Apache Flink是一个面向分布式数据流处理和批量数据处理的开源计算平台，它能够基于同一个Flink运行时，提供支持流处理和批处理两种类型应用的功能。

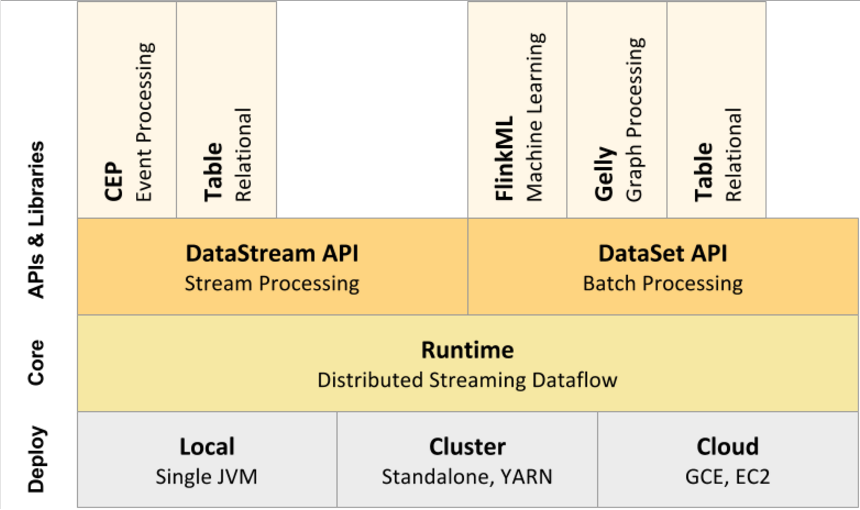
现有的开源计算方案，会把流处理和批处理作为两种不同的应用类型，因为它们所提供的SLA（Service-Level-Aggreement）是完全不相同的：流处理一般需要支持低延迟、Exactly-once保证，而批处理需要支持高吞吐、高效处理。

Flink从另一个视角看待流处理和批处理，将二者统一起来：Flink是完全支持流处理，也就是说作为流处理看待时输入数据流是无界的；**批处理被作为一种特殊的流处理，只是它的输入数据流被定义为有界的**。

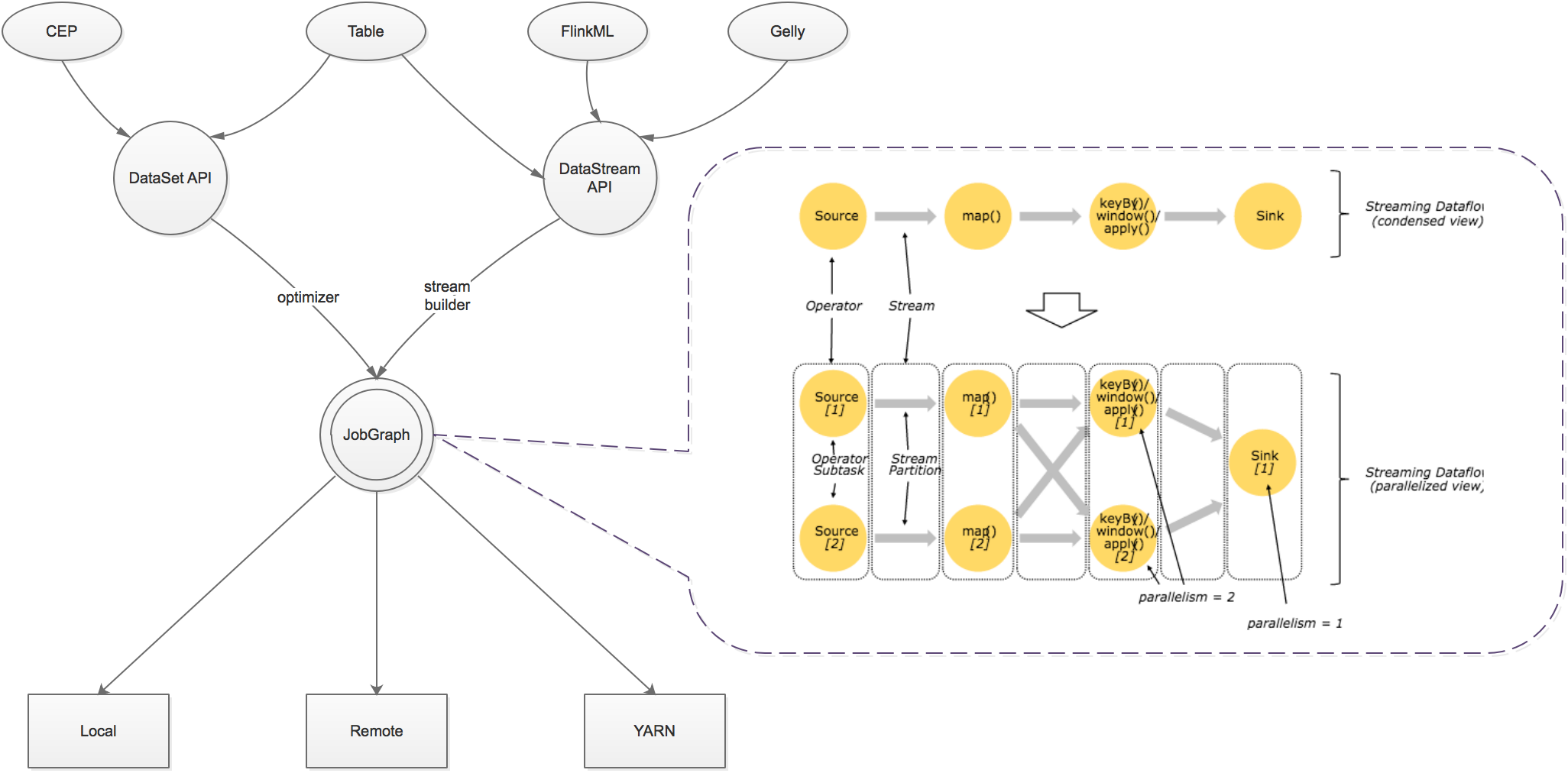
Flink流处理特性：

1. 支持高吞吐、低延迟、高性能的流处理
2. 支持带有事件时间的窗口（Window）操作
3. 支持有状态计算的Exactly-once语义
4. 支持高度灵活的窗口（Window）操作，支持基于time、count、session，以及data-driven的窗口操作
5. 支持具有Backpressure功能的持续流模型
6. 支持基于轻量级分布式快照（Snapshot）实现的容错
7. 一个运行时同时支持Batch on Streaming处理和Streaming处理
8. Flink在JVM内部实现了自己的内存管理
9. 支持迭代计算
10. 支持程序自动优化：避免特定情况下Shuffle、排序等昂贵操作，中间结果有必要进行缓存

