是可以由用户来通过**setParallelism(parallelism)**来设置的。并行度设置的越低，申请的内存和对网络资源的占用就越少。

在调用时，首先应该设置参数，然后再调用**generate()**方法。

|  |  |  |
| --- | --- | --- |
| Comlete Graph（完全图） | 每个顶点之间都恰连有一条边的简单图 |  |
| Cycle Graph （循环图） | 顶点连接成一个封闭的链 |  |
| Empty Graph（空图） | 没有边的图 |  |
| Grid Graph（网格图） | 形似右图的图 |  |
| Hypercube Graph（超立方体图） | 形似右图的图 |  |
| Path Graph | 一个无向图，其中所有边形成单个路径 |  |
| RMat Graph | 使用递归矩阵（R-Mat）模型生成的定向或无向功率定律图 |  |
| Singleton Edge Graph | 包含独立双路径的无向图 |  |
| Star Graph（星型图） | 一个无向图，其中一个中心定点有连接到其他所有叶节点的边 |  |

# 6. Internals

## 6.1 General Architecture and Process Model

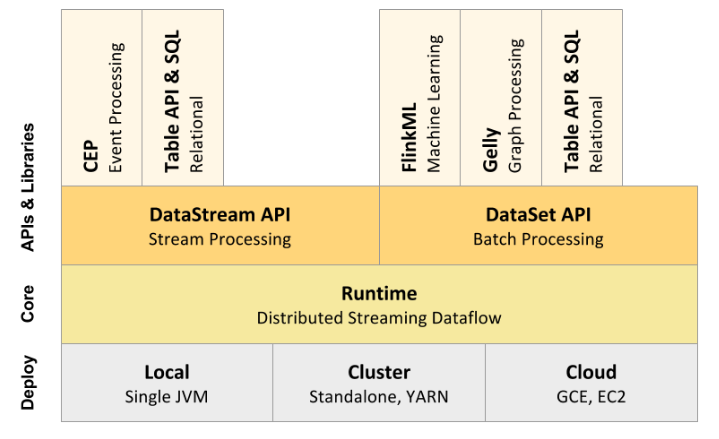


**执行过程（The Processes）**

当Flink系统启动之后，生成一个JobManager和一个或多个TaskManager, JobManager是Flink系统的协调器，而TaskManager是真正用来执行部分并行任务的工作节点。当以本地模式启动Flink时，它会在同一个JVM中启动一个JobManager和一个TaskManager.

当用户提交一个任务时将会创建一个客户端。这个客户端用来预处理程序，使得提交的任务变成并行的data flow，然后放在TaskManager中被执行。

**组件栈（Component Stack）**



**Runtime**: 接收**JobGraph**形式的程序，一个JobGraph是一个通用的带有任意任务的消费和产生数据流的parallel data flow。

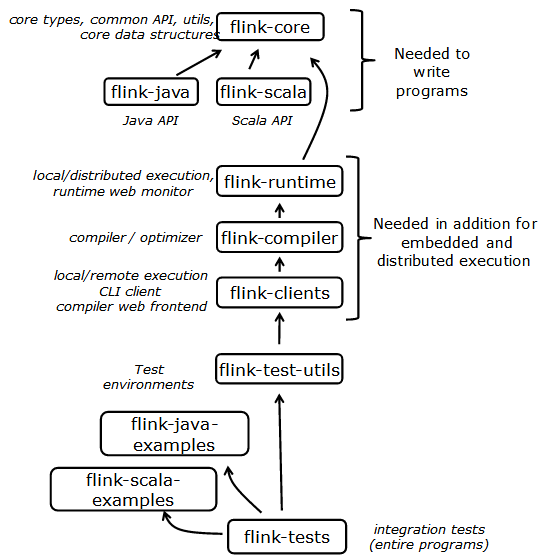
**DataStream API & DataSet API**: 他们通过各自的编译程序，都能生成JobGraph。DataSet API使用了一个优化器来决定程序的优化计划（Optimal Plan），而DataStream API使用了一个流建造器（Stream Builder）。

