

目录

Learn Flink.....	3
1. Introduction	3
2. Concepts.....	3
2.1 Programs and Dataflows	3
2.1.1 并行的流（Parallel Dataflows）	4
2.1.2 任务和操作链（Tasks & Operator Chains）	5
2.2 Distributed Execution.....	6
2.2.1 Master, Worker, Client.....	6
2.2.2 Workers, Slots, Resources	7
2.3 Time and Windows.....	8
2.3.1 Time.....	8
2.3.2 Windows	9
2.4 State and Fault Tolerance 状态保持和错误容忍	10
3. Quick Start.....	10
3.1 Setup	10
3.1.1 Basic Setup	10
3.1.2 Run Example.....	11
3.1.3 Cluster Setup	12
3.2 Execution.....	12
3.2.1 Local Execution.....	12
3.2.2 Cluster Execution.....	14
4. APIs.....	15
4.1 Basic API Concepts	15
4.1.1 Anatomy of a Flink Program.....	15
4.1.2 Accumulators	15
4.2 Streaming (DataStream API).....	16
4.2.1 DataStream Transformations	16
4.2.2 Data Sources.....	19
4.2.3 Data Sinks	20
4.2.4 Iterations	20
4.2.5 Controlling Latency.....	20
4.3 Batch (DataSet API)	20
4.3.1 DataSet Transformations.....	20
4.3.2 Data Sources.....	43
4.3.3 Data Sinks	43
4.3.4 Iteration Operators.....	43
4.3.5 Operating on data objects in functions.....	44
4.4 ExecutionEnvironment	45
5. Libraries.....	45
5.1 Gelly	45
5.1.1 Graph API	45
5.1.2 Iterative Graph Processing	48

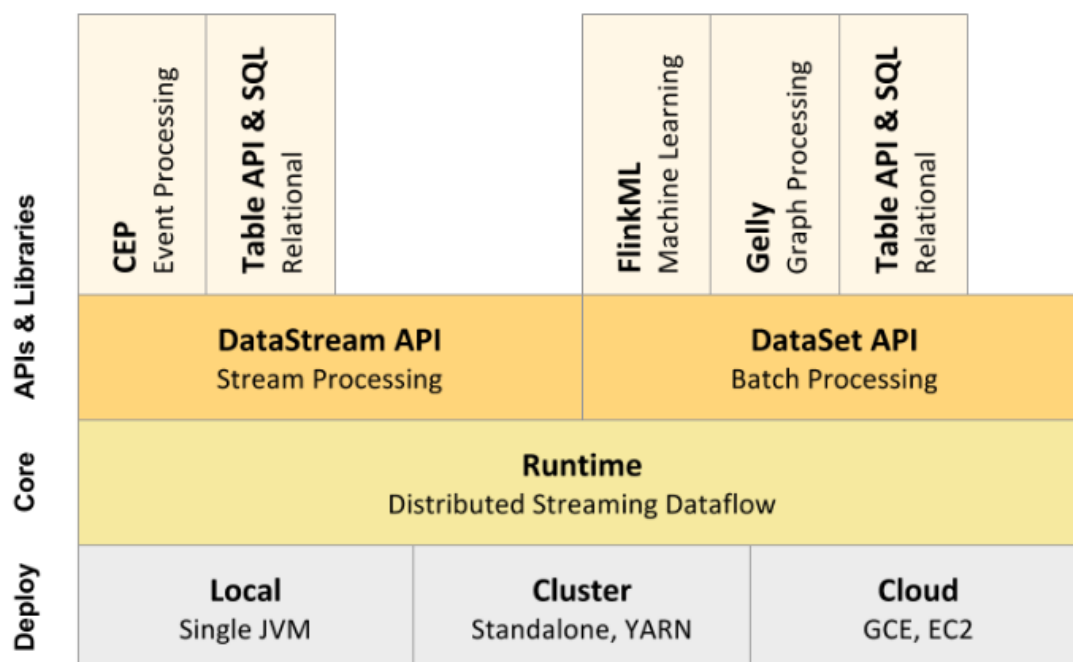
5.1.2 Library Methods	52
5.1.3 Graph Algorithms	52
5.1.4 Graph Generators.....	53
6. Internals	55
6.1 General Architecture and Process Model	55
6.2 Jobs and Scheduling	57
6.3 Akka & Actor System	61
6.4 Data Exchange between Tasks.....	61
6.5 Data Types and Serialization	64
7. Monitoring & Debugging.....	65
8. Source Analysis.....	65
# Preview.....	65
# Programming.....	65
# 补充.....	66
#1. 幂律分布.....	66
#2. Vertex-centric iterations.....	67

Learn Flink

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：

```

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));

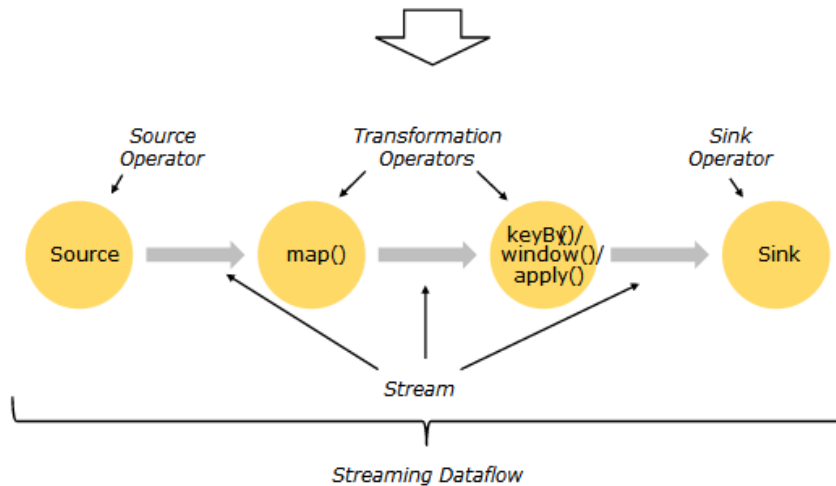
```

Source

Transformation

Transformation

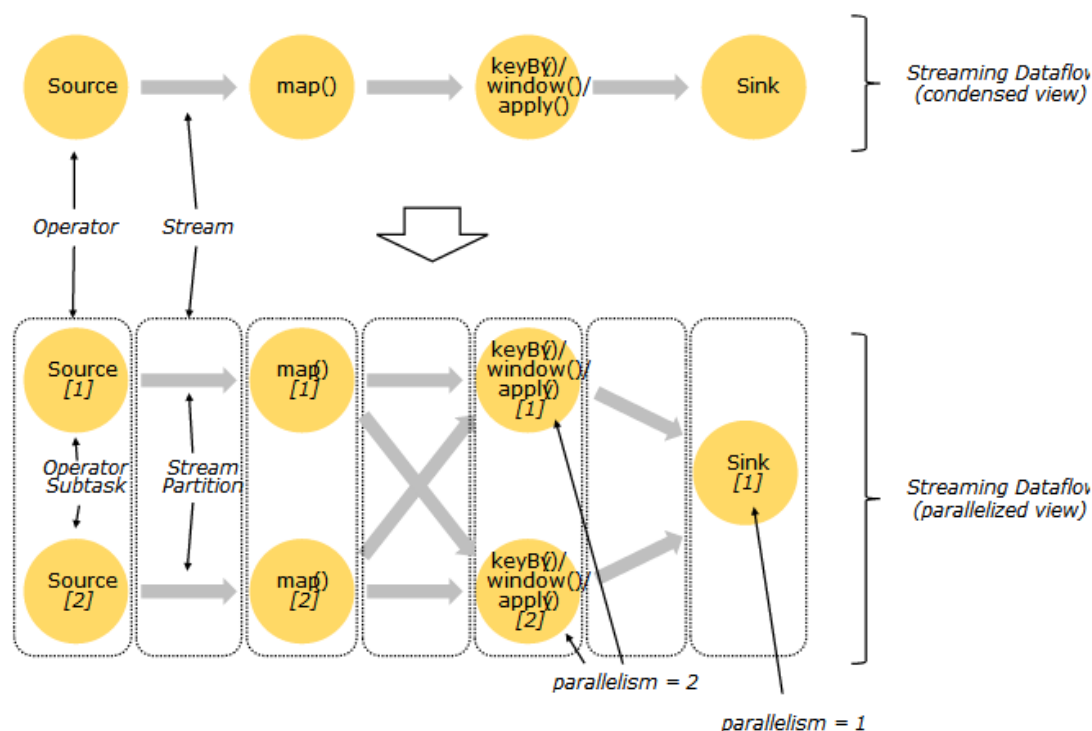
Sink



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（operator 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。

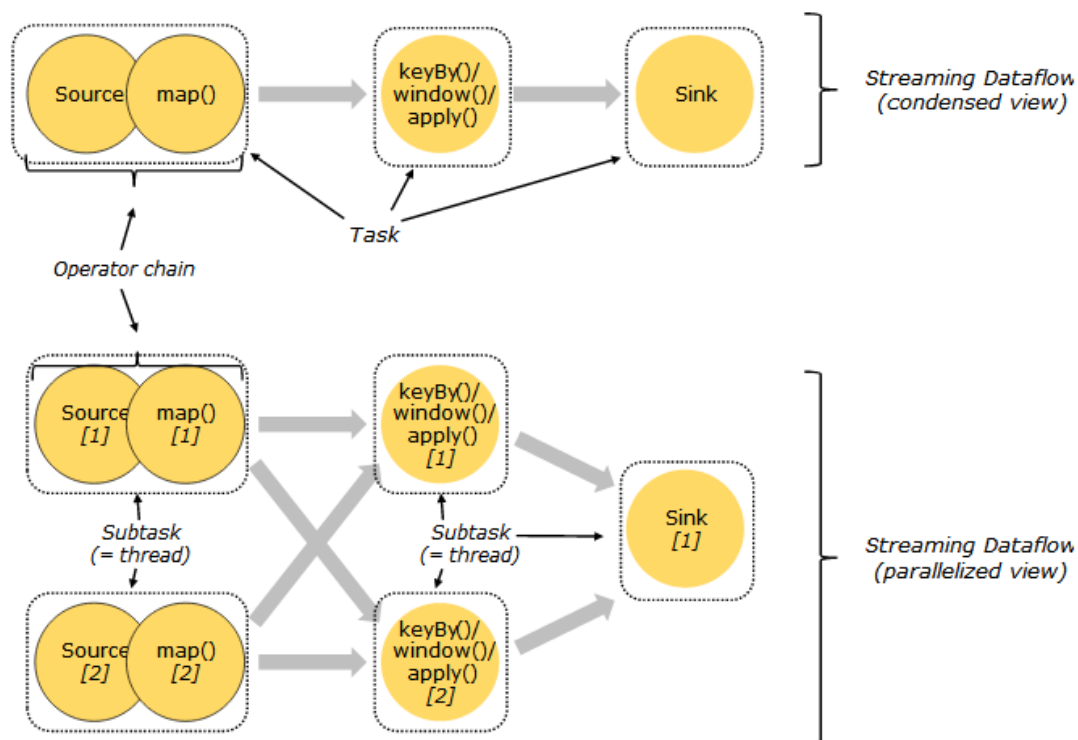
由上图可以看出节点时操作，连接是数据。stream 连接不同的 operator。如上所示，流可以和 operator 是一一对应的关系，也可以是从新分配的关系。