## 一、架构

Flink以层级式系统形式组件其软件栈，不同层的栈建立在其下层基础上，并且各层接受程序不同层的抽象形式。  


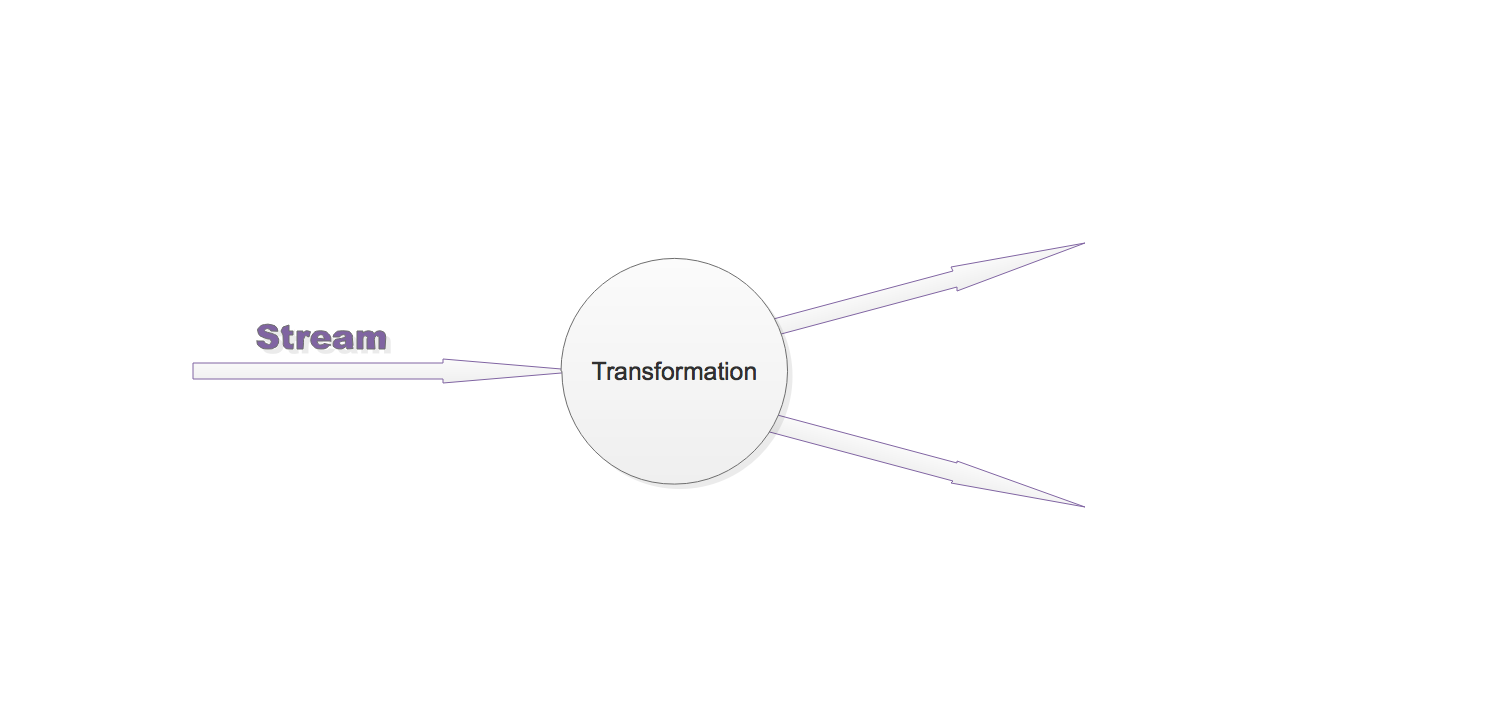
1. 运行时层以JobGraph形式接收程序。JobGraph即为一个一般化的并行数据流图（data flow），它拥有任意数量的Task来接收和产生data stream。
2. DataStream API和DataSet API都会使用单独编译的处理方式生成JobGraph。DataSet API使用optimizer来决定针对程序的优化方法，而DataStream API则使用stream builder来完成该任务。
3. 在执行JobGraph时，Flink提供了多种候选部署方案（如local，remote，YARN等）。
4. Flink附随了一些产生DataSet或DataStream API程序的的类库和API：处理逻辑表查询的Table，机器学习的FlinkML，图像处理的Gelly，复杂事件处理的CEP。



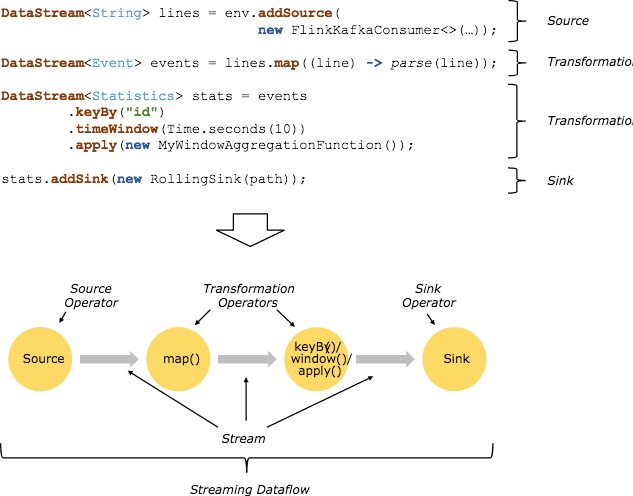
## 二、原理

### 1. [流、转换、操作符](https://ci.apache.org/projects/flink/flink-docs-release-1.2/concepts/programming-model.html#next-steps)

Flink程序是由Stream和Transformation这两个基本构建块组成，其中Stream是一个中间结果数据，而Transformation是一个操作，它对一个或多个输入Stream进行计算处理，输出一个或多个结果Stream。

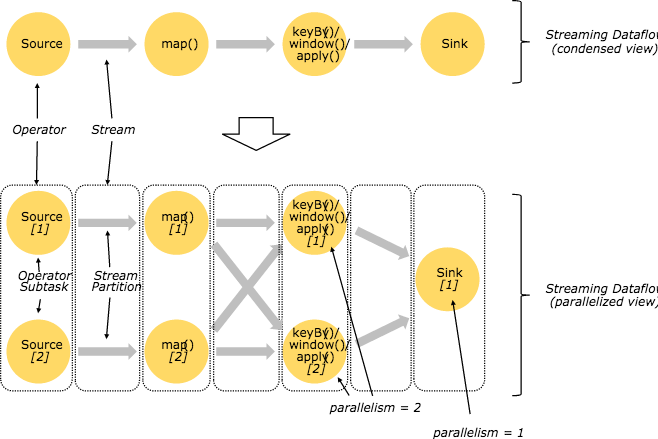


Flink程序被执行的时候，它会被映射为Streaming Dataflow。一个Streaming Dataflow是由一组Stream和Transformation Operator组成，它类似于一个DAG图，在启动的时候从一个或多个Source Operator开始，结束于一个或多个Sink Operator。



### 2. 并行数据流

一个Stream可以被分成多个Stream分区（Stream Partitions），一个Operator可以被分成多个Operator Subtask，每一个Operator Subtask是在不同的线程中独立执行的。一个Operator的并行度，等于Operator Subtask的个数，一个Stream的并行度总是等于生成它的Operator的并行度。

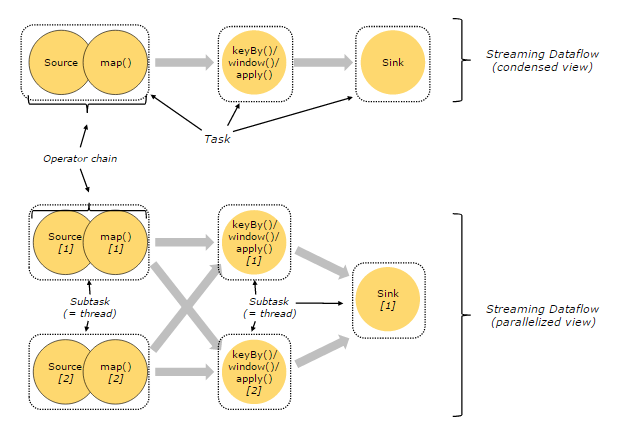


**One-to-one模式**  
比如从Source[1]到map()[1]，它保持了Source的分区特性（Partitioning）和分区内元素处理的有序性，也就是说map()[1]的Subtask看到数据流中记录的顺序，与Source[1]中看到的记录顺序是一致的。

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

### 3. 任务、操作符链

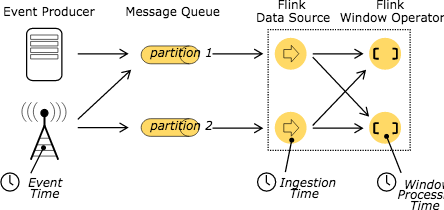
Flink分布式执行环境中，会将多个Operator Subtask串起来组成一个Operator Chain，实际上就是一个执行链，每个执行链会在TaskManager上一个独立的线程中执行。