**Deploy**: JobGraph在不同的部署模式（local, remote, YARN等等）下被执行。

**Libraries & APIs**: 在DataSet和DataStream API之上，构建了很多库函数和常用API。如提供在逻辑表格上查询的Table,提供机器学习API的FlinkML, 提供图处理的Gelly。

**Projects and Dependencies**

Flink系统的代码被分散在不同的子项目中。拆分的目的是为了减少实现Flink项目所需的依赖，并且小的模块更加方便测试。工程之间的依赖如下所示：

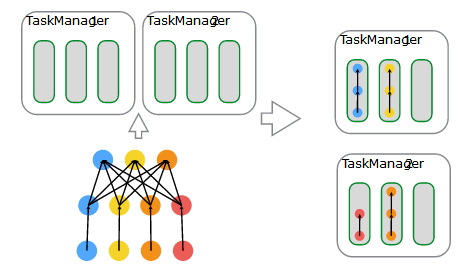


## 6.2 Jobs and Scheduling

**计划（Scheduling）**

在Flink中，资源被定义为**Task Slots**。每个TaskManager拥有一个或多个task slot,每个task slot都可以运行一个并行的任务**管道**（one pipeline of parallel tasks）。一个管道有多个相关联的任务组成，例如并行任务MapFunction中的第n个子任务和ReduceFunction中的第n个子任务可以构成一个管道。注意到Flink通常是并行的执行相关联的任务：对流处理程序来说，在任何情况下都会这样做；对批处理程序来说，并行执行的频率也非常高。

下面用图举例说明。假设有一个程序包含三个部分：数据源、MapFunction、ReduceFunction，数据源和MapFunction的并行度为4，ReduceFunction的并行度为3。一个pipeline是一个Source到Map到Reduce的一个序列。假设该程序在有2个TaskManager（每个TaskManager包含3个slots）的集群上运行，程序的执行结果如下：



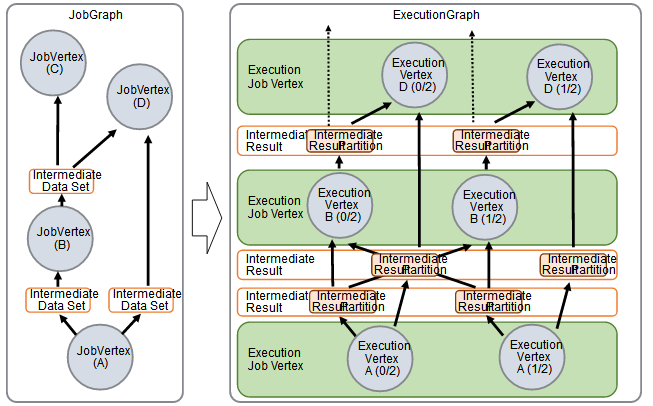
在内部实现时，Flink通过定义SlotSharingGroup和CoLocationGroup这两个类来实现多个task共享一个slot.（物理执行计划的生成，上面的任务是如何划分的）

**数据结构（JobManager Data Structures）**

在Job的执行期间，JobManager记录了分布式任务并决定下一步的任务该何时执行，并且标记任务执行完毕还是执行失败。

JobManager收到的是一个称之为**JobGraph**的数据结构，这个数据结构是一个数据流（dataflow）中的操作（**JobVertex**）和中间结果（**IntermediateDataSet**）的集合。每个操作都有自己的属性，例如并行度和执行逻辑等。而JobGraph也有一系列的在操作被执行时所需要的库。

JobManager将接收到的JobGraph转换成**ExecutionGraph**。这个ExecutionGraph是JobGraph的并行版本。针对JobGraph中的每个JobVertex，一个并行的子任务都对应了一个**ExecutionVertex**。举例来说如果一个操作的并行度为100，那么他将映射成一个JobVertex和100个ExecutionVertex。ExecutionVertex记录了每个子任务的执行状态。这里我们使用**ExecutionJobVertex**来hold住一个JobVertex的所有的ExecutioVertex，这个ExecutionJobVertex就记录了这个操作的整体执行状态。除了有顶点（即操作），ExecutionGraph还包含**IntermediateResult**以及**IntermediateResultPartition**。前者用来记录IntermediateDataSet的状态，侯着用来记录IntermediateDataSet每个部分的状态。如下图所示：