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

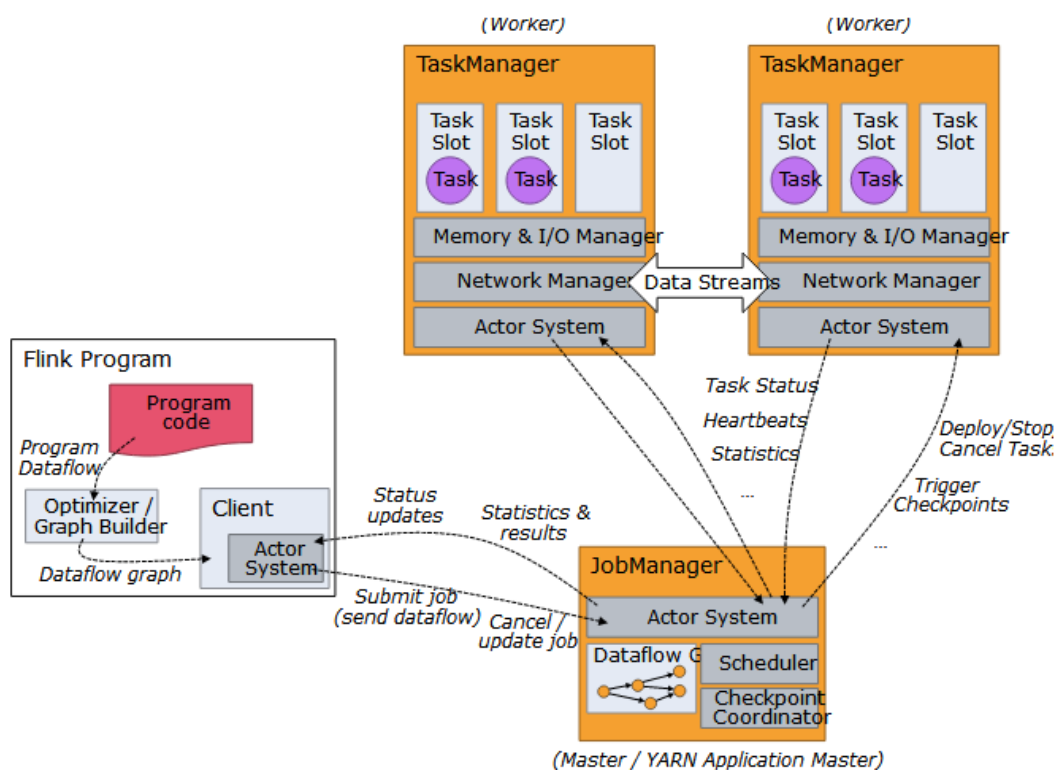
2.2.1 Master, Worker, Client

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。

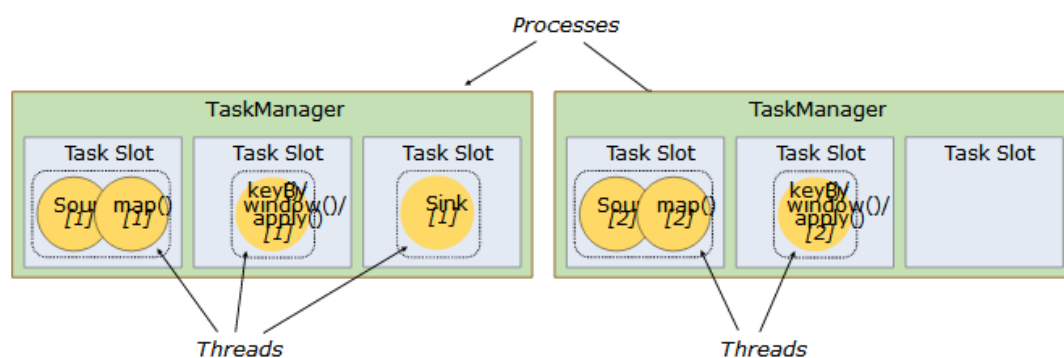
共享 **slot** 的操作可以通过 **API** 来设置，一般说来，当 **task slots** 的数量和 **CPU** 核数相同

时最佳。

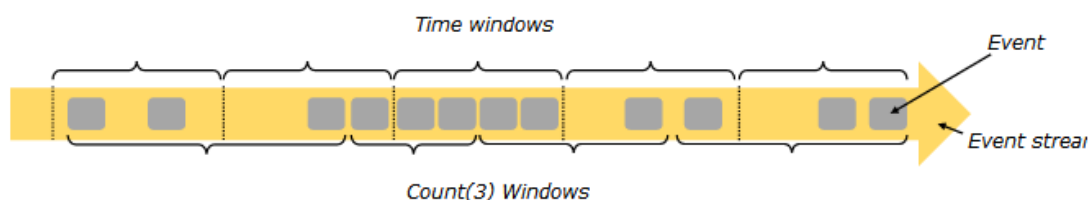
2.3 Time and Windows

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: **tumbling 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>

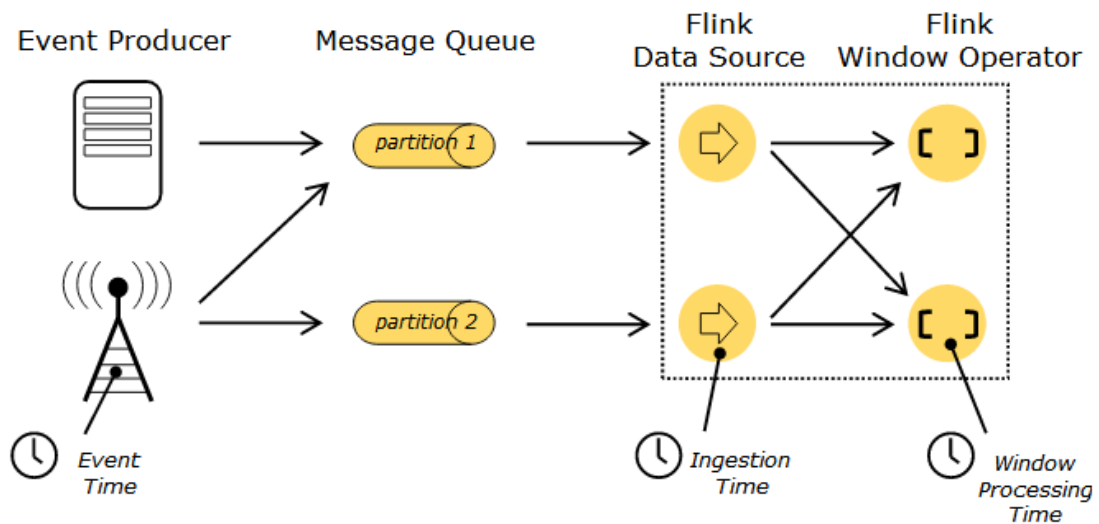
2.3.1 Time

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 能够通过 **timestamp assigners** 获取事件的时间戳。

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

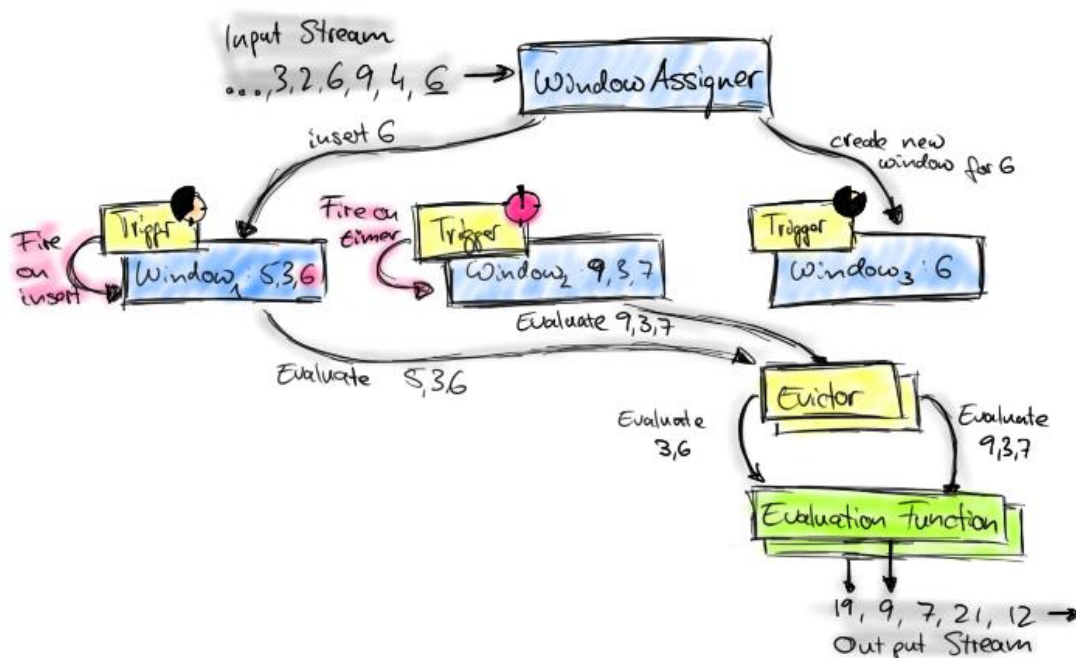
2.3.2 Windows

Blog: <https://flink.apache.org/news/2015/12/04/Introducing-windows.html>

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

Flink 使用 Window 操作将原来的无限的 DataStream 根据时间戳或其他标准来切割成有限的一个一个的切片，其中一个切片就是一个 window。这种切分的意义使得在无限的 DataStream 上使用聚合操作（**aggregate**）成为可能。

大多数情况下我们谈论的都是 Keyed Windowing。即 windows 是在 KeyedStream 上进行切割的。Keyed Windows 有诸多好处，比如在启动用户自定义的函数前，元素被按照 key 和 window 进行了划分，这样不同的 key 对应的 window 能够被独立并行的执行。除非有必要你才使用不经过 key 划分的 window。



Basics

一个 window 操作你至少需要一个 key，一个 window assigner 和一个 window function。

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

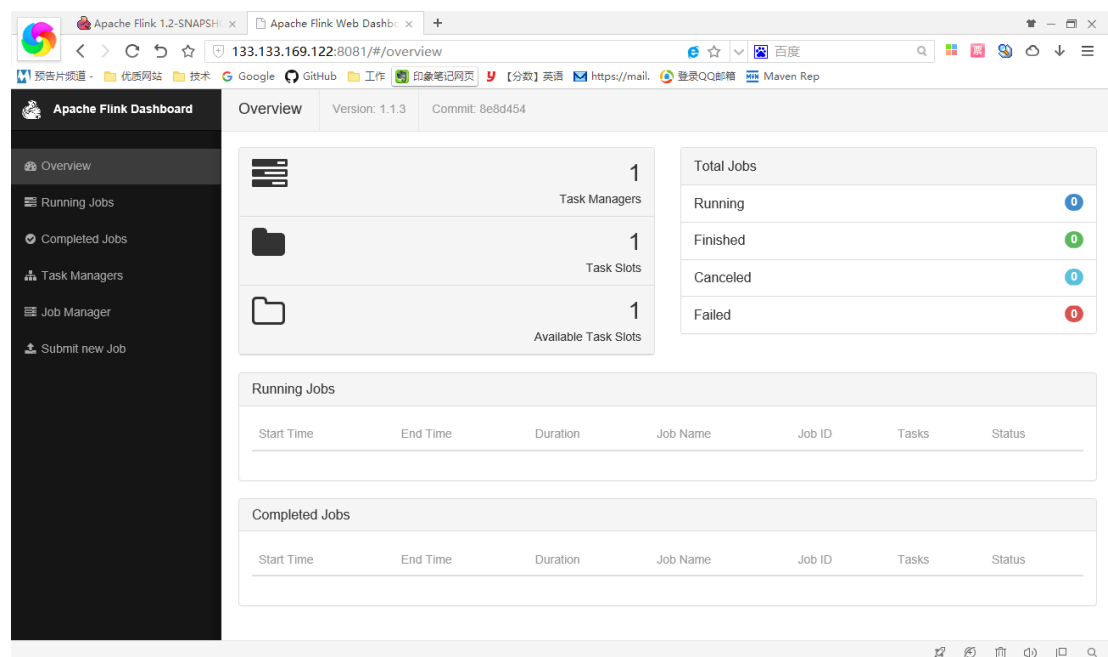
3.1.1 Basic Setup

(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：

The screenshot shows the Apache Flink Dashboard interface. The left sidebar contains navigation links: Overview, Running Jobs, Completed Jobs, Task Managers, Job Manager, and Submit new Job. The main content area is titled 'Overview' and shows the following metrics:

- Task Managers: 1
- Task Slots: 1
- Available Task Slots: 0

On the right, there is a 'Total Jobs' summary table:

Job Status	Count
Running	1
Finished	0
Canceled	0
Failed	0

Below this, the 'Running Jobs' table is displayed:

Start Time	End Time	Duration	Job Name	Job ID	Tasks	Status
2016-10-25, 22:46:12	2016-10-25, 22:49:50	3m 37s	Socket Window WordCount	acffe6e9a1bd8262f017479d1f8f068f	2/0/2/0/0/0	RUNNING

The 'Completed Jobs' section at the bottom is currently empty.