### 4. [时间](https://ci.apache.org/projects/flink/flink-docs-release-1.2/dev/event_time.html)

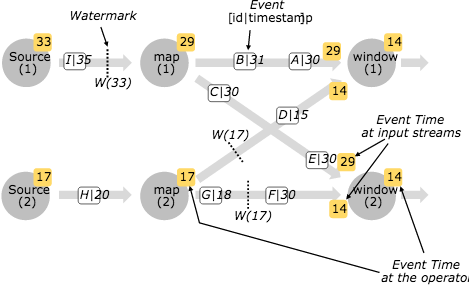
处理Stream中的记录时，记录中通常会包含各种典型的时间字段：

1. Event Time：表示事件创建时间
2. Ingestion Time：表示事件进入到Flink Dataflow的时间
3. Processing Time：表示某个Operator对事件进行处理的本地系统时间



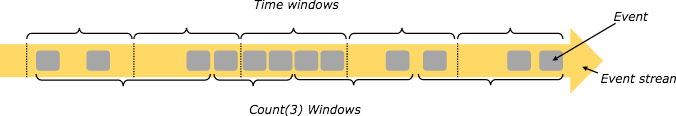
Flink使用WaterMark衡量时间的时间，WaterMark携带时间戳t，并被插入到stream中。

1. WaterMark的含义是所有时间t'< t的事件都已经发生。
2. 针对乱序的的流，WaterMark至关重要，这样可以允许一些事件到达延迟，而不至于过于影响window窗口的计算。
3. 并行数据流中，当Operator有多个输入流时，Operator的event time以最小流event time为准。



### 5. [窗口](https://flink.apache.org/news/2015/12/04/Introducing-windows.html)

Flink支持基于时间窗口操作，也支持基于数据的窗口操作：



窗口分类：

1. 按分割标准划分：timeWindow、countWindow
2. 按窗口行为划分：Tumbling Window、Sliding Window、自定义窗口

#### Tumbling/Sliding Time Window

// Stream of (sensorId, carCnt)

val vehicleCnts: DataStream[(Int, Int)] = ...

val tumblingCnts: DataStream[(Int, Int)] = vehicleCnts

// key stream by sensorId

.keyBy(0)

// tumbling time window of 1 minute length

.timeWindow(Time.minutes(1))

// compute sum over carCnt

.sum(1)

val slidingCnts: DataStream[(Int, Int)] = vehicleCnts

.keyBy(0)

// sliding time window of 1 minute length and 30 secs trigger interval

.timeWindow(Time.minutes(1), Time.seconds(30))

.sum(1)

#### Tumbling/Sliding Count Window

// Stream of (sensorId, carCnt)

val vehicleCnts: DataStream[(Int, Int)] = ...

val tumblingCnts: DataStream[(Int, Int)] = vehicleCnts

// key stream by sensorId

.keyBy(0)

// tumbling count window of 100 elements size

.countWindow(100)

// compute the carCnt sum

.sum(1)

val slidingCnts: DataStream[(Int, Int)] = vehicleCnts

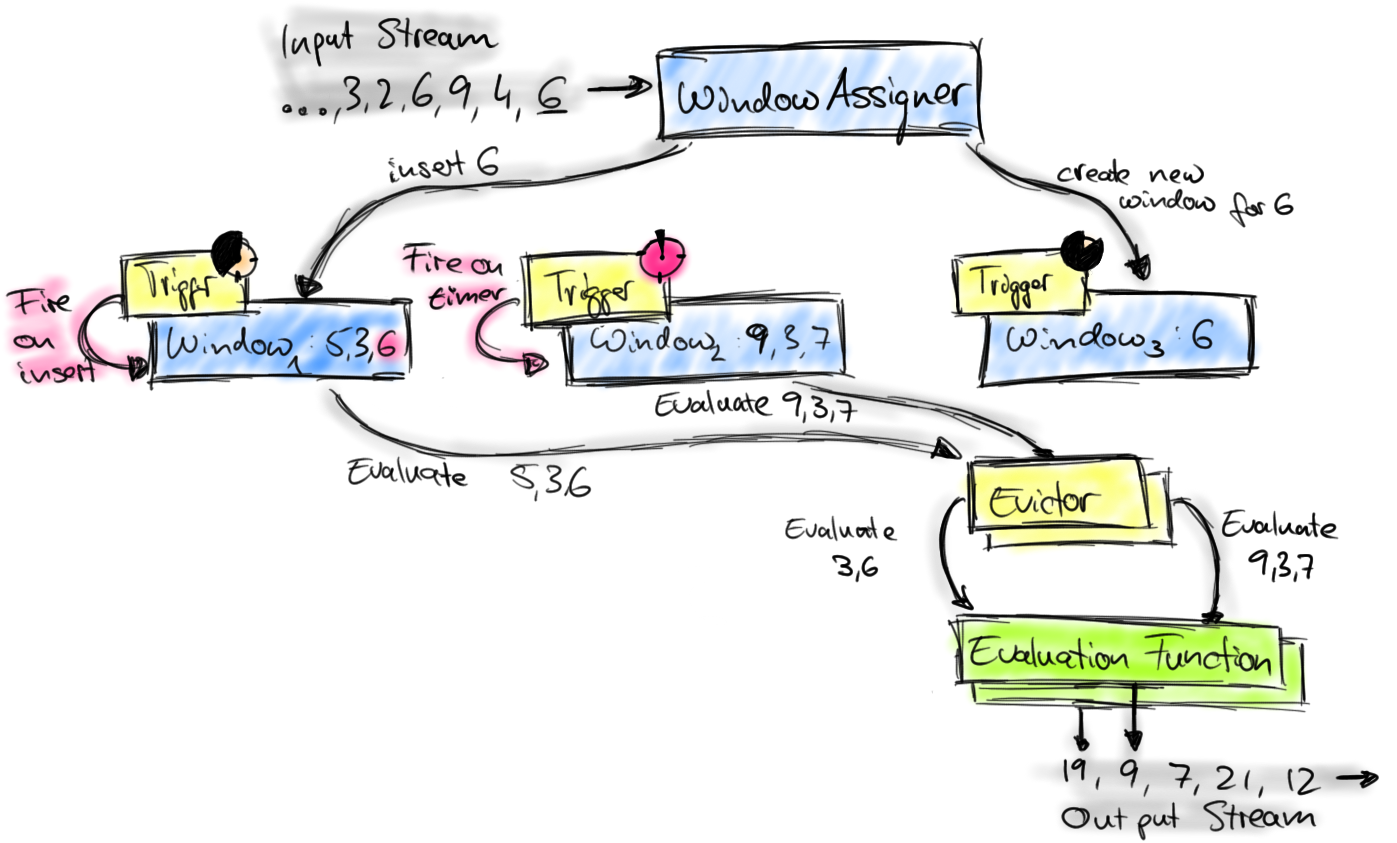
.keyBy(0)

// sliding count window of 100 elements size and 10 elements trigger interval

.countWindow(100, 10)

.sum(1)

#### 自定义窗口

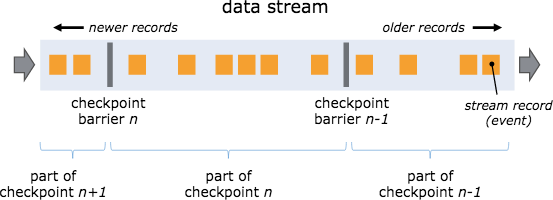


基本操作：

1. window：创建自定义窗口
2. trigger：自定义触发器
3. evictor：自定义evictor
4. apply：自定义window function