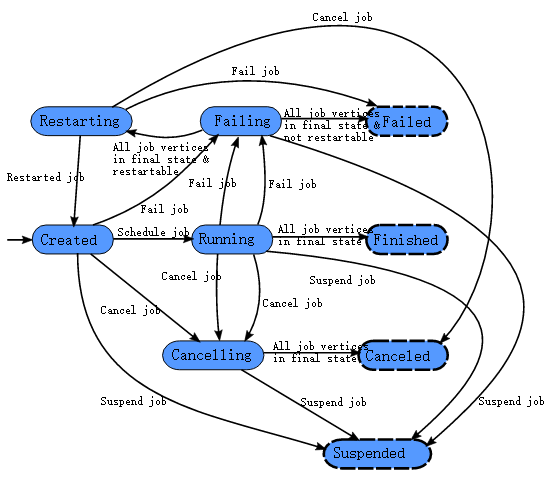


每个ExecutionGraph都附带的有一个（任务状态）job status。这个job status表明当前这个任务的执行状态。

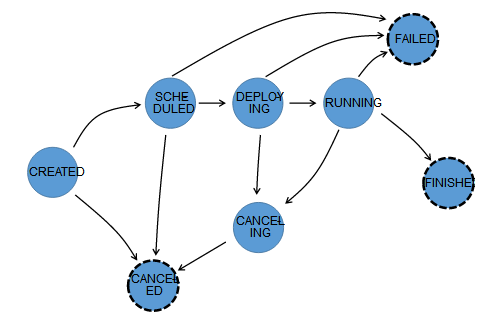
一个Flink的job，首先进入的是已创建状态（**created**），执行完所有的任务之后就会转到已完成状态（**finished**）。如果遇到任务执行失败的情况，job首先会转到失败中的状态（**failing**），并且取消所有正在执行的任务。如果所有的顶点（job vertex）都到达了最终状态而且这个job不能够重启，则这个job将会转到失败状态（**failed**）。如果这个job能够被重启，它将会进入到重启状态（**restarting**），一旦job完全被重启了，他又进入了已创建的状态（created）。

如果用户取消了这个job,它将会进入到取消状态（**cancelling**）。这需要取消所有正在运行的任务。一旦所有正在运行的任务都到达了最终的状态，这个job就会转到已取消的状态（**cancelled**）。

与finished状态不同，cacelled和failed状态会导致一个全局的结束状态，并且会触发针对这个job的清洗（clean up）动作，而暂停（**suspended**）状态指示局部的结束。局部结束意味着这个job的执行被当前的JobManager所终止，但是Flink集群中的其他的JobManager仍然能够在持久化介质中检索到这个job并且重启它。因此一旦一个job进入到了暂停状态,它不会立即被清理出去。



在ExecutionGraph的执行期间，每个并行的task都会尽力好几个状态，从被创建到结束或者执行失败。下面的流程图展示了不同状态之间的转换。一个task可能会被多次执行（例如在失效恢复中重启失败的task）。因此，一个ExecutionVertex的执行过程被**Execution**所跟踪。针对每个ExecutionVertex都有一个当前的Execution和先前的Execution.

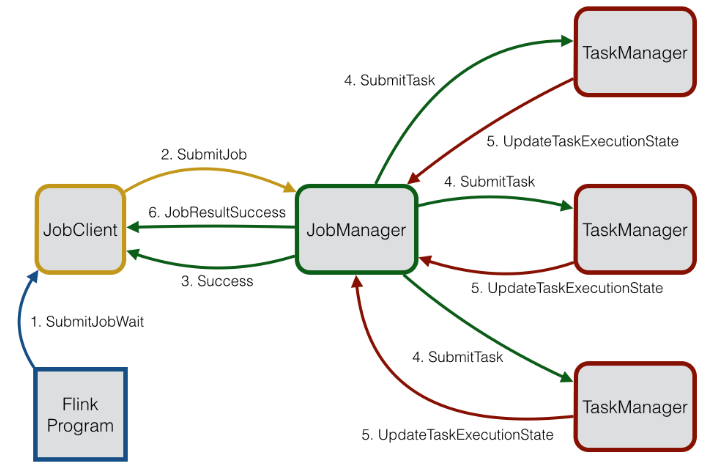


## 6.3 Akka & Actor System

译文：<http://blog.csdn.net/yanghua_kobe/article/details/51156218>

原文：<https://cwiki.apache.org/confluence/display/FLINK/Akka+and+Actors>

Flink的内部通信机制是通过Akka来实现的。通过Akka的使用，所有的远程调用都通过异步消息来实现。例如Flink中的JobManager, TaskManager和JobClient三者之间的通信主要是用Akka实现的。如下图所示：