3.1.3 Cluster Setup

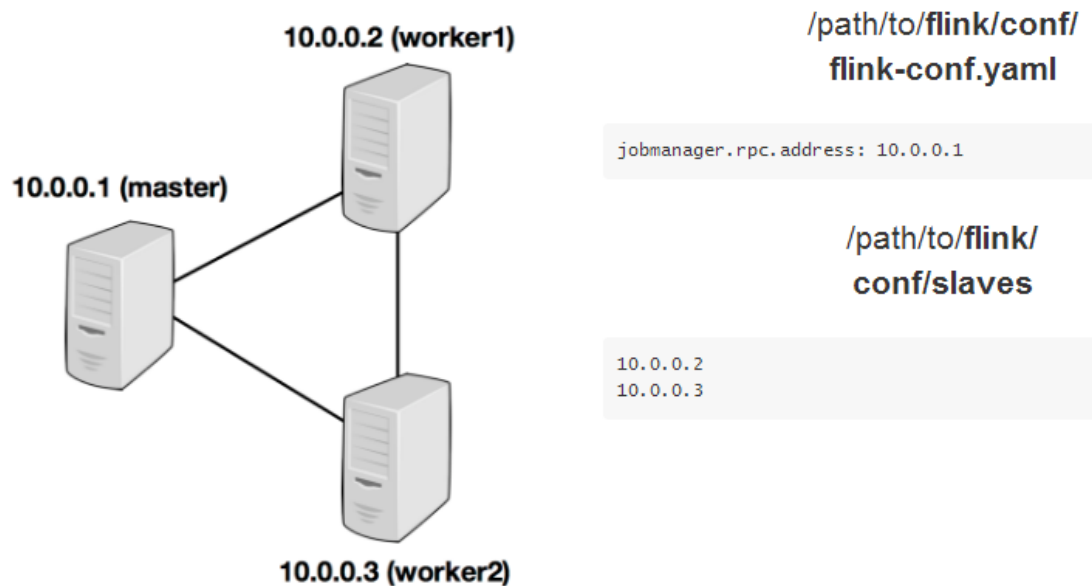
1. 配置主从节点

(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.createLocalE
nvironment(conf);
```

Local Environment 示例：

```
public static void main(String[] args) throws Exception {
    ExecutionEnvironment env = ExecutionEnvironment.createLocalE
nvironment();
    DataSet<String> data = env.readTextFile("file:///path/to/fil
e");
    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 fro
m 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

4.1.2 Accumulators

4.2 Streaming (DataStream API)

完整的原文：

https://ci.apache.org/projects/flink/flink-docs-release-1.2/dev/datastream_api.html

4.2.1 DataStream Transformations

4.2.1.1 Map

`DataStream` → `DataStream`

一对一的映射。将数据流中的一个元素映射成另外一个元素。

4.2.1.2 FlatMap

`DataStream` → `DataStream`

一对多的映射。将数据流中的一个元素映射成 0 个、1 个或多个元素。

4.2.1.3 Filter

`DataStream` → `DataStream`

过滤掉数据流中不符合要求的元素。

4.2.1.4 KeyBy

`DataStream` → `KeyedStream`

将一个流划分成不想交的几个部分（不想交是指数据没有交集）。每个部分都包含相同的 key 值。在内部这种划分是通过 hash 算法来进行划分的。

4.2.1.5 Reduce

`KeyedStream` → `DataStream`

运行在一个 `keyed data stream` 上的 `reduce` 操作。能够将当前的元素和上一次 `reduce` 结果的元素进行聚合，产生一个新的元素。

4.2.1.6 Fold

`KeyedStream` → `DataStream`

运行在一个 `keyed data stream` 上的带有初始值的 `fold` 操作。能够将当前的元素和上一

次 fold 结果进行聚合，产生一个新的元素。与 Reduce 不同在于该操作有个初始值。

4.2.1.7 Aggregations

KeyedStream → DataStream

运行在一个 keyed data stream 上的 aggregation 操作。例如 sum, min, max, minBy, maxBy 等。注意到 min 和 minBy 的区别是 min 返回的是指定域上的最小值，而 minBy 返回的是指定域上最小值的那个元素。（max 和 maxBy 同理）

4.2.1.8 Window

KeyedStream → WindowedStream

运行在一个 keyed data stream 上的 window 操作。window 操作能够将每个以 key 分组中的元素按照某种规则（例如每 5s 或每 10 个元素）再次进行窗口聚合。

4.2.1.9 WindowAll

DataStream → AllWindowedStream

Window 操作也可以直接运行在 data stream 上，这就是 windowAll 操作。这个操作能够将数据流按照某种规则（例如每 5s 或每 10 个元素）进行窗口聚合。

4.2.1.10 Window Apply

WindowedStream → DataStream

AllWindowedStream → DataStream

可以在整个 window 窗口内执行某个函数。注意运行在 Window 上的函数是 WindowFunction，运行在 windowAll 上的函数是 AllWindowFunction。

4.2.1.11 Window Reduce

WindowedStream → DataStream

运行在一个 window stream 上的 reduce 操作。将整个 window 中的元素聚合成一个元素。聚合的方式是两两聚合。

4.2.1.12 Window Fold

WindowedStream → DataStream

运行在一个 window stream 上的 fold 操作。fold 操作是一个迭代的操作。每次使用上次结果和当前的元素进行组合，然后下次迭代时就会使用到该计算结果。而且 fold 操作带

有初始值。

4.2.1.13 Aggregations on windows

WindowedStream \rightarrow DataStream

Aggregation 操作也可以运行在 window 上。类似 Aggregations on data stream.

4.2.1.14 Union

DataStream* \rightarrow DataStream

可以将多个流合并起来组成一个流。

4.2.1.15 Window Join

DataStream, DataStream \rightarrow DataStream

Join 操作运行在两个数据流的 window 上。

```
dataStream.join(otherStream)
    .where(0).equalTo(1)
    .window(TumblingEventTimeWindows.of(Time.seconds(3)))
    .apply (new JoinFunction () {...});
```

4.2.1.16 Window CoGroup

DataStream, DataStream \rightarrow DataStream

CoGroup 也可以运行在 data stream 的 window 上。

4.2.1.17 Connect

DataStream, DataStream \rightarrow ConnectedStreams

连接两个不同的流，并且保留他们的类型信息。Connect 是为了共享他们的状态信息。

4.2.1.18 CoMap, CoFlatMap

ConnectedStreams \rightarrow DataStream

运行在 ConnectedStreams 上的 CoMap 和 CoFlatMap。

4.2.1.19 Split

DataStream → SplitStream

根据某种标准，将原来的 data stream 分割成 2 个或更多的 stream.

4.2.1.20 Select

SplitStream → DataStream

从 SplitStream 中选取一个或者多个流组合成一个 data stream.

4.2.1.21 Iterate

DataStream → IterativeStream → DataStream

实现了循环的流。

4.2.1.22 Extract Timestamps

提取一个流中的时间元素以便后面的 window 操作根据此进行划分 window.

4.2.1.23 Project

运行在 Tuple data stream 上的投影操作。

4.2.1.24 Physical partitioning

Custom partitioning: 使用用户自定义的分片规则对数据进行分片 -> `.partitionCustom()`

Random partitioning: 使用均匀策略对数据进行分片 -> `.shuffle()`

Rebalancing (Round-robin partitioning): 使用 round-robin 算法划分 -> `.rebalance()`

Rescaling: 根据上下流的并行度重新划分数据 -> `.rescale()`

Broadcasting: 将元素广播到各个操作 -> `.broadcast()`

4.2.2 Data Sources

用户可以实现 `SourceFunction` 来自定义数据源，也可以通过 `StreamExecutionEnvironment` 提供的创建数据源的方法，例如 `readTextFile(path)`，`readFile(fileInputFormat, path)`等。不再赘述。

4.2.3 Data Sinks

把数据流中的数据写入到文件、socket、其他的系统或者仅仅打印他们。即消费数据流。DataStream API 提供了大量的函数用来 sink data.例如 `writeAsText()/writeAsCsv(...)`

4.2.4 Iterations

实现了迭代计算的能力。

4.2.5 Controlling Latency

默认情况下，Flink 中的元素传输不是一个一个的，而是一个 buffer 一个 buffer 的。使用 buffer 进行传输能够显著提高吞吐率，但另外一方面，这也增加了延迟，因为可能需要等待很久才能将一个 buffer 填满。为此，Flink 提供了接口允许用户设置一个 buffer 的 timeout 时间，即经过多长时间就发送该 buffer 数据，即使该 buffer 没有填满。如下所示：

```
LocalStreamEnvironment env = StreamExecutionEnvironment.createLocalEnvironment();
env.setBufferTimeout(timeoutMillis);

env.generateSequence(1,10).map(new MyMapper()).setBufferTimeout(timeoutMillis);
```

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->

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。Partition 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);
    }
}

```

4.1.1.4 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>
     * IMPORTANT: 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;
    }
}

```

```
}
```

4.1.1.5 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 loses 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, In
teger>
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：

- if (type instanceof CompositeType)**
- 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。
- 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 times) 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

Join 操作可以将两个数据集合并成一个数据集。这两个数据集在 join 时使用的 key 的选取方式可以有如下几种：

- a key expression
- a key-selector function
- one or more field position keys (Tuple DataSet only).
- Case Class Fields

(1) Default Join (Join into Tuple2)

默认情况下，通过 Join 操作，会产生一个新的 Tuple2 类型的数据集。它有两个属性域，其中一个是第一个数据集中的数据，另一个是匹配上第一个元素 key 值的第二个数据集类型中的数据。使用规则如下：

```
public static class User { public String name; public int zip; }
public static class Store { public Manager mgr; public int zip; }
DataSet<User> input1 = // [...]
DataSet<Store> input2 = // [...]
// result dataset is typed as Tuple2
DataSet<Tuple2<User, Store>>
    result = input1.join(input2)
    .where("zip")           // key of the first input (users)
    .equalTo("zip");       // key of the second input (stores)
```

结果集是两个数据集在某个 key 上的交集。

(2) Join with Join Function

Join 操作也允许用户自定义 Join 的处理逻辑，而不是像默认情况下返回一个包含这两种数据类型的 Tuple（见（1）Default Join）。Join Function 接收第一个和第二个数据集各一个元素，并且返回一个元素。Join Function 是在两个数据集的元素在指定的 Key 上匹配之后触发。