### 6. [容错](https://ci.apache.org/projects/flink/flink-docs-release-1.2/internals/stream_checkpointing.html)

Barrier机制：

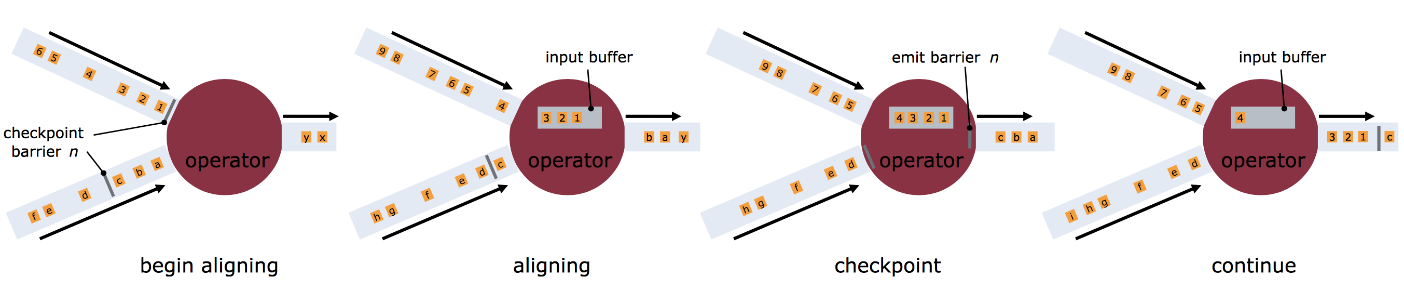


1. 出现一个Barrier，在该Barrier之前出现的记录都属于该Barrier对应的Snapshot，在该Barrier之后出现的记录属于下一个Snapshot。
2. 来自不同Snapshot多个Barrier可能同时出现在数据流中，也就是说同一个时刻可能并发生成多个Snapshot。
3. 当一个中间（Intermediate）Operator接收到一个Barrier后，它会发送Barrier到属于该Barrier的Snapshot的数据流中，等到Sink Operator接收到该Barrier后会向Checkpoint Coordinator确认该Snapshot，直到所有的Sink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

对齐：

当Operator接收到多个输入的数据流时，需要在Snapshot Barrier中对数据流进行排列对齐：

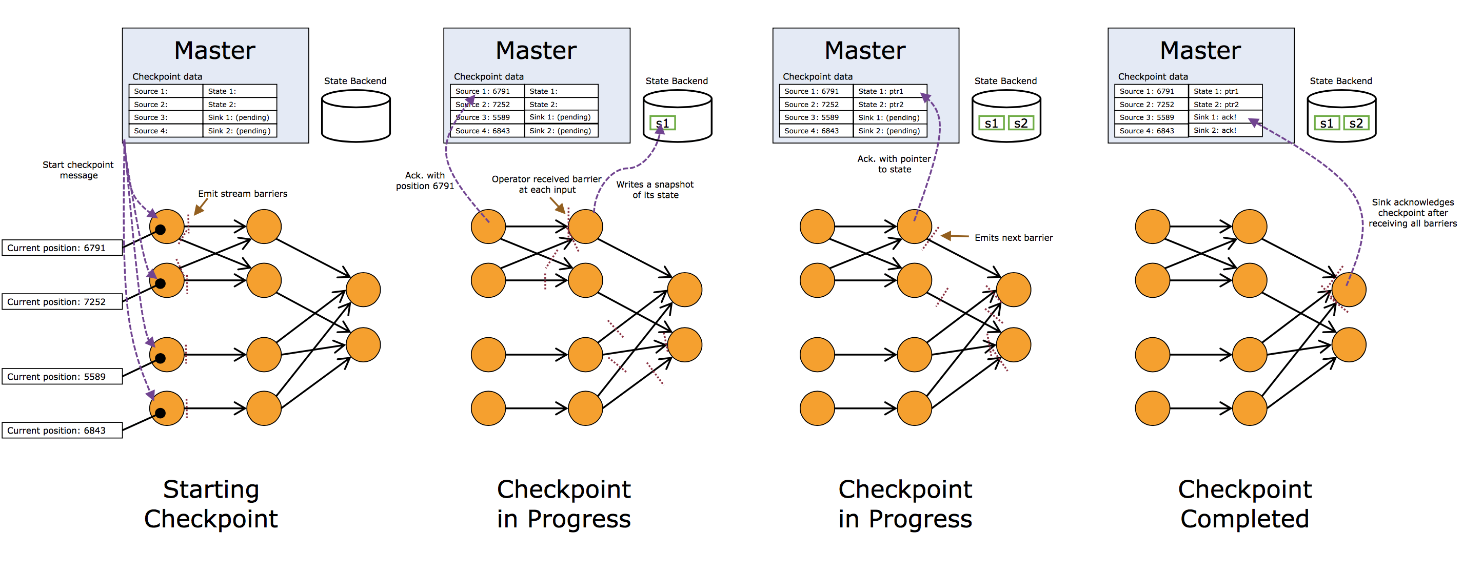
1. Operator从一个incoming Stream接收到Snapshot Barrier n，然后暂停处理，直到其它的incoming Stream的Barrier n（否则属于2个Snapshot的记录就混在一起了）到达该Operator
2. 接收到Barrier n的Stream被临时搁置，来自这些Stream的记录不会被处理，而是被放在一个Buffer中。
3. 一旦最后一个Stream接收到Barrier n，Operator会emit所有暂存在Buffer中的记录，然后向Checkpoint Coordinator发送Snapshot n。
4. 继续处理来自多个Stream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



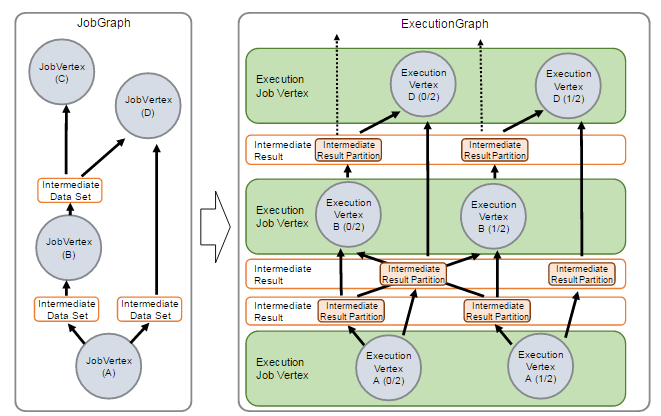
基于Stream Aligning操作能够实现Exactly Once语义，但是也会给流处理应用带来延迟，因为为了排列对齐Barrier，会暂时缓存一部分Stream的记录到Buffer中，尤其是在数据流并行度很高的场景下可能更加明显，通常以最迟对齐Barrier的一个Stream为处理Buffer中缓存记录的时刻点。在Flink中，提供了一个开关，选择是否使用Stream Aligning，如果关掉则Exactly Once会变成At least once。

CheckPoint：  
Snapshot并不仅仅是对数据流做了一个状态的Checkpoint，它也包含了一个Operator内部所持有的状态，这样才能够在保证在流处理系统失败时能够正确地恢复数据流处理。状态包含两种：

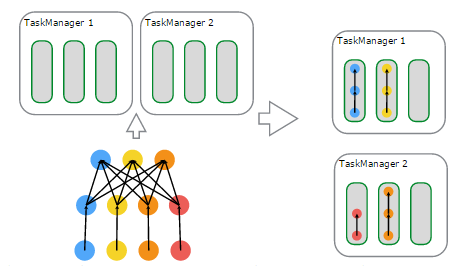
1. 系统状态：一个Operator进行计算处理的时候需要对数据进行缓冲，所以数据缓冲区的状态是与Operator相关联的。以窗口操作的缓冲区为例，Flink系统会收集或聚合记录数据并放到缓冲区中，直到该缓冲区中的数据被处理完成。
2. 一种是用户自定义状态（状态可以通过转换函数进行创建和修改），它可以是函数中的Java对象这样的简单变量，也可以是与函数相关的Key/Value状态。



### 7. 调度

在JobManager端，会接收到Client提交的JobGraph形式的Flink Job，JobManager会将一个JobGraph转换映射为一个ExecutionGraph，ExecutionGraph是JobGraph的并行表示，也就是实际JobManager调度一个Job在TaskManager上运行的逻辑视图。  