JobClient接收用户提交的Flink Job,然后将它再提交给JobManager，Jobmanager针对该Job制定合理的执行方案，首先，它将分配执行Job所需的足够的资源，这些资源包括TaskManger中的slot。资源分配完成后，JobManager将这些独立的子任务分配到TaskManager中，TaskManager收到一个子任务之后，就会开启一个线程来执行该任务，并且将任务的执行状态（例如正在执行或执行完毕）返回给JobManager。JobManager根据这些反馈回来的状态信息来控制Job的执行流程，直至它执行结束。一旦这个job执行结束，JobManager将会将任务的执行结果反馈给JobClient，这样就可以告知用户job的执行结果。

## 6.4 Data Exchange between Tasks

原文：<https://cwiki.apache.org/confluence/display/FLINK/Data+exchange+between+tasks>  
译文：

<http://mp.weixin.qq.com/s?__biz=MzI0NTIxNzE1Ng==&mid=2651214945&idx=1&sn=e3a9f31898694de37b7c7ba100f56d8d&scene=19#wechat_redirect>

Flink的数据交换遵循如下原则：

1. 数据交换的控制流（例如初始化交换的信息）是在接收端初始化的，就像原始的MapReduce;

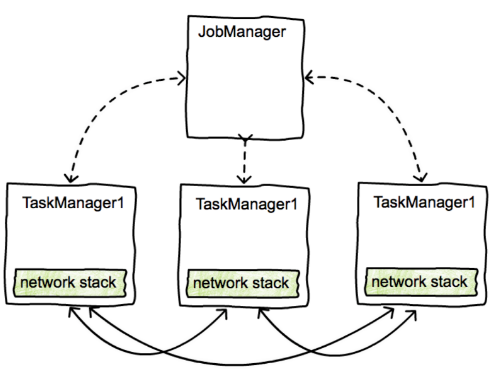
2. 数据交换的数据流（在网络上最终传输的数据）被抽象成一个IntermediateResult概念，他是可插拔的。这意味着系统基于相同的实现逻辑，既可以支持流数据的传输，又可以支持批数据的传输。

数据传输涉及一下几个组件：

**JobManager**: 主节点，主要用来任务调度，失效恢复和协调资源，通过ExecutionGraph这个数据结构拥有一个job的整个蓝图。

**TaskManager**:工作节点，一个TaskManager通过多线程的方式来并发执行多个任务。每个TaskManager都包含一个CommunicationManager和一个MemoryManager（即同个TaskManager中的多个任务共享这两个组件）。TaskManager之间的数据交换是通过TCP连接来实现的。

值得注意的是，在Flink中，是TaskManager而不是tasks在网络上交换数据。而在同一个TaskManager中的不同任务是通过多路复用来实现的。

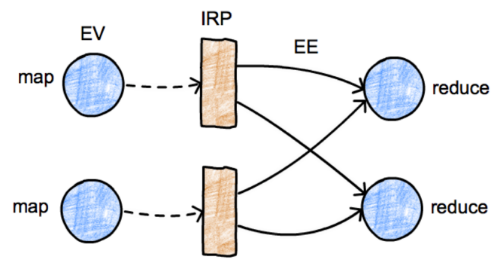


**ExecutionGraph**:这个数据结构其实是job计算时真正用到的数据结构。它包含

（1）若干个顶点（**ExecutionVertex**），代表计算任务；

（2）中间结果（**IntermediateResultPartition**），代表任务产生的数据。

（3）边（**ExecutionEdge**），连接顶点和中间结果。



这是JobManager中保存的逻辑数据结构。他们都有各自对应的运行时的数据结构来保存真正的数据处理结果，而这些是放在TaskManager。例如IntermediateResultPartition对应的运行时刻的数据结构称之为ResultPartition。

**ResultPartition**：代表一个由BufferWriter写入的一个数据块。一个ResultPartition是多个ResultSubpartition的集合。这是为了来区别不同接收端的数据。例如针对一个reduce或者join操作的shuffle.