JoinFunction

当来自不同数据集的两个元素在指定的 key 上 match 之后，便会触发 JoinFunction 的执行。JoinFunction 接收来自两个数据集各自一个元素，然后输出一个元素。如下所示：

```
public interface JoinFunction<IN1, IN2, OUT> extends Function,
    Serializable {

    /**
     * The join method, called once per joined pair of elements.
     *
     * @param first The element from first input.
     * @param second The element from second input.
     * @return The resulting element.
```

```

*
* @throws Exception This method may throw exceptions. Throwing an
exception will cause the operation
*
* to fail and may trigger recovery.
*/
OUT join(IN1 first, IN2 second) throws Exception;
}

```

(3) Join with Flat-Join Function

Join 和 FlatJoin 之间的区别和 Map 与 FlatMap 的区别一样。FlatJoin 接收两个数据集中的元素各一个，并返回 0 个、1 个或多个元素。

```

Flat Join
public interface FlatJoinFunction<IN1, IN2, OUT> extends Function,
Serializable {

    /**
     * The join method, called once per joined pair of elements.
     *
     * @param first The element from first input.
     * @param second The element from second input.
     * @param out The collector used to return zero, one, or more
elements.
     *
     * @throws Exception This method may throw exceptions. Throwing an
exception will cause the operation
     *
     * to fail and may trigger recovery.
     */
    void join (IN1 first, IN2 second, Collector<OUT> out) throws
Exception;
}

```

(4) Join with Projection (Java/Python Only)

有时候 Join 的结果会包含用户不感兴趣的信息，因此可以通过投影来选取一部分数据作为最终的 Join 结果，用法如下：

```

DataSet<Tuple3<Integer, Byte, String>> input1 = // [...]
DataSet<Tuple2<Integer, Double>> input2 = // [...]
DataSet<Tuple4<Integer, String, Double, Byte>
    result =
    input1.join(input2)
// key definition on first DataSet using a field position k
ey
.where(0)
// key definition of second DataSet using a field position
key

```

```
.equalTo(0)
// select and reorder fields of matching tuples
.projectFirst(0,2).projectSecond(1).projectFirst(1);
```

(5) Join with DataSet Size Hint

有时候为了指导 Join 的优化策略，用户也可以手动的添加这些提示信息，如下所示：

```
DataSet<Tuple2<Integer, String>> input1 = // [...]
DataSet<Tuple2<Integer, String>> input2 = // [...]

DataSet<Tuple2<Tuple2<Integer, String>, Tuple2<Integer, S
tring>>>
    result1 =
        // hint that the second DataSet is very small
        input1.joinWithTiny(input2)
            .where(0)
            .equalTo(0);

DataSet<Tuple2<Tuple2<Integer, String>, Tuple2<Integer, S
tring>>>
    result2 =
        // hint that the second DataSet is very large
        input1.joinWithHuge(input2)
            .where(0)
            .equalTo(0);
```

(6) Join Algorithm Hints

用户除了可以显式的指定优化策略，还可以使用 Flink 提供的很多现成的 Join 优化策略。如下所示：

```
DataSet<SomeType> input1 = // [...]
DataSet<AnotherType> input2 = // [...]

DataSet<Tuple2<SomeType, AnotherType> result =
    input1.join(input2, JoinHint.BROADCAST_HASH_FIRST)
        .where("id").equalTo("key");
```

系统提供的优化方案有如下几种：

- **OPTIMIZER_CHOOSES**: 这种优化策略相当于用户放弃自己选择，交给系统自己去抉择如何优化。
- **BROADCAST_HASH_FIRST**: 将 Join 中的第一个数据集进行广播并建立 Hash 表，传递给第二个数据集。这种做法适合第一个数据集比较小的情况。
- **BROADCAST_HASH_SECOND**: 将 Join 中的第二个数据集进行广播并建立 Hash 表，传递给第一个数据集。这种做法适合第二个数据集比较小的情况。
- **REPARTITION_HASH_FIRST**: 系统打散所有的输入（除非输入已经被打散），并且为第一个数据集建立一个 hash 表。这种策略适合两个数据集都比较大，但第一个

数据集相对小一点点的情况。注意到这是默认情况下（例如无法估计数据集的大小并且没有先前存在的分片或者排序结果能够被重用）系统的优化策略。

- **REPARTITION_HASH_SECOND**: 系统打散所有的输入（除非输入已经被打散），并且为第二个数据集建立一个 **hash** 表。这种策略适合两个数据集都比较大，但第二个数据集相对小一点点的情况。
- **REPARTITION_SORT_MERGE**: 系统打散所有的输入（除非输入已经被打散），并且对每个输入都进行排序（除非已经是有序的）。这种策略适合其中一个或两个都已经排好序了的情况。

4.1.1.19 Outer Join

Outer Join 和 **(Inner) Join** 类似，不同之处是 **(Inner) Join** 只保留在两个数据集中都出现的 **key**，而 **Outer Join** 会完整的保留其中一个数据集 (**left, right**) 或两个都保留 (**full**)。Outer Join 一般分为四步：

```
dataSet1
    .leftOuterJoin(dataSet2)
    .where("zip")//选第一个数据集的 key.
    .equalTo("zip") //选第二个数据集的 key.
    .with(new JoinFunction()) //传入一个 Join/FlatJoin Function
```

注意到第四步 **with()** 在 **Inner Join** 中不是必须的，但是在 **Outer Join** 中是必须的，因为可能会出现左边 (**left join/full join**)、右边 (**right join/full join**) 为空的现象，当出现这种情况时，需用用户自定义处理逻辑来如何抉择遇到空的情况。

和 **Inner Join** 一样，**Outer Join** 的 **key** 选取策略也有如下四种：

- a key expression
- a key-selector function
- one or more field position keys (Tuple DataSet only).
- Case Class Fields

也有 **JoinFunction** 和 **FlatJoinFunction** 两种处理逻辑。

也有相应的优化策略。

4.1.1.20 Cross

Cross 操作是将两个数据集合并成一个数据集。它的操作是将两个数据集中的元素两两组合，形成笛卡尔积。输出的结果既可以根据用户自定义的函数 (**CrossFunction**) 来实现自定义的输出形式，也可以是默认的 **Tuple2<A,B>** 类型的输出 (**A, B** 为这两个输入数据集的类型)。

(1) CrossFunction

CrossFunction

```

public interface CrossFunction<IN1, IN2, OUT> extends Function,
Serializable {

    /**
     * Cross UDF method. Called once per pair of elements in the
     Cartesian product of the inputs.
     *
     * @param val1 Element from first input.
     * @param val2 Element from the second input.
     * @return The result element.
     *
     * @throws Exception The function may throw Exceptions, which will
     cause the program to cancel,
     *                    and may trigger the recovery logic.
     */
    OUT cross(IN1 val1, IN2 val2) throws Exception;
}

```

(2) Cross with Projection

用户可能只关心两个数据集中的一部分数据，因此可以在 Cross 的结果中选择投影。只截取用户关心的维度。如下所示：

```

DataSet<Tuple3<Integer, Byte, String>> input1 = // [...]
DataSet<Tuple2<Integer, Double>> input2 = // [...]
DataSet<Tuple4<Integer, Byte, Integer, Double>
    result =
        input1.cross(input2)
        // select and reorder fields of matching tu
ples
        .projectSecond(0).projectFirst(1,0).projec
tSecond(1);

```

(3) Cross with DataSet Size Hint

类似 Join 一样，用户也可以显示的指出相交的两个数据集的大小相关的信息。如 crossWithTiny() 函数。

4.1.1.21 CoGroup

CoGroup 提供分组处理数据集的能力。两个数据集根据同一个 key 进行分组，分到同一个组的元素可以使用自定义的 **CoGroupFunction** 来进行处理。如果针对某个特定的 key，其中一个数据集有分组元素，另一个为空，为空的分组则会传入一个空组。CoGroupFunction 能够在各自分组的元素上进行迭代，并且允许用户返回 0 个，1 个或多个元素。同样道理，key 的选取有多种不同方式。

CoGroupFunction

```
public interface CoGroupFunction<IN1, IN2, O> extends Function,
Serializable {

    /**
     * This method must be implemented to provide a user implementation
     of a
     * coGroup. It is called for each pair of element groups where the
     elements share the
     * same key.
     *
     * @param first The records from the first input.
     * @param second The records from the second.
     * @param out A collector to return elements.
     *
     * @throws Exception The function may throw Exceptions, which will
     cause the program to cancel,
     *
     * and may trigger the recovery logic.
     */
    public void coGroup(Iterable<IN1> first, Iterable<IN2> second,
Collector<O> out) throws Exception;
}
```

实例如下：

```
DataSet<Tuple2<String, Integer>> ival1 = // [...]
DataSet<Tuple2<String, Double>> dval1 = // [...]
DataSet<Double> output = ival1.coGroup(dval1)
    // group first DataSet on first tuple field
    .where(0)
    // group second DataSet on first tuple field
    .equalTo(0)
    // apply CoGroup function on each pair of groups
    .with(new MyCoGrouper());
```

4.1.1.22 Union

将多个数据集合并成一个数据集。示例如下：

```
DataSet<Tuple2<String, Integer>> vals1 = // [...]
DataSet<Tuple2<String, Integer>> vals2 = // [...]
DataSet<Tuple2<String, Integer>> vals3 = // [...]
DataSet<Tuple2<String, Integer>> unioned = vals1.union(vals2).union(vals3);
```

4.1.1.23 Rebalance

重新对数据集的分片进行均衡以消除数据倾斜。

```
DataSet<String> in = // [...]
// rebalance DataSet and apply a Map transformation.
DataSet<Tuple2<String, String>> out = in.rebalance()
                                   .map(new Mapper());
```

4.1.1.24 Hash-Partition

在给定的 key 上对一个数据集进行 hash 分片。key 的选取原则如前所述。

```
DataSet<Tuple2<String, Integer>> in = // [...]
// hash-partition DataSet by String value and apply a MapP
// partition transformation.
DataSet<Tuple2<String, String>>
out = in.partitionByHash(0)
      .mapPartition(new PartitionMapper());
```

4.1.1.25 Range-Partition

在给定的 key 上对数据集进行 Range-partition.

```
DataSet<Tuple2<String, Integer>> in = // [...]
// range-partition DataSet by String value and apply a Map
// Partition transformation.
DataSet<Tuple2<String, String>>
out = in.partitionByRange(0)
      .mapPartition(new PartitionMapper());
```

4.1.1.26 Sort Partition

根据给定的 key 和特定的排序规则，本地排序所有的数据集分片。

```
DataSet<Tuple2<String, Integer>> in = // [...]
// Locally sort partitions in ascending order on the secon
// d String field and
// in descending order on the first string field.
// Apply a MapPartition transformation on the sorted parti
// tions.
DataSet<Tuple2<String, String>> out =
  in.sortPartition(1, Order.ASCENDING)
    .sortPartition(0, Order.DESENDING)
    .mapPartition(new PartitionMapper());
```

4.1.1.27 First-n

返回数据集中前 N 个元素。

```
DataSet<Tuple2<String, Integer>> in = // [...]
// Return the first five (arbitrary) elements of the DataSet
out1 = in.first(5);

// Return the first two (arbitrary) elements of each string group
DataSet<Tuple2<String, Integer>> out2 = in.groupBy(0)
    .first(2);

// Return the first three elements of each string group ordered by the Integer field
DataSet<Tuple2<String, Integer>> out3 =
    in.groupBy(0)
    .sortGroup(1, Order.ASCENDING)
    .first(3);
```