物理上进行调度，基于资源的分配与使用的一个例子：



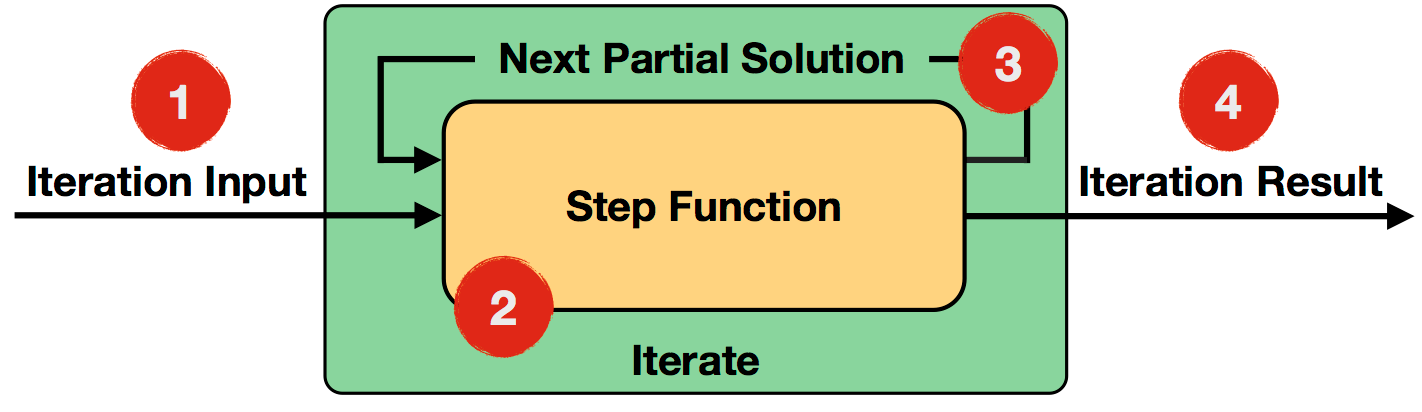
1. 左上子图：有2个TaskManager，每个TaskManager有3个Task Slot
2. 左下子图：一个Flink Job，逻辑上包含了1个data source、1个MapFunction、1个ReduceFunction，对应一个JobGraph
3. 左下子图：用户提交的Flink Job对各个Operator进行的配置——data source的并行度设置为4，MapFunction的并行度也为4，ReduceFunction的并行度为3，在JobManager端对应于ExecutionGraph
4. 右上子图：TaskManager 1上，有2个并行的ExecutionVertex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

### 8. [迭代](https://ci.apache.org/projects/flink/flink-docs-release-1.2/dev/batch/iterations.html)

[**机器学习**](http://lib.csdn.net/base/machinelearning)和图计算应用，都会使用到迭代计算，Flink通过在迭代Operator中定义Step函数来实现迭代[**算法**](http://lib.csdn.net/base/datastructure)，这种迭代算法包括Iterate和Delta Iterate两种类型。

#### Iterate

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



流程伪代码：

IterationState state = getInitialState();

while (!terminationCriterion()) {

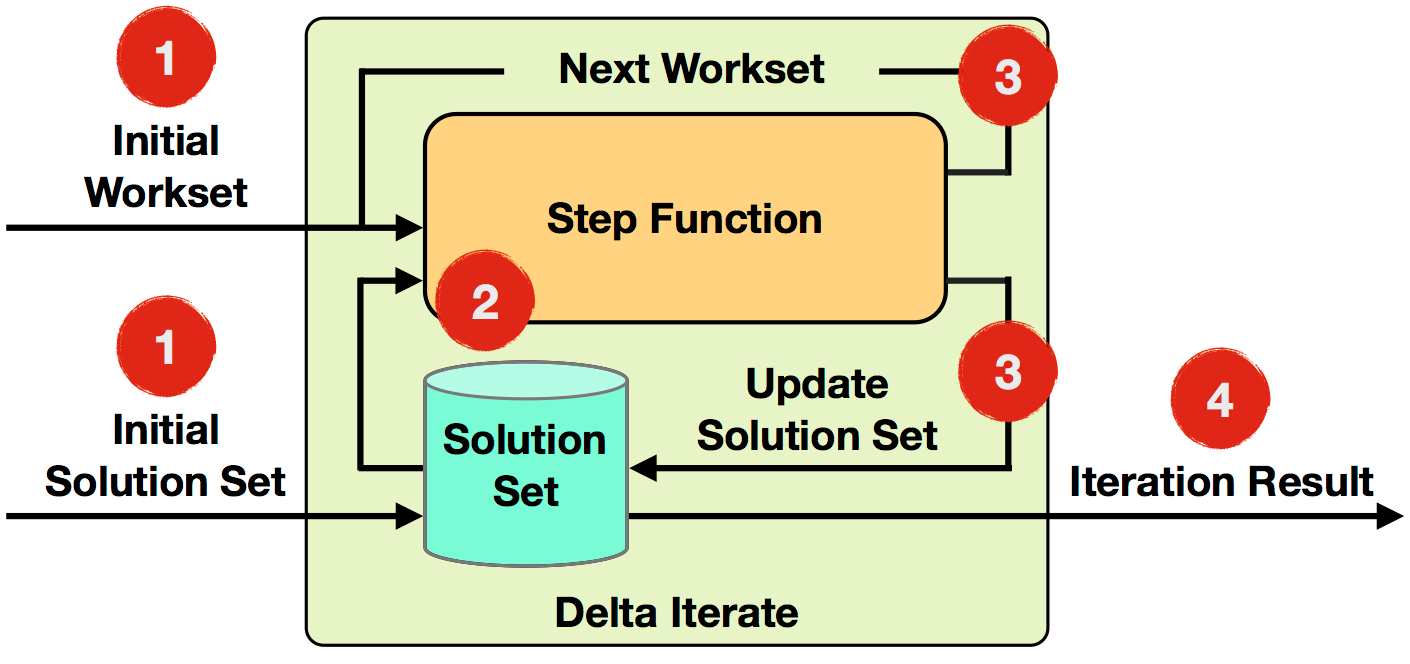
state = step(state);

}

setFinalState(state);

#### Delta Iterate

Delta Iterate Operator实现了增量迭代。



流程伪代码：

IterationState workset = getInitialState();

IterationState solution = getInitialSolution();