**ResultSubpartition**：代表一个操作（operator）创建的一个数据分区，和逻辑一起传给下一个接收操作（receiving operator）。ResultSubpartition的实现方式决定了数据的传输逻辑，这个是这种插件化设计方式使得系统支持各种各样的数据传输需求。例如，**PipelinedSubpartition**是一个pipelined的实现用来支持流数据的交换，而SpillableSubpartition是一种blocking的实现用来支持批数据的交换。

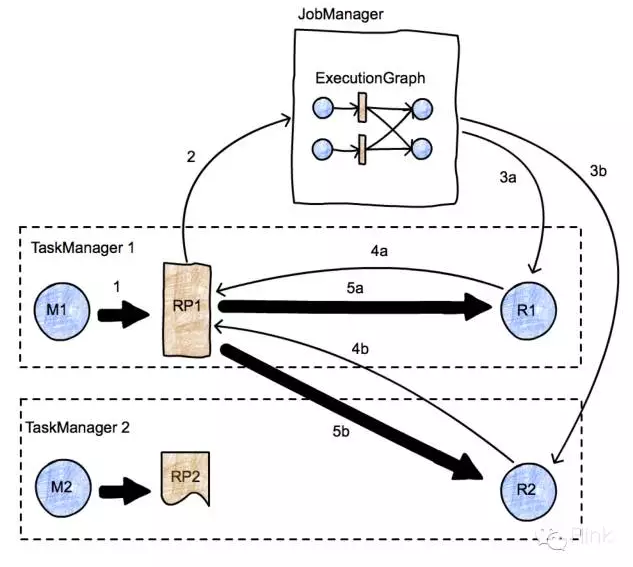
**InputGate**: 在接收端，逻辑上等价于RP。它用于处理并收集来自上游的buffer中的数据。

**InputChannel**: 在接收端，逻辑上等价于RS。用于接收某个特定的分区的数据。

**Buffer**: 参见memory-management

序列化器、反序列化器用于可靠得将类型化的数据转化为纯粹的二进制数据，处理跨buffer的数据。

**数据交换中的控制流**



这个图展示了一个并行度为2的Map-Reduce任务中数据是如何进行交换的。在这里我们有两个TaskManager(TM)，每个TM中有两个子任务（一个是Map,另外一个是Reduce）在运行，还有一个JobManager(JM)在第三个节点上运行。我们重点关注任务M1和R2之间是如何完成数据交换的。在途中，数据交换使用的是粗箭头，信息交换使用的是细箭头。首先，M1产生了一个ResultPartition(RP1)（如箭头1所示）。当这块数据可用的时候，它会通知JobManager（如箭头2所示），然后JobManager会通知这块数据的接收者（R1和R2）数据已经可以被消费了。如果接收者还没有启动，那么这条消息将会触发这两个接收者的部署（如箭头3a，3b所示）。然后接收者向RP发出数据请求（如箭头4a,4b所示）。这将会启动数据传输（如箭头5a,5b所示），可以看出这种数据传输既可能是本地传输（如5a），也可能是通过TaskManager的网络栈传输（如5b）。在这期间，RP完全可以决定何时来通知JM自己已经可以使用了。例如如果RP1中的数据完全产生完之后再通知JM，这就类似批处理的数据交换了，和Hadoop的实现一样。如果RP1刚产生完一条数据就赶快通知JM，这就是流式数据交换。

数据是pull还是push模式。

## 6.5 Data Types and Serialization

# 7. Monitoring & Debugging

# # Preview

**函数式编程**

**Maven打Jar包（今天解决这个问题）：**[**https://ci.apache.org/projects/flink/flink-docs-release-1.2/dev/cluster\_execution.html**](https://ci.apache.org/projects/flink/flink-docs-release-1.2/dev/cluster_execution.html)

# # Programming

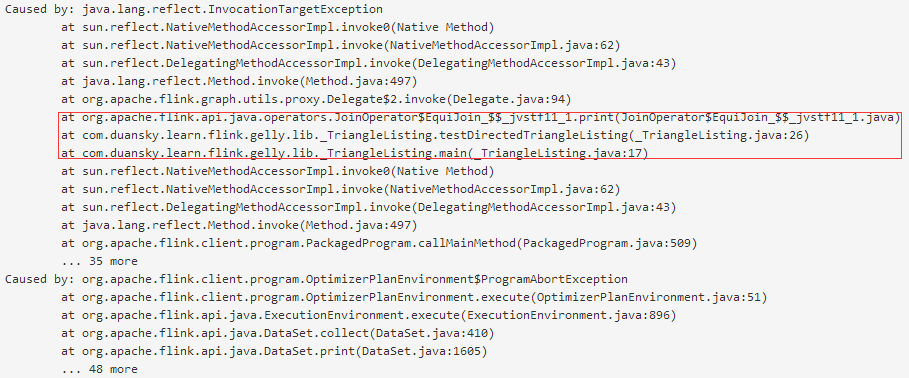
1.如果提交给集群中的任务，包含一些原生集群中不存在的类（例如这些类是用户自定义的）则可能导致任务无法执行，并报错：

java.lang.RuntimeException: The initialization of the DataSource's outputs caused an error: Could not read the user code wrapper: com.duansky.learn.flink.\_DataSet$LineSplitter