4.3.2 Data Sources

Data Source 是用来创建初始的数据集。Flink 允许从一个文件或者一个 Java 集合中创建。创建一个数据集的通用的抽象操作为 **InputFormat**。Flink 提供了好几种内置的方法用来创建数据集，大多数方法都附加在 `ExecutionEnvironment` 对象上。例如 `readTextFile(path)`

4.3.3 Data Sinks

Data Sinks 操作用来消费数据集，将数据集存储或者打印出来。Data Sink 操作被 `OutputFormat` 类来描述。同样 Flink 提供了好多内置的方法，这些方法都附加在 `DataSet` 对象上。例如 `print()`

4.3.4 Iteration Operators

关于 Flink 的迭代模型：

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

迭代操作使得 Flink 的程序具有了循环处理的能力。迭代操作使得程序的一部分能够重复的执行，将其中的一次迭代结果输入到下次的迭代过程中。这里有两种迭代操作：**BulkIteration & DeltaIteration**。

(1) Bulk Iterations

在 Flink 中，使用 `DataSet` 的 `iterate(int)` 方法来创建一个 `BulkIteration`。这将会返回一个 `IterativeDataSet`，这个数据集可以像 `DataSet` 数据集一样使用常规的一些操作。该函数中传入的参数为最大迭代次数。使用 `DataSet` 的 `closeWith(DataSet<T>)` 方法来结束一个迭代操作。传入的参数为一个数据集，该数据会被用到下次的迭代计算中。返回的结果是此次迭代计算的最终结果。当然用户也可以自定义终止规则--使用 `closeWith(DataSet, DataSet)` 方法。该方法会评估第二个传入的数据集，如果这个数据集是空的，则终止迭代。

如采用如下的方式来估计 π 的值。

```
public static void getPi(ExecutionEnvironment env) throws Exception {
    DataSet<Integer> dataSet = env.fromElements(0); //the initial data
    set. Actually its used for counter.
    IterativeDataSet<Integer> iter = dataSet.iterate(size); // the
    initial iterative data set.
    DataSet<Integer> middleDataSet = iter.map((MapFunction<Integer,
    Integer>) (Integer value) -> {
        double x = Math.random();
        double y = Math.random();
        return value + ((x * x + y * y < 1) ? 1 : 0);
    }); //if the random point(x,y) fill in the unit circle, the counter
    increase by 1.
    DataSet<Integer> result = iter.closeWith(middleDataSet); // use the
    middle data set as the iterative middle result.
    result.map((MapFunction<Integer, Double>) value -> value * 4.0 /
    size).print(); // get the pi.
}
```

(2) Delta Iterations

4.3.5 Operating on data objects in functions

Flink 的运行态 (Runtime) 和用户定义的函数以 Java 对象 (Object) 的形式来交换数据。Functions 从 runtime 中接收对象 (input objects) 作为函数参数的输入，并且输出对象 (output objects) 最为函数的输出。因为这些对象需要同时被用户自定义的函数和 Runtime 所访问，所以理解和遵循用户代码访问 (读取和修改) 这些对象的规则是非常重要的。

用户自定的函数从 Flink 的 Runtime 中接收对象有两种方式，一种是最为常规的参数 (例如 `MapFunction` 接收一个一个的元素)，另一种是接收一个迭代器的参数 (例如 `GroupReduceFunction`) 来遍历这些对象。我们将这种由 Runtime 传给 user function 的对象成为 **input objects**，user function 也可以产生对象并返回给 Runtime，我们将这种由 user function 返回给 runtime 的对象称之为 **output objects**。

Flink 的 `DataSet` API 为 Flink 的 Runtime 的创建和使用 input objects 提供了两种模式。

4.4 ExecutionEnvironment

```
public JobExecutionResult execute(String jobName) throws Exception {
    PlanExecutor executor = getExecutor();

    Plan p = createProgramPlan(jobName);

    // Session management is disabled, revert this commit to enable
    //p.setJobId(jobID);
    //p.setSessionTimeout(sessionTimeout);

    JobExecutionResult result = executor.executePlan(p);

    this.lastJobExecutionResult = result;
    return result;
}
```

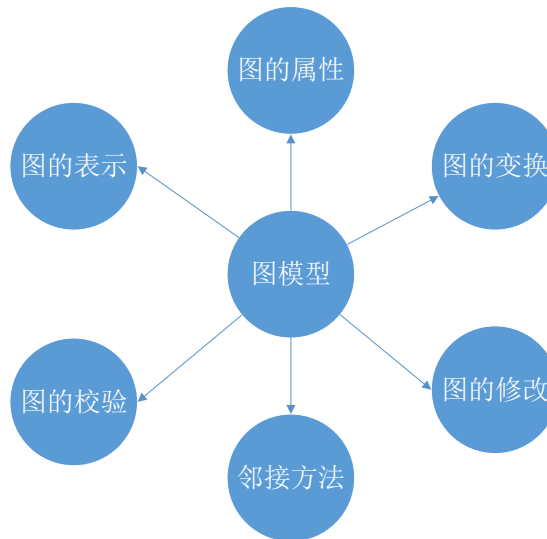
5. Libraries

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, Null
value.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.getExecutionEnvi
ronment();

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, VW>> 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

```

(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)

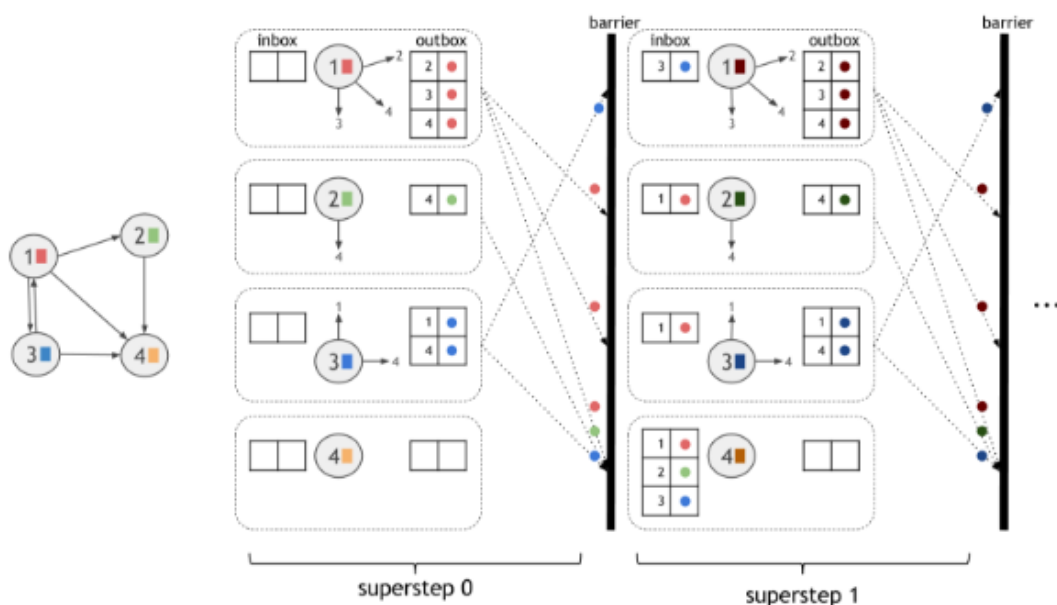
5.1.2 Iterative Graph Processing

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。一个顶点只要知道了其他顶点的 ID，它就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);
    }
}

```

5.1.2.2 Scatter-Gather Iterations

Scatter-Gather 模型，也称之为 Signal/Collect 模型，是以顶点的视角来进行计算的。计算过程是以同步迭代的方式一步一步进行的，这里的步称之为超步（Supersteps）。在每个超步中，一个顶点向其他的顶点发送数据，并且会根据它接收的数据来更新它自己的值。为了使用 Gelly 中的 Scatter-Gather 迭代模型，用户只需要定义顶点在超步中如何计算：

Scatter: 一个顶点产生消息并且将这个消息发送给其他的顶点；

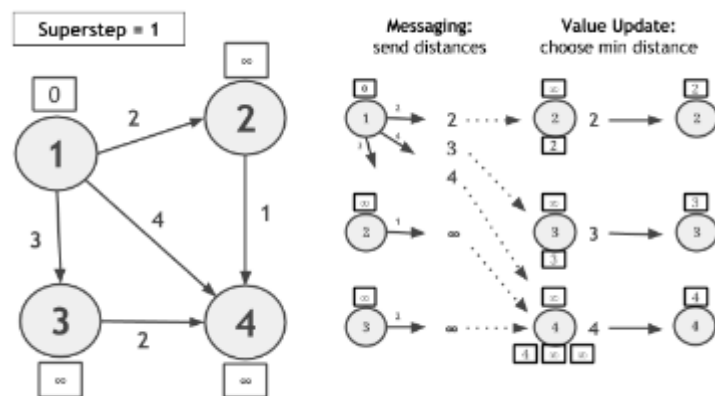
Gather: 一个顶点根据接收到的消息来更新自己的数据。

Gelly 为 Scatter-Gather 迭代模型提供了两个方法接口：**ScatterFunction** 和 **GatherFunction**。ScatterFunction 允许一个节点能够向其他节点发送消息，在同一个超步中，发送和接收消息

都能够完成；**GatherFunction** 定义了一个节点根据它接收到的消息来怎样更新数据。这两个函数再加上一个最大迭代次数共同构成了 Gelly 中 **runScatterGatherIteration** 方法所需要的参数。这个方法将会在一个图中执行 Scatter-Gather 迭代模型，并且输出一个更新了节点的值的新图。

Scatter-Gather 迭代模型能够接收一些附加信息，例如节点总数，入度和出度，邻节点的类型（in/out/all）等。默认情况下，根据入边来更新节点的值，根据出边来向其他节点发送消息。

让我们以单源点最短路径为例，来讲解如何使用 Scatter-Gather 迭代模型。如下图所示，假设源点为 1。在每个超步中，每个节点发送一个候选的距离消息给它的邻居节点。这个消息是节点的值加上边的值。根据所有接收到的消息，节点计算其中的最短距离，如果这个最短距离比它当前的值小，则更新当前的值为这个最短距离。如果在一个超步中，一个节点没有改变它本身的值，那么在下一个超步中，这个节点也就不会向它的邻居节点发送消息。当没有值被更新时，算法结束。



5.1.2.3 Gather-Sum-Apply Iterations

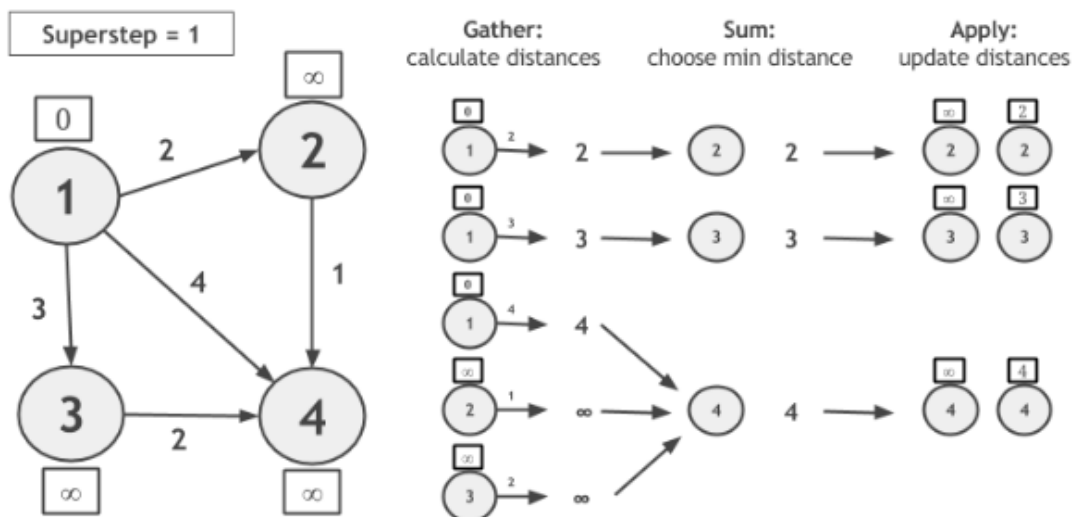
和 Scatter-Gather 模型一样，Gather-Sum-Apply 模型也是按照多个同步迭代的步运行的。这里的每步称为超步。