while (!terminationCriterion()) {

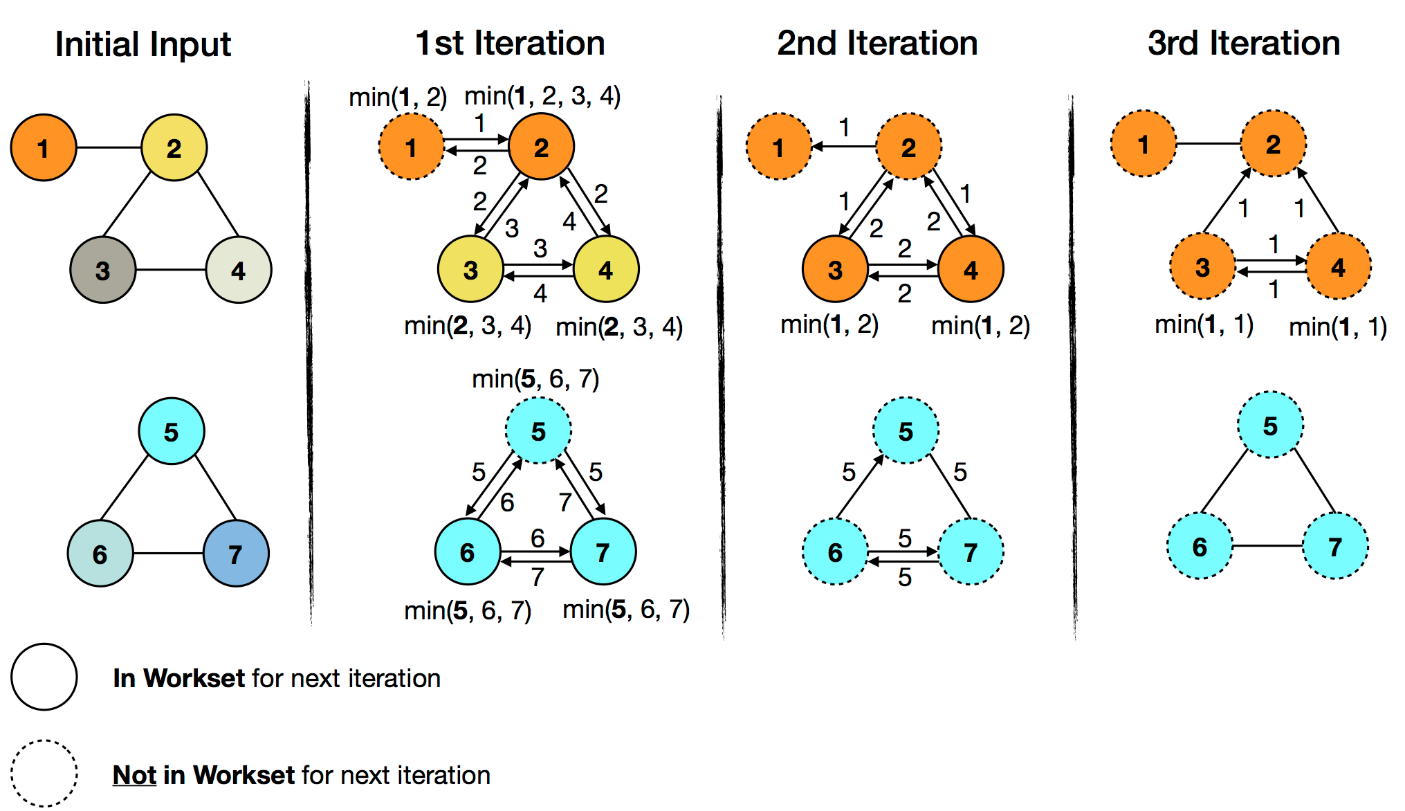
(delta, workset) = step(workset, solution);

solution.update(delta)

}

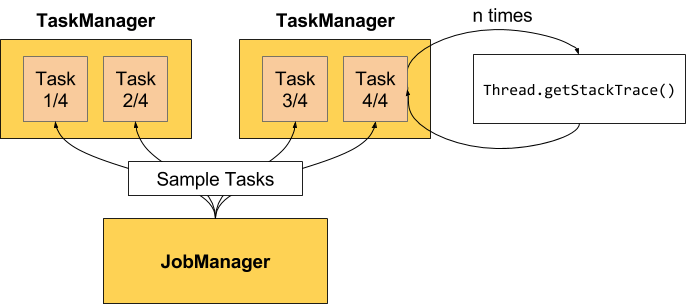
setFinalState(solution);

最小值传播：



### 9. [Back Pressure监控](https://ci.apache.org/projects/flink/flink-docs-release-1.2/monitoring/back_pressure.html)

流处理系统中，当下游Operator处理速度跟不上的情况，如果下游Operator能够将自己处理状态传播给上游Operator，使得上游Operator处理速度慢下来就会缓解上述问题，比如通过告警的方式通知现有流处理系统存在的问题。

Flink Web界面上提供了对运行Job的Backpressure行为的监控，它通过使用Sampling线程对正在运行的Task进行堆栈跟踪采样来实现。  


默认情况下，JobManager会每间隔50ms触发对一个Job的每个Task依次进行100次堆栈跟踪调用，过计算得到一个比值，例如，radio=0.01，表示100次中仅有1次方法调用阻塞。Flink目前定义了如下Backpressure状态：  
OK: 0 <= Ratio <= 0.10  
LOW: 0.10 < Ratio <= 0.5  
HIGH: 0.5 < Ratio <= 1

## 三、库

### 1. [Table](https://ci.apache.org/projects/flink/flink-docs-release-1.2/dev/table_api.html)

Flink的Table API实现了使用类SQL进行流和批处理。

详情参考：<https://ci.apache.org/projects/flink/flink-docs-release-1.2/dev/table_api.html>

### 2. [CEP](https://ci.apache.org/projects/flink/flink-docs-release-1.2/concepts/programming-model.html#next-steps)

Flink的CEP（Complex Event Processing）支持在流中发现复杂的事件模式，快速筛选用户感兴趣的数据。

详情参考：<https://ci.apache.org/projects/flink/flink-docs-release-1.2/concepts/programming-model.html#next-steps>

### 3. [Gelly](https://ci.apache.org/projects/flink/flink-docs-release-1.2/dev/libs/gelly/index.html)

Gelly是Flink提供的图计算API，提供了简化开发和构建图计算分析应用的接口。

详情参考：<https://ci.apache.org/projects/flink/flink-docs-release-1.2/dev/libs/gelly/index.html>

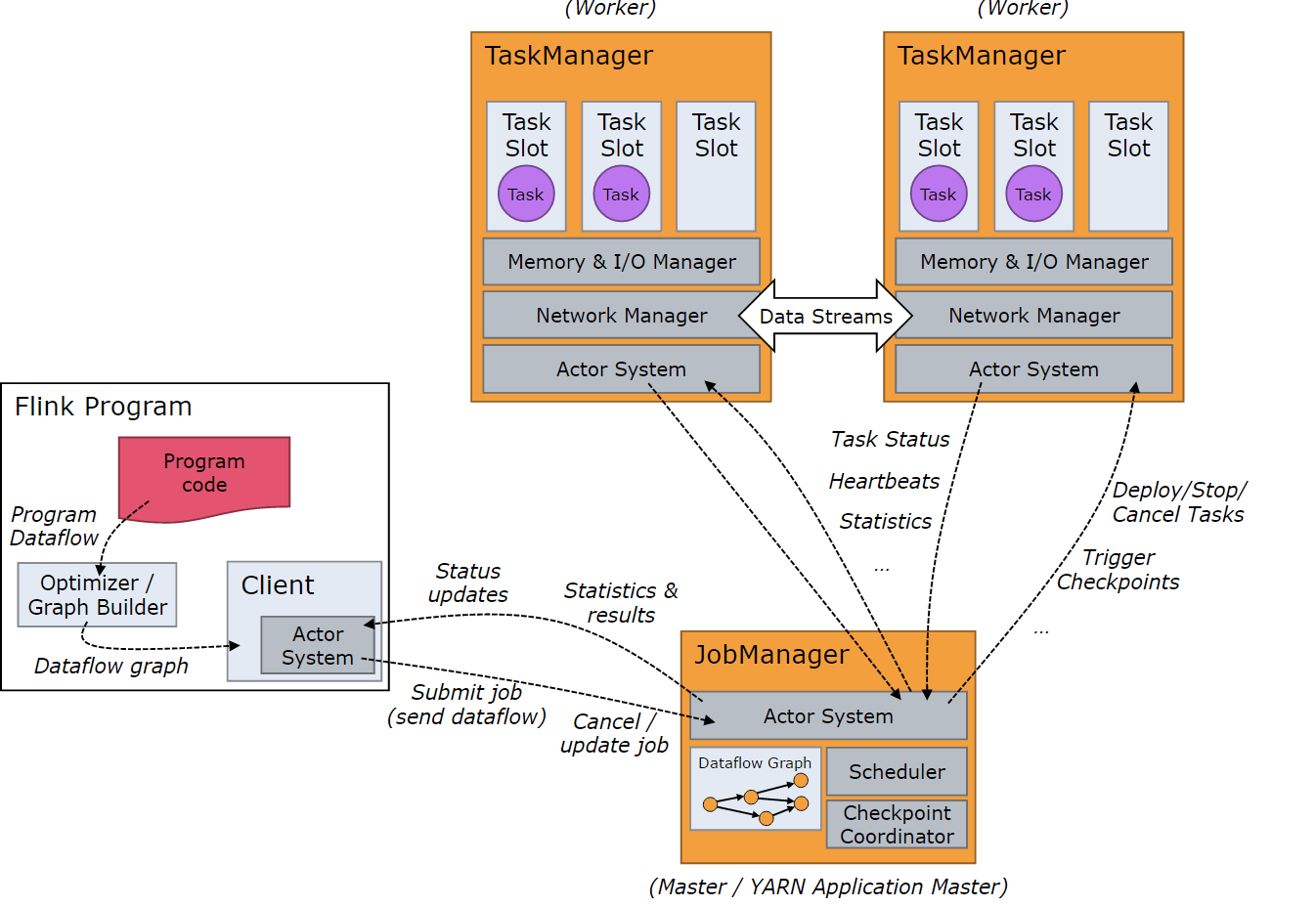
### 4. [FlinkML](https://ci.apache.org/projects/flink/flink-docs-release-1.2/dev/libs/ml/index.html)