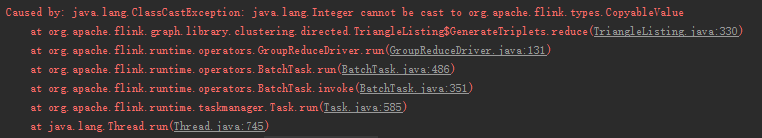
所以一般情况下，比较靠谱的方式是将自己的程序打成jar包，然后上传到集群中进行运行。

2. 如果启动过程中报class Not found，多半情况下是因为Maven依赖没有下完整，检查对应的Maven依赖包是否完全下载。

3. Flink在打印DataSet中的数据集时，数据集中的定义必须是LongValue,而使用原生态的Long就会报not CopyableValue 错误。



4. 同问题3，原因是DataSet的类型必须是XXXValue类型的。基本类型不可用？



5. 如果提交的程序申请内存过多，会直接导致整个集群崩掉。

# # 补充

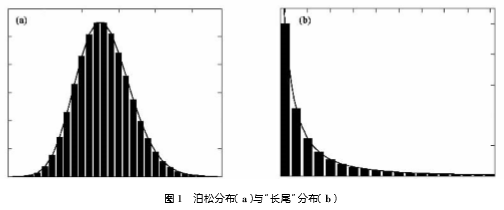
## #1. 幂律分布

参考论文：幂律分布研究简史.

**泊松分布**：泊松分布适合于描述单位时间内随机事件发生的次数的概率分布。

如果以人的身高为横坐标，以该身高的人数或概率为纵坐标，绘制的曲线如图(a)所示，复合泊松分布。

**长尾分布**：“长尾”分布表明，绝大多数个体的尺度很小，而只有少数个体的尺度相当大。如果以收入为横坐标，以不低于该收入的个体数或概率为纵坐标，绘制的曲线如图（b）所示，复合长尾分布。



**Zipf定律**：把单词出现的频率按照由大到小排列，则每个单词出现的频率和它的名次的常数次幂存在简单的反比关系：。该定律说明只有极少数的单词被频繁的使用，而绝大多数词很少使用。

**Pareto定律**：19世纪意大利经济学家Pareto研究了个人收入的统计分布，提出了80/20原则，即20%的人占据了80%的财富。个人收入X不小于某个特定值x的概率与x的常数次幂亦存在简单的反比关系

Zipf定律和Pareto定律都是简单的幂函数，称之为幂律分布。通式可以写成

其中x,y是正的随机变量，c,r均为大于0的常数。这种分布的共性是**绝大多数事件规模比较小，只有少数事件的规模比较大。**

**图算法中的幂律分布：**图中的度的分布符合幂律分布。

## #2. Vertex-centric iterations

该迭代模型属于Bulk Synchronous Parallel模型。这个计算模型也被Google Pregel和Apache Giraph 所使用。

关于Bulk Synchronous Parallel模型(整体同步并行计算模型/BSP模型)：

<https://en.wikipedia.org/wiki/Bulk_synchronous_parallel>

BSP模型是并行计算中的3中基本模型之一。其他两种分别是PRAM模型和LogP模型。

BSP模型主要由如下三部分组成：

* 能够处理内存事务的组件，比如处理器
* 在组件之间进行消息路由的网络
* 能够实现所有组件之间或者组件子集之间的同步的硬件设施

BSP的概念由Valiant(1990)提出的“块”同步模型，是一种异步MIMD-DM模型，支持消息传递系统，块内异步并行，块间显式同步。

BSP计算模型不仅是一种体系结构模型，也是设计并行程序的一种方法。BSP程序设计准则是 bulk同步 (bulk synchrony)，其独特之处在于**超步**(superstep)概念的引入。一 个BSP程序同时具有水平和垂直两个方面的结构。从垂直上看,一个BSP程序由一系列串行的超步(superstep)组成，这种结构类似于一个串行程序结构。从水平上看， 在一个超步中， 所有的进程并行执行局部计算。