Gather: 一个用户定义的函数，一个顶点产生一个局布值，并且将这个值发送给它的邻接点；

Sum: 在 Gather 阶段产生的局布值被聚合成一个单一的值，用户可以自定义聚合函数；

Apply: 每个顶点根据在 Sum 阶段获得的聚合值以及当前的值来更新其值。

让我们以单源点最短路径为例，讲述如何运用 GSA 模型来进行计算。如下图所示，假设源点为 1，在 **Gather** 阶段，我们通过将每个节点当前的值和边上的权重相加的方式来计算候选距离；在 **Sum** 阶段，通过顶点 ID 来将 Gather 阶段产生的候选距离分组，并且每组中选择一个最小距离；在 **Apply** 阶段，新计算得到的距离和当前的值进行比较，选出最小的距离作为当前节点的值。



注意到，如果一个节点在一个 **superstep** 中其值没有改变，那么在下一个 **superstep** 中他不会去计算候选距离。这个算法在没有任何节点的值改变的时候结束。

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。 <pre>DataSet<Vertex<K, LongValue>> inDegree = graph .run(new VertexInDegree() .setIncludeZeroDegreeVertices(true));</pre> 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。 <code>graph.run(new TranslateGraphIds(new LongValueToStringValue()));</code>
	TranslateVertexValues	根据给定的转换函数，转换图中的顶点的值。 <code>graph.run(new TranslateVertexValues(new LongValueAddOffset(vertexCount)));</code>
	TranslateEdgeValues	根据给定的转换函数，转换图中的边的值。 <code>graph.run(new TranslateEdgeValues(new Nullify()));</code>

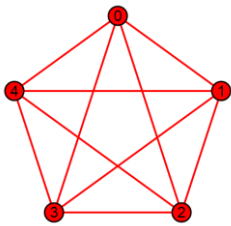
5.1.4 Graph Generators

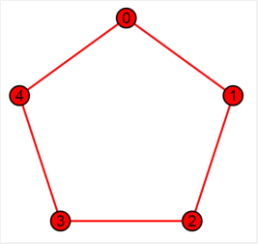

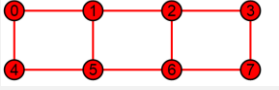
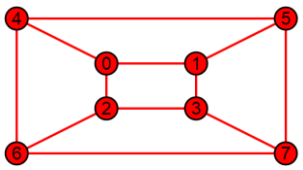

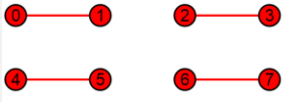
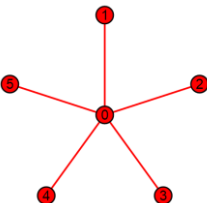
org.apache.flink.graph.generator 包中提供了很多图生成相关的 API。这些图生成算法有如下特点：

- 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。

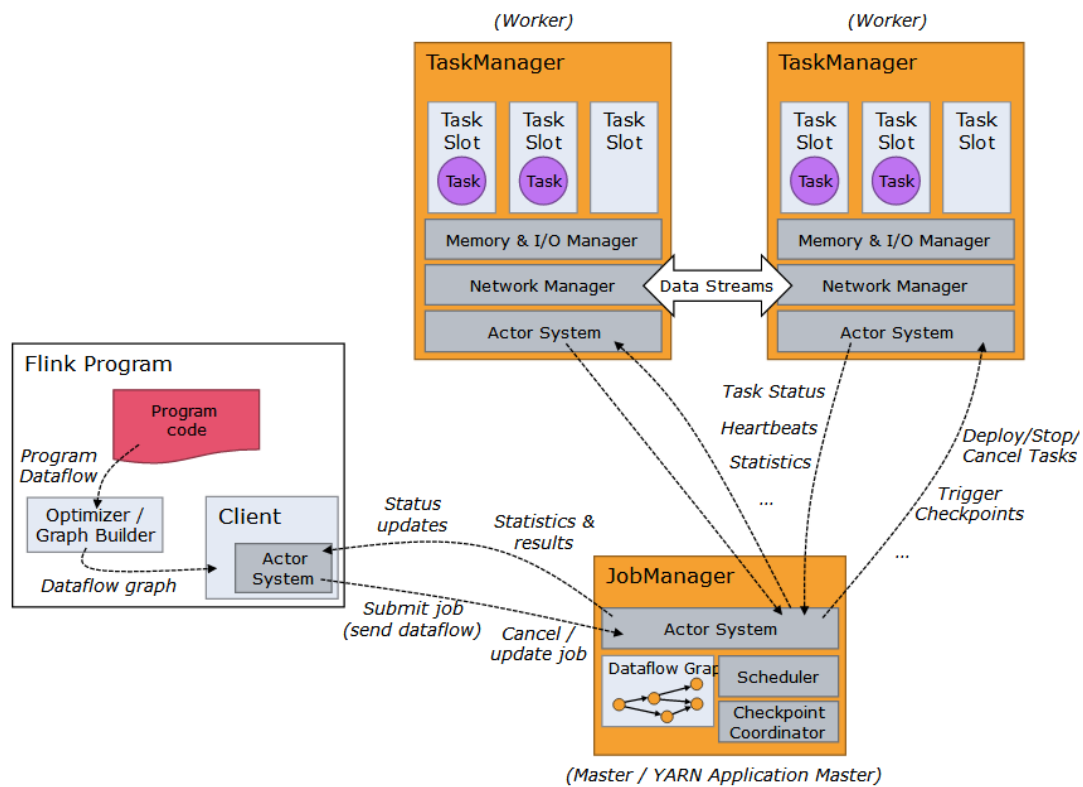
在调用时，首先应该设置参数，然后再调用 **generate()**方法。

Complete 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

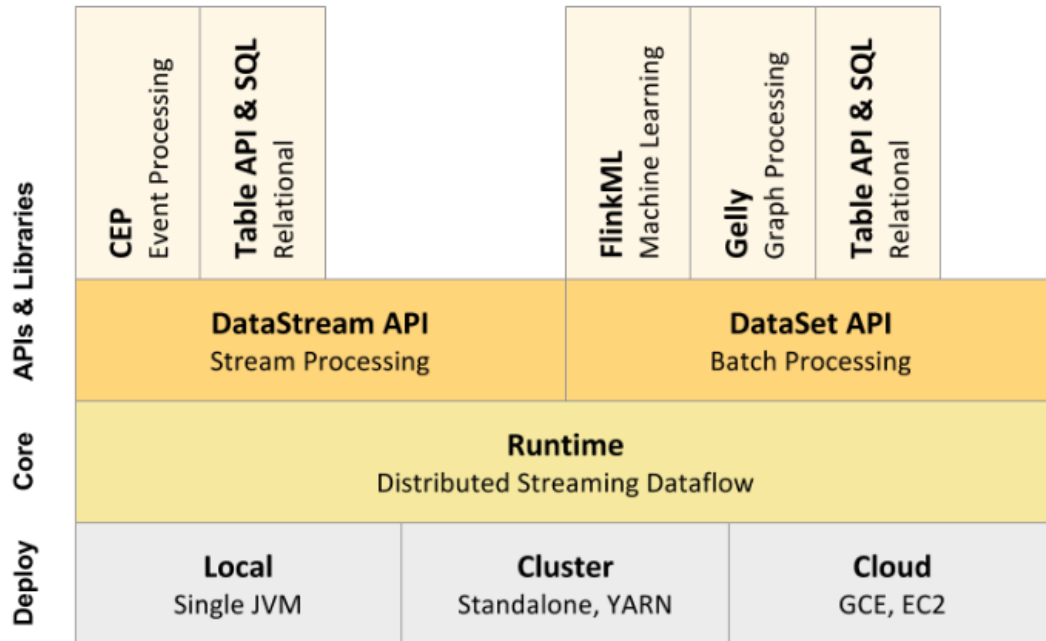


执行过程 (The Processes)

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。

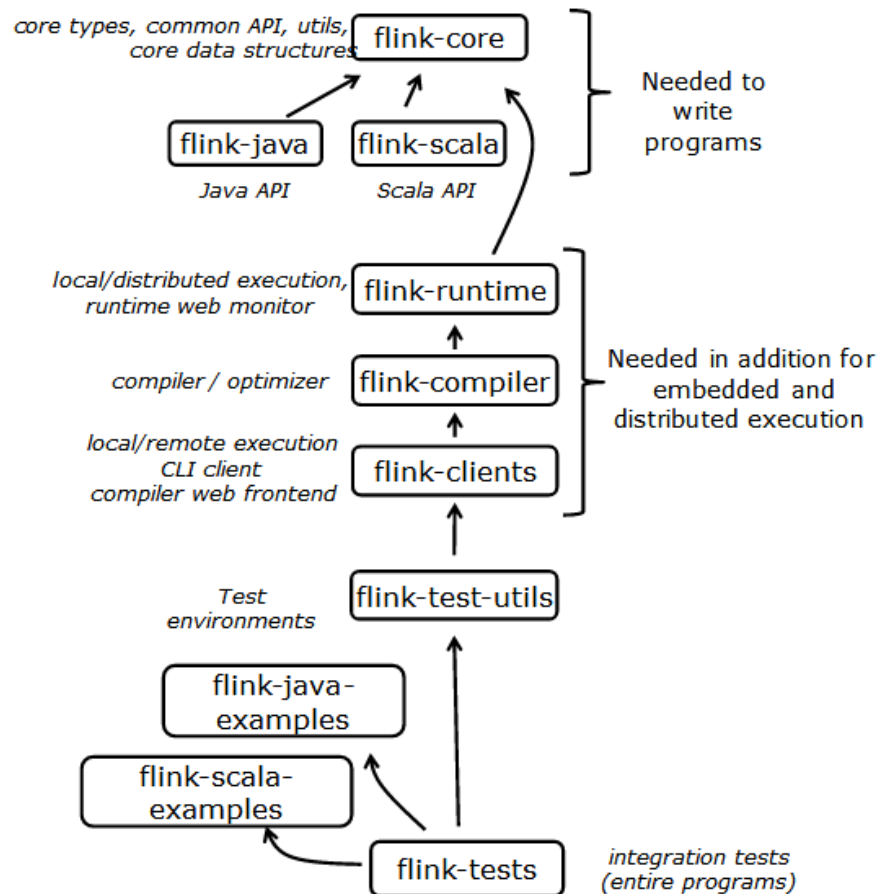
DataStream API & DataSet API: 他们通过各自的编译程序，都能生成 JobGraph。DataSet API 使用了一个优化器来决定程序的优化计划（Optimal Plan），而 DataStream API 使用了一个流建造器（Stream Builder）。