FlinkML是Flink提供的机器学习库，提供了可扩展的机器学习算法、简洁的API和工具简化机器学习系统的开发。

详情参考：<https://ci.apache.org/projects/flink/flink-docs-release-1.2/dev/libs/ml/index.html>

## 四、部署

当Flink系统启动时，首先启动JobManager和一至多个TaskManager。JobManager负责协调Flink系统，TaskManager则是执行并行程序的worker。当系统以本地形式启动时，一个JobManager和一个TaskManager会启动在同一个JVM中。  
当一个程序被提交后，系统会创建一个Client来进行预处理，将程序转变成一个并行数据流的形式，交给JobManager和TaskManager执行。



### 1. 启动测试

编译flink，本地启动。

$ java -version

java version "1.8.0\_111"

$ git clone https://github.com/apache/flink.git

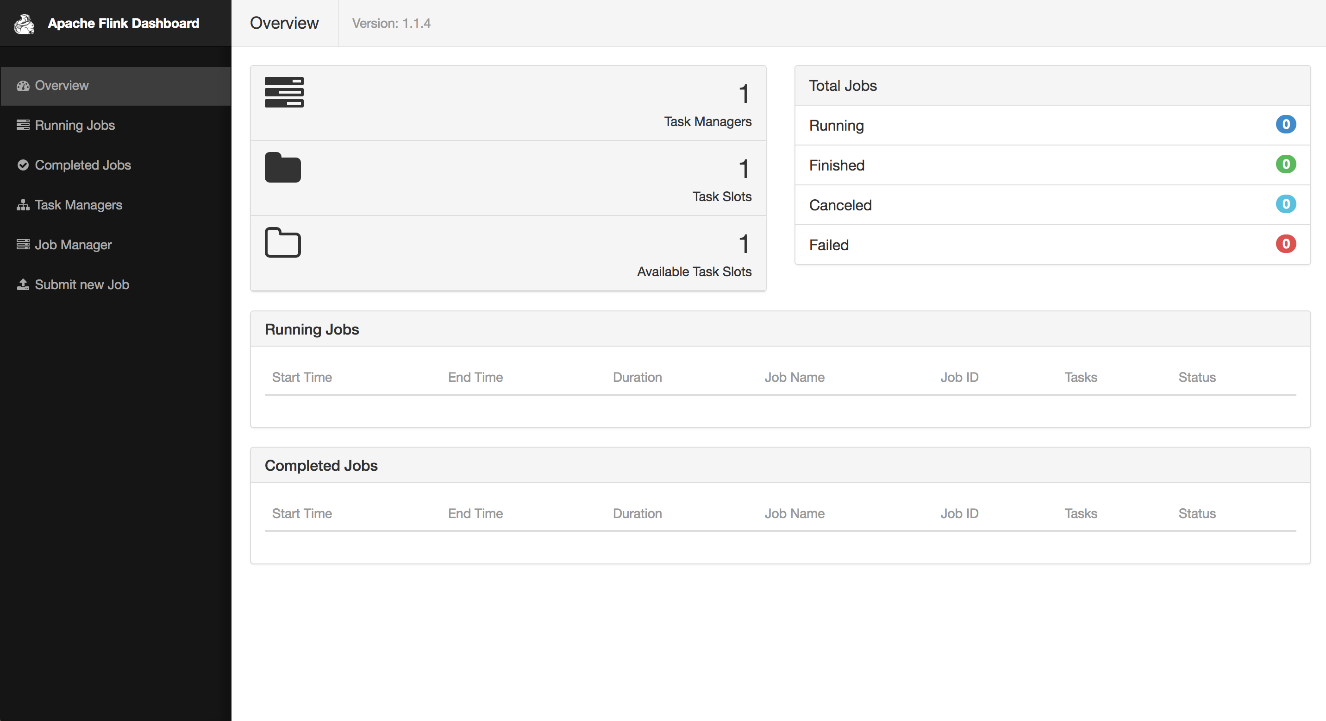
$ git checkout release-1.1.4 -b release-1.1.4

$ cd flink

$ mvn clean package -DskipTests

$ cd flink-dist/target/flink-1.1.4-bin/flink-1.1.4

$ ./bin/start-local.sh



编写本地流处理demo。

SocketWindowWordCount.[**Java**](http://lib.csdn.net/base/javase)

public class SocketWindowWordCount {

public static void main(String[] args) throws Exception {

// the port to connect to

final int port;

try {

final ParameterTool params = ParameterTool.fromArgs(args);

port = params.getInt("port");

} catch (Exception e) {

System.err.println("No port specified. Please run 'SocketWindowWordCount --port <port>'");

return;

}

// get the execution environment

final StreamExecutionEnvironment env = StreamExecutionEnvironment.getExecutionEnvironment();

// get input data by connecting to the socket

DataStream<String> text = env.socketTextStream("localhost", port, "\n");

// parse the data, group it, window it, and aggregate the counts

DataStream<WordWithCount> windowCounts = text

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

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

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

out.collect(new WordWithCount(word, 1L));

}

}

})

.keyBy("word")

.timeWindow(Time.seconds(5), Time.seconds(1))

.reduce(new ReduceFunction<WordWithCount>() {

public WordWithCount reduce(WordWithCount a, WordWithCount b) {

return new WordWithCount(a.word, a.count + b.count);

}

});

// print the results with a single thread, rather than in parallel

windowCounts.print().setParallelism(1);

env.execute("Socket Window WordCount");

}

// Data type for words with count

public static class WordWithCount {

public String word;

public long count;

public WordWithCount() {}

public WordWithCount(String word, long count) {

this.word = word;

this.count = count;

}

@Override

public String toString() {

return word + " : " + count;

