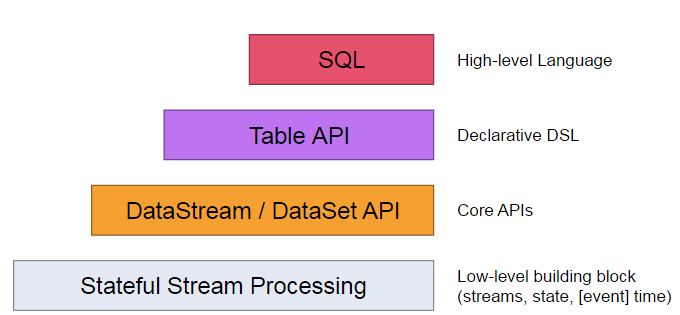
**Flink**

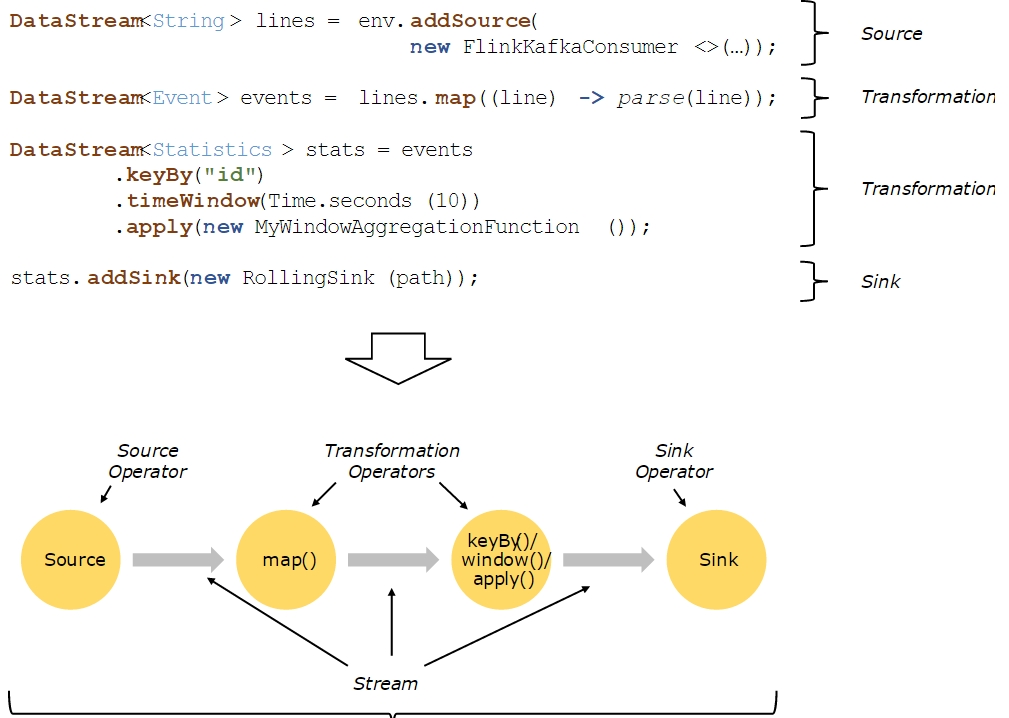
# [Dataflow Programming Model](https://ci.apache.org/projects/flink/flink-docs-release-1.2/concepts/programming-model.html)

## 1．Abstract Level



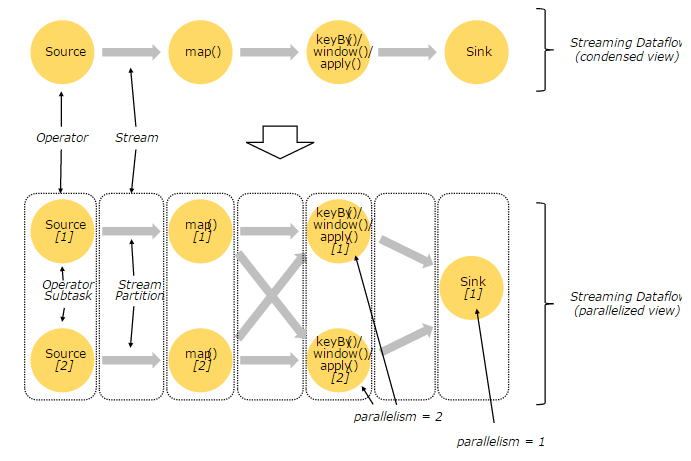
1. 最低层次的抽象仅仅提供stateful streaming 。能够将数据转换成DataStream，允许用户使用一个或者多个streams数据，并且保证使用一致性容错，用户可是注册事件时间，并用回调函数，处理复杂的计算（当然大多数情况是不需要的）。
2. 程序基于主要的API进行开发，比如 [DataStream API](https://ci.apache.org/projects/flink/flink-docs-release-1.2/dev/datastream_api.html) (bounded/unbounded streams) 以及 [DataSet API](https://ci.apache.org/projects/flink/flink-docs-release-1.2/dev/batch/index.html) (bounded data sets)。这些流式API提供transformations, joins, aggregations, windows, state, 等操作。这些数据的类型是作为class存在。
3. 在TableAPI 表中数据会随着streams的改变。并且提供一些SQL类似的操作：select, project, join, group-by, aggregate以及自定义UDF并且进行优化器优化之后执行。TableAPI可以和DataStream and DataSet APIs进行无缝转换。
4. 支持sql语句查询

## 2. Programs and Dataflows



## 3. Parallel Dataflows

并行处理的流程图：



* **One-to-one** streams :

**如上面的数据从source到map的处理，将会保持原有的partition，以及数据的排序，这就意味着map** operator**的子任务将会处理**Source operator**的对应的子任务的数据。**

* **Redistributing** streams :

**如同上面的map和keyBy/window，以及keyBy/window 和sink之间数据流的partition将会改变，每种操作的子任务将会把数据发送到不通的目标task进行处理，这取决于变换操作。例如keyBy() (基于hash算法进行重分区), broadcast(),rebalance() (随机进行分区)。再一次重分配，交换元素的排序仅仅保留每一对发送和接收子任务（比如在map的子任务一和keyBy/window的子任务二中将保留排序）。在这个例子中,虽然要求在每个key的顺序是保留的,但是由于不同key的聚合结果的到达顺序使并行性并引入非确定性。**

# Flink DataStream API Programming Guide

## OverView

* [DataStream Transformations](https://ci.