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：

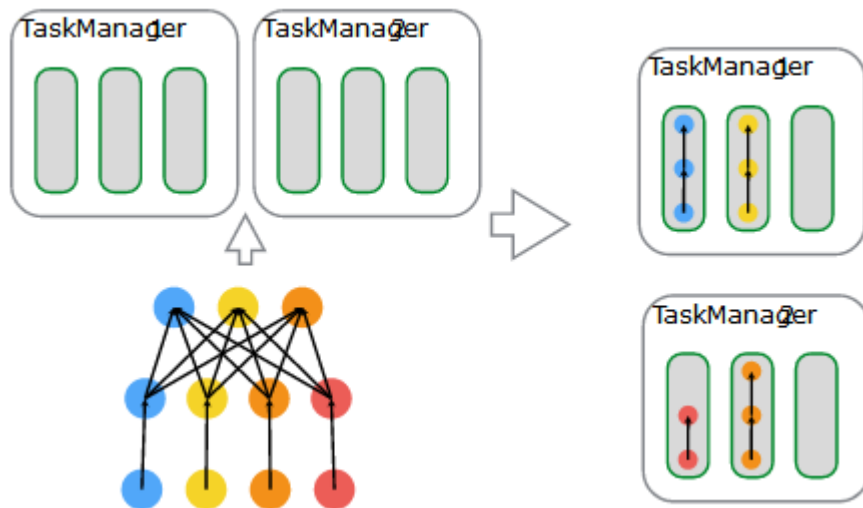


6.2 Jobs and Scheduling

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：



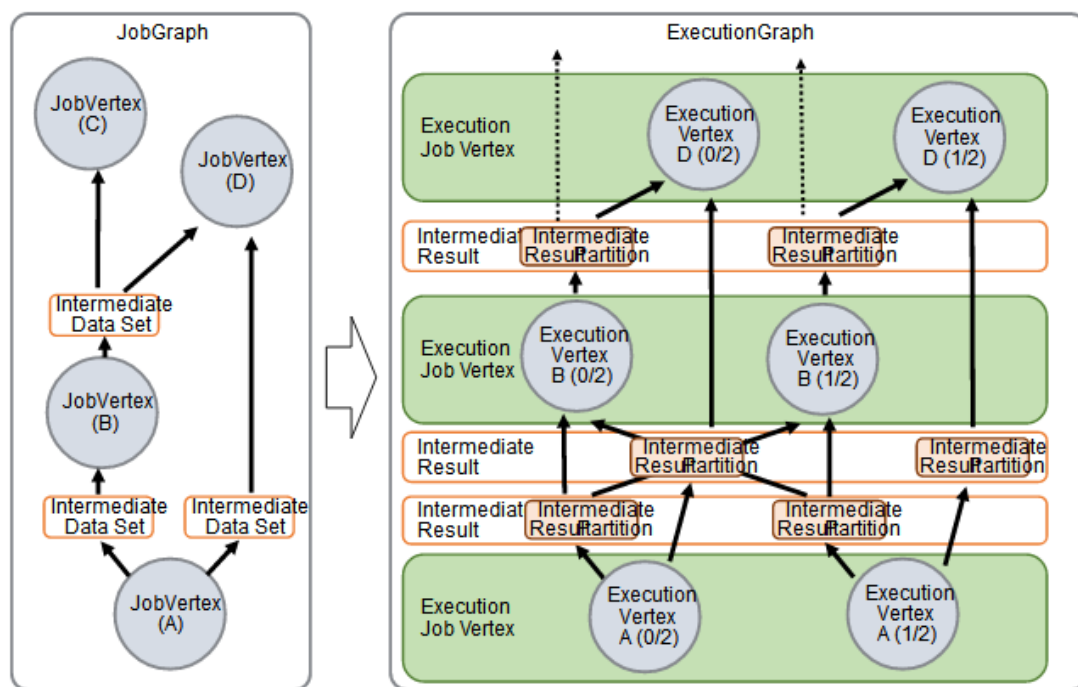
在内部实现时，Flink 通过定义 `SlotSharingGroup` 和 `CoLocationGroup` 这两个类来实现多个 task 共享一个 slot.（物理执行计划的生成，上面的任务是如何划分的）

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 ExecutionVertex，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，后者用来记录 IntermediateDataSet 每个部分的状态。如下图所示：

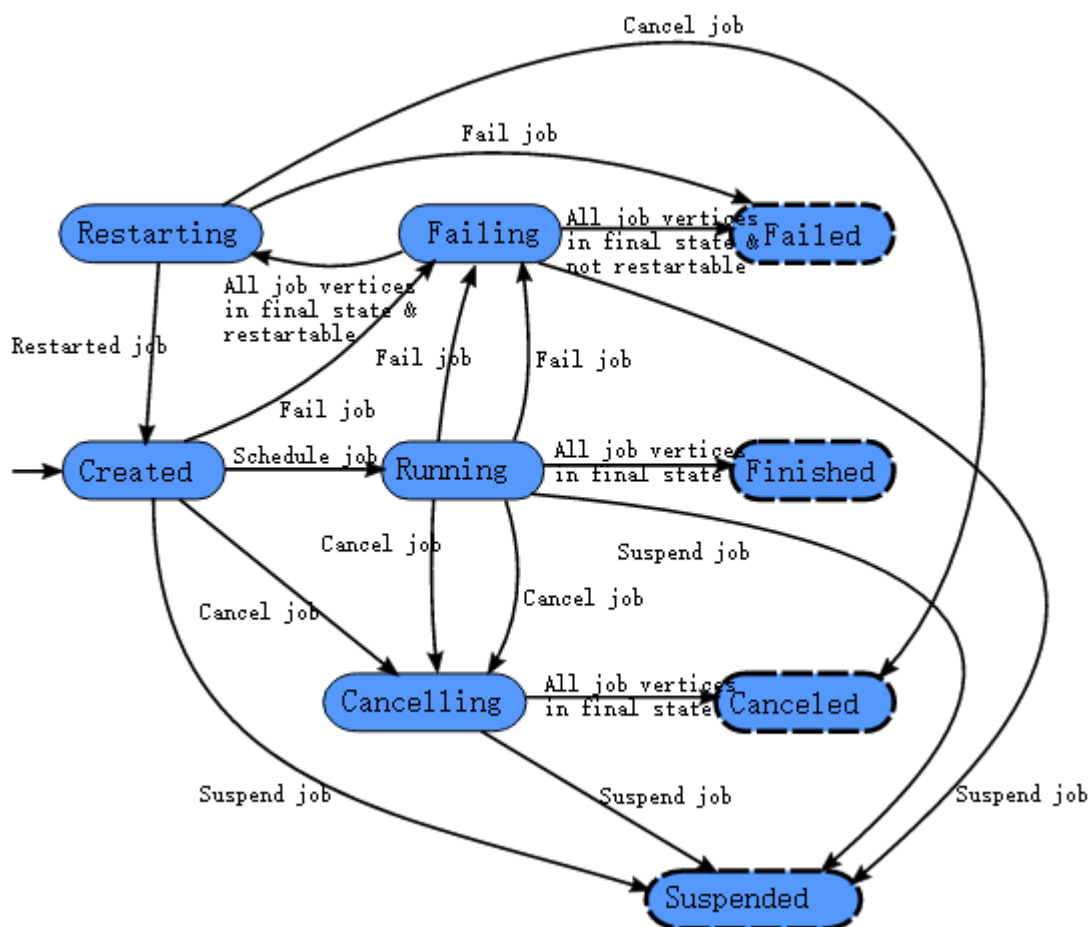


每个 **ExecutionGraph** 都附带的有一个（任务状态）**job status**。这个 **job status** 表明当前这个任务的执行状态。

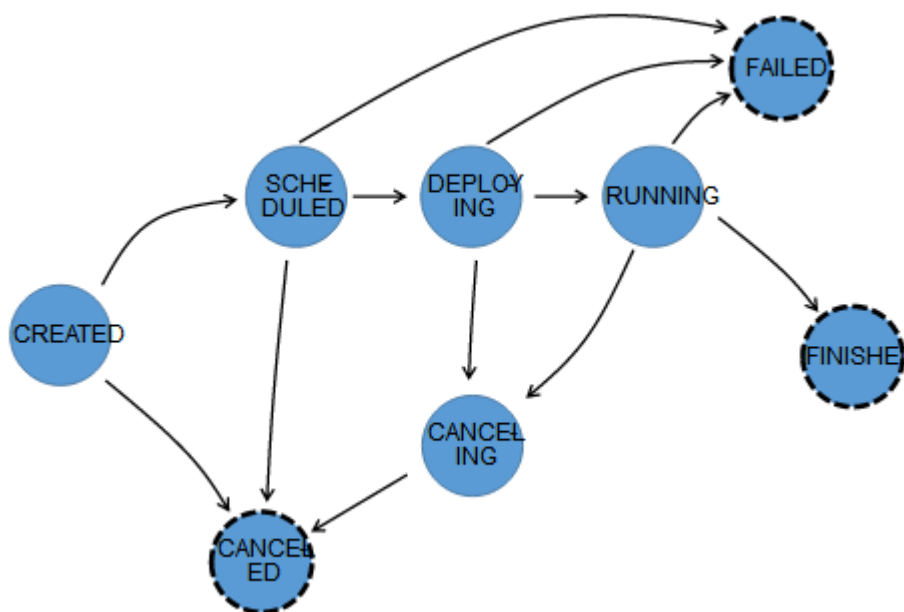
一个 Flink 的 **job**，首先进入的是已创建状态（**created**），执行完所有的任务之后就会转到已完成状态（**finished**）。如果遇到任务执行失败的情况，**job** 首先会转到失败中的状态（**failing**），并且取消所有正在执行的任务。如果所有的顶点（**job vertex**）都到达了最终状态而且这个 **job** 不能够重启，则这个 **job** 将会转到失败状态（**failed**）。如果这个 **job** 能够被重启，它将会进入到重启状态（**restarting**），一旦 **job** 完全被重启了，他又进入了已创建的状态（**created**）。

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。

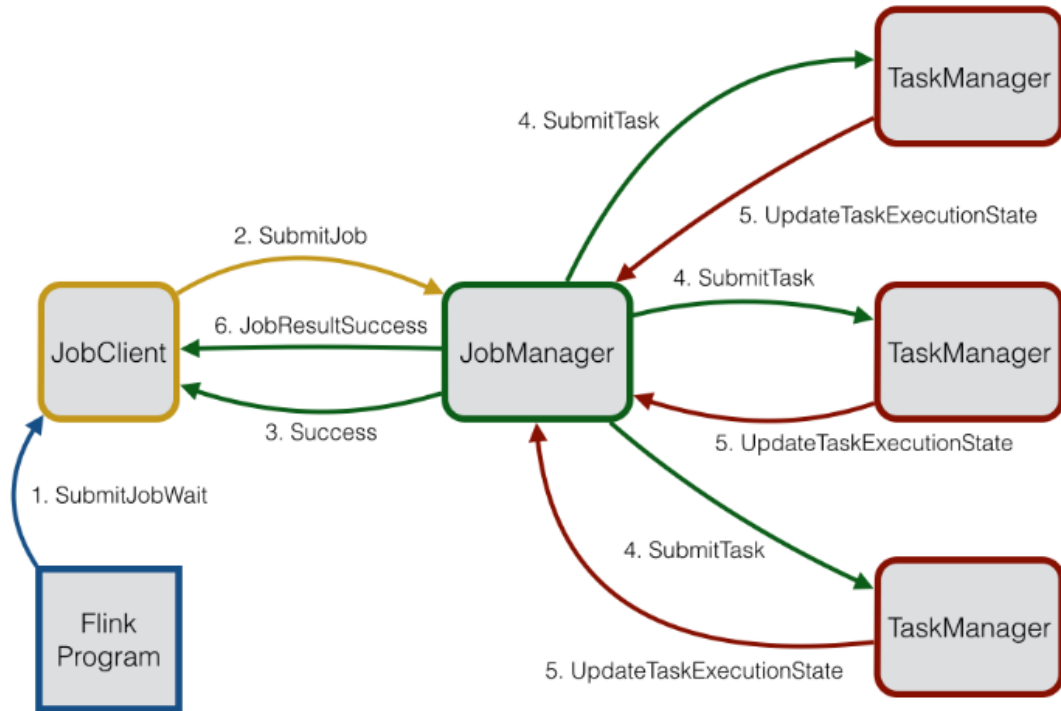


6.3 Akka & Actor System

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;