}

}

}

pom.xml

<!-- Use this dependency if you are using the DataStream API -->

<dependency>

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

<artifactId>flink-streaming-java\_2.10</artifactId>

<version>1.1.4</version>

</dependency>

<!-- Use this dependency if you are using the DataSet API -->

<dependency>

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

<artifactId>flink-java</artifactId>

<version>1.1.4</version>

</dependency>

<dependency>

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

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

<version>1.1.4</version>

</dependency>

执行mvn构建。

$ mvn clean install

$ ls target/flink-demo-1.0-SNAPSHOT.jar

开启9000端口，用于输入数据：

$ nc -l 9000

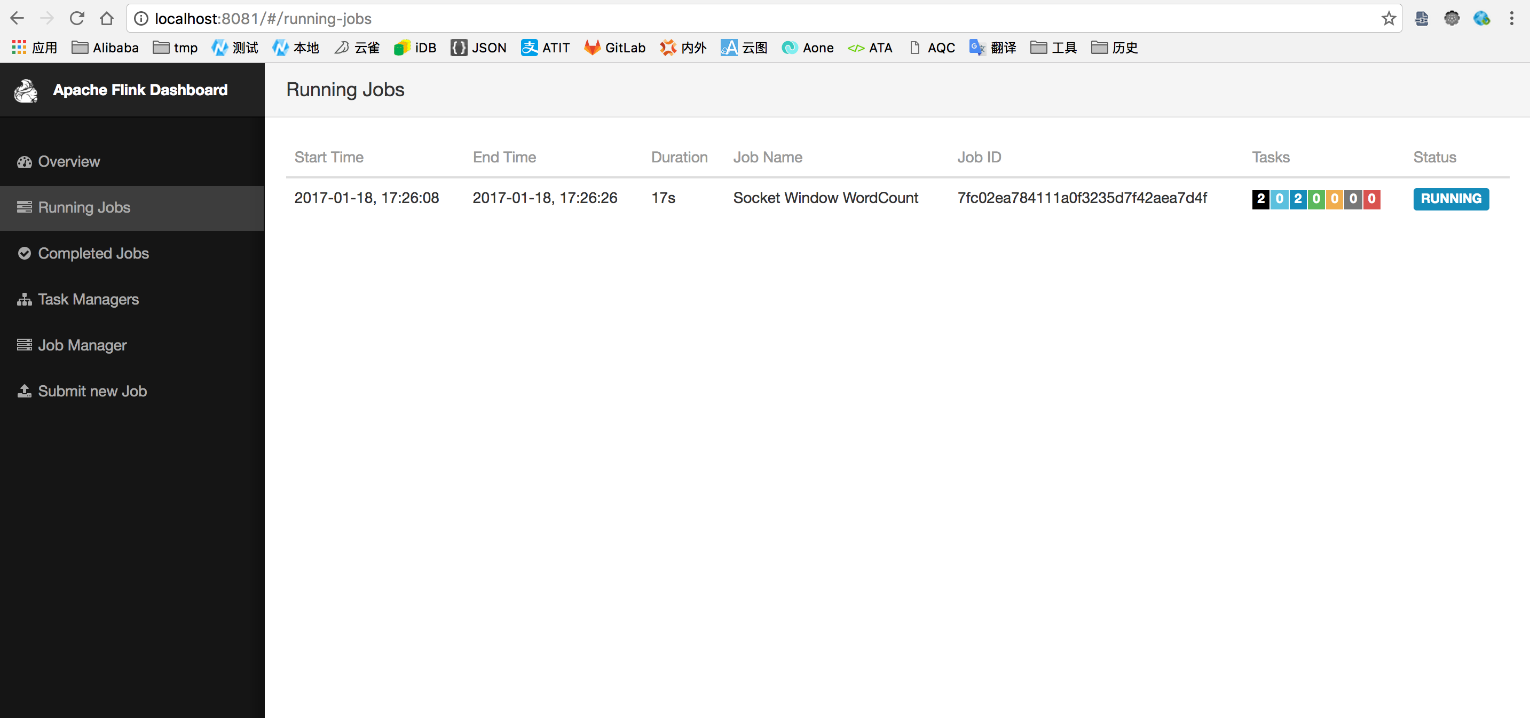
提交flink任务：

$ ./bin/flink run -c com.demo.florian.WordCount $DEMO\_DIR/target/flink-demo-1.0-SNAPSHOT.jar --port 9000

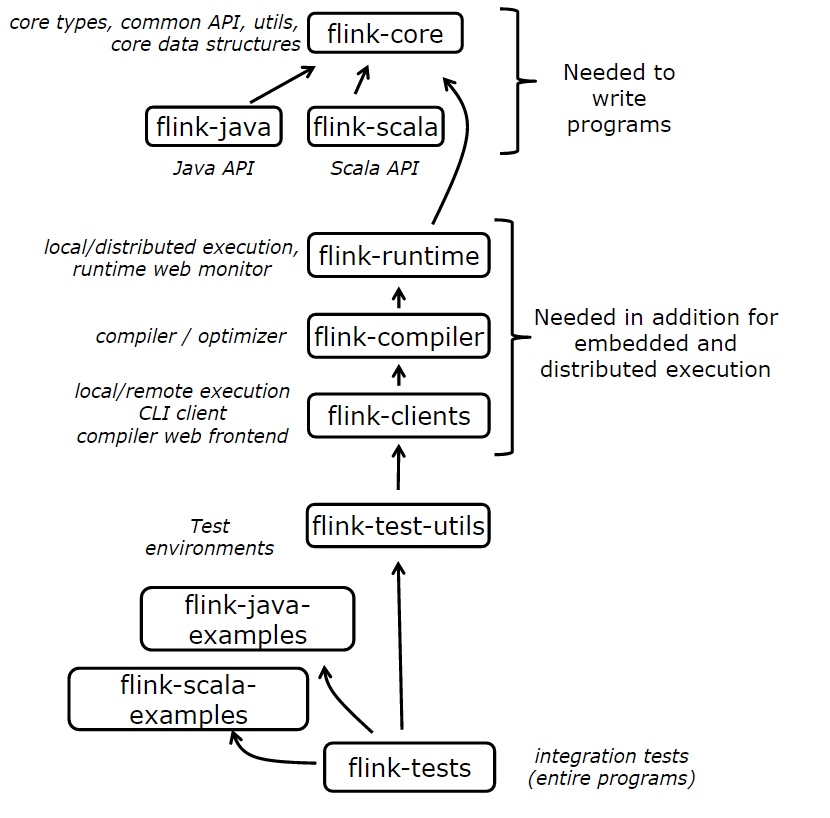
在nc里输入数据后，查看执行结果：

$ tail -f log/flink-\*-jobmanager-\*.out

查看flink web页面：localhost:8081



### 2. 代码结构

Flink系统核心可分为多个子项目。分割项目旨在减少开发Flink程序需要的依赖数量，并对[**测试**](http://lib.csdn.net/base/softwaretest)和开发小组件提供便捷。  


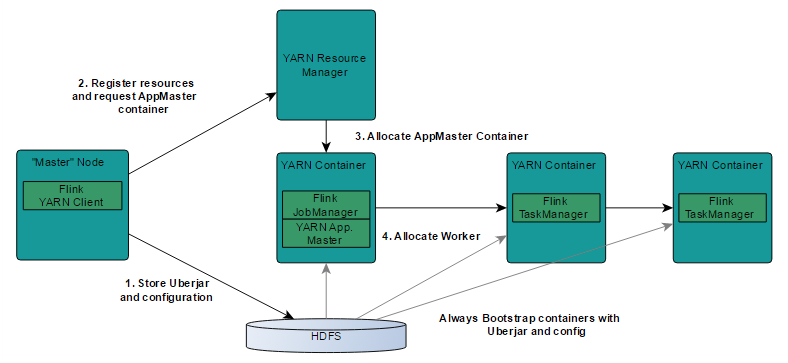
Flink当前还包括以下子项目：

1. Flink-dist：distribution项目。它定义了如何将编译后的代码、脚本和其他资源整合到最终可用的目录结构中。
2. Flink-quick-start：有关quickstart和教程的脚本、maven原型和示例程序
3. flink-contrib：一系列有用户开发的早起版本和有用的工具的项目。后期的代码主要由外部贡献者继续维护，被flink-contirb接受的代码的要求低于其他项目的要求。

### 3. Flink On YARN

Flink在YARN集群上运行时：Flink YARN Client负责与YARN RM通信协商资源请求，Flink JobManager和Flink TaskManager分别申请到[**Container**](http://lib.csdn.net/base/docker)去运行各自的进程。

YARN AM与Flink JobManager在同一个Container中，这样AM可以知道Flink JobManager的地址，从而AM可以申请Container去启动Flink TaskManager。待Flink成功运行在YARN集群上，Flink YARN Client就可以提交Flink Job到Flink JobManager，并进行后续的映射、调度和计算处理。



1. 设置Hadoop环境变量

$ export HADOOP\_CONF\_DIR=/etc/hadoop/conf

1. 以集群模式提交任务，每次都会新建flink集群

$ ./bin/flink run -m yarn-cluster -c com.demo.florian.WordCount $DEMO\_DIR/target/flink-demo-1.0-SNAPSHOT.jar

1. 启动共享flink集群，提交任务

$ ./bin/yarn-session.sh -n 4 -jm 1024 -tm 4096 -d

$ ./bin/flink run -c com.demo.florian.WordCount $DEMO\_DIR/target/flink-demo-1.0.SNAPSHOT.jar