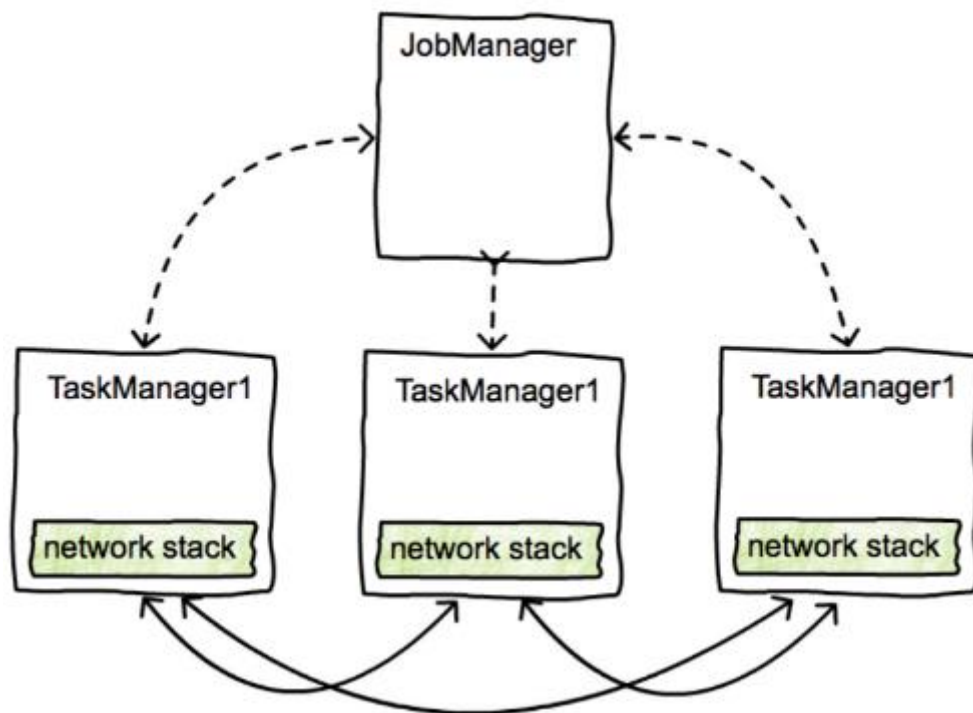
2. 数据交换的数据流（在网络上最终传输的数据）被抽象成一个 `IntermediateResult` 概念，他是可插拔的。这意味着系统基于相同的实现逻辑，既可以支持流数据的传输，又可以支持批数据的传输。

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。

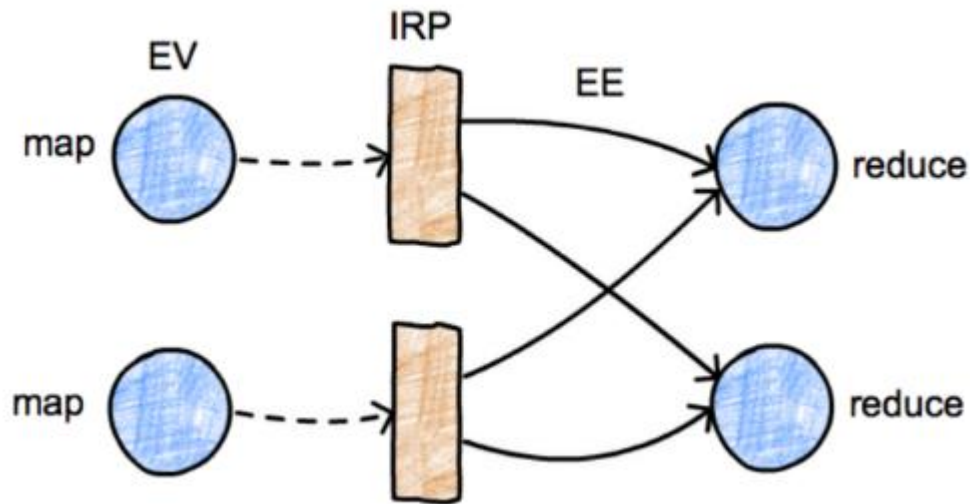
TaskManager: 工作节点，一个 `TaskManager` 通过多线程的方式来并发执行多个任务。每个 `TaskManager` 都包含一个 `CommunicationManager` 和一个 `MemoryManager`（即同个 `TaskManager` 中的多个任务共享这两个组件）。`TaskManager` 之间的数据交换是通过 TCP 连接来实现的。

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。

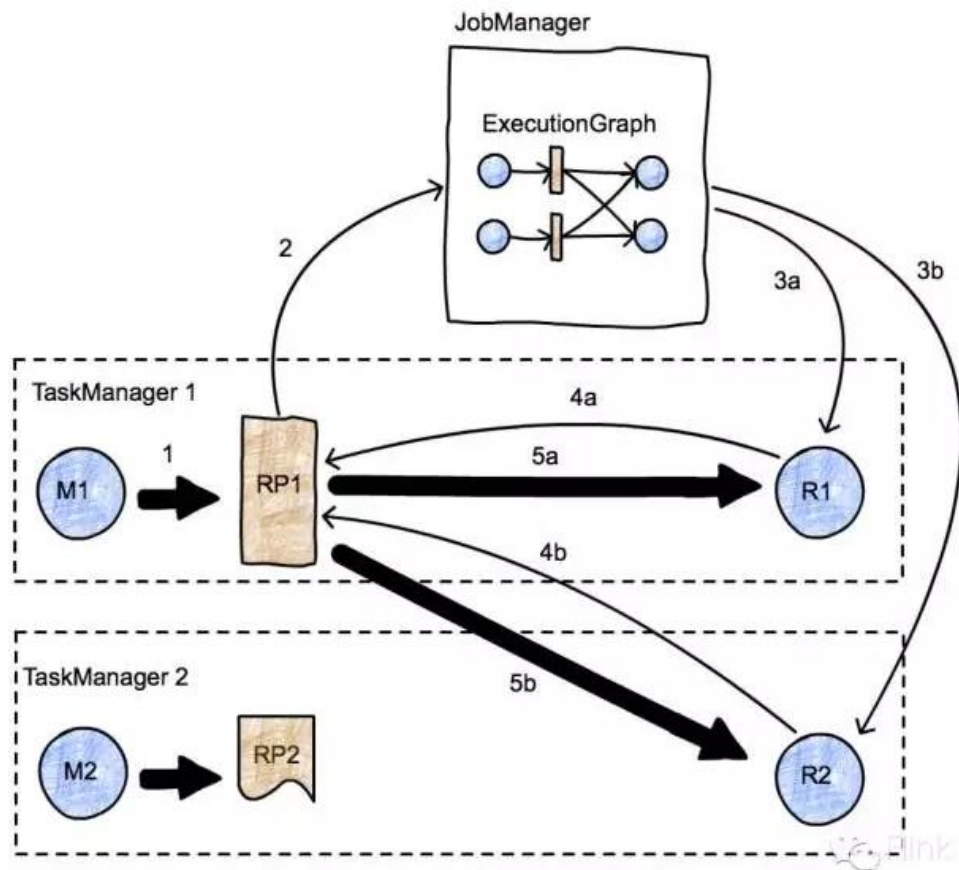
InputGate: 在接收端，逻辑上等价于 RP。它用于处理并收集来自上游的 buffer 中的数据。

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 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

8. Source Analysis

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

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` 中的数据集中时, 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。

```

Caused by: java.lang.reflect.InvocationTargetException
    at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
    at sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java:62)
    at sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.java:43)
    at java.lang.reflect.Method.invoke(Method.java:497)
    at org.apache.flink.graph.utils.proxy.Delegate$2.invoke(Delegate.java:94)
    at org.apache.flink.api.java.operators.JoinOperator$EquiJoin_$$jvstf11_1.print(JoinOperator$EquiJoin_$$jvstf11_1.java)
    at com.duansky.learn.flink.gelly.lib._TriangleListing.testDirectedTriangleListing(_TriangleListing.java:26)
    at com.duansky.learn.flink.gelly.lib._TriangleListing.main(_TriangleListing.java:17)
    at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
    at sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java:62)
    at sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.java:43)
    at java.lang.reflect.Method.invoke(Method.java:497)
    at org.apache.flink.client.program.PackagedProgram.callMainMethod(PackagedProgram.java:509)
    ... 35 more
Caused by: org.apache.flink.client.program.OptimizerPlanEnvironment$ProgramAbortException
    at org.apache.flink.client.program.OptimizerPlanEnvironment.execute(OptimizerPlanEnvironment.java:51)
    at org.apache.flink.api.java.ExecutionEnvironment.execute(ExecutionEnvironment.java:896)
    at org.apache.flink.api.java.DataSet.collect(DataSet.java:410)
    at org.apache.flink.api.java.DataSet.print(DataSet.java:1605)
    ... 48 more

```

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？

```

Caused by: java.lang.ClassCastException: java.lang.Integer cannot be cast to org.apache.flink.types.CopyableValue
    at org.apache.flink.graph.library.clustering.directed.TriangleListing$GenerateTriplets.reduce(TriangleListing.java:330)
    at org.apache.flink.runtime.operators.GroupReduceDriver.run(GroupReduceDriver.java:131)
    at org.apache.flink.runtime.operators.BatchTask.run(BatchTask.java:486)
    at org.apache.flink.runtime.operators.BatchTask.invoke(BatchTask.java:351)
    at org.apache.flink.runtime.taskmanager.Task.run(Task.java:585)
    at java.lang.Thread.run(Thread.java:745)

```

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

参考文献：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。

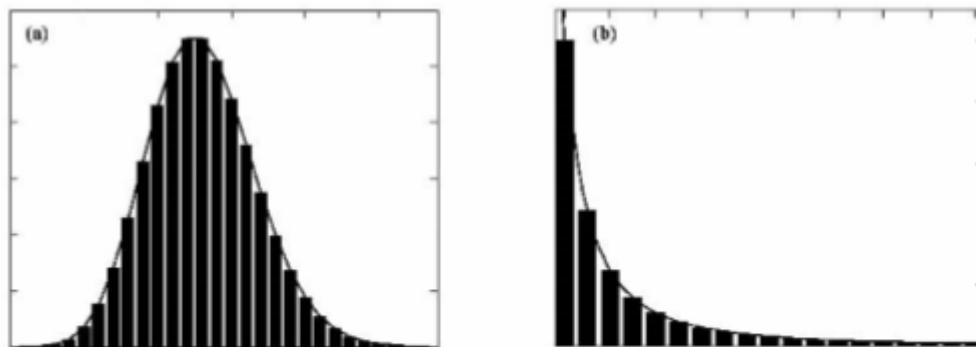


图1 泊松分布(a)与“长尾”分布(b)

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: $P(r) \sim r^{-\alpha}$ 。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 $P(X \geq x) \sim x^{-\alpha}$

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

$$y = c x^{-r}$$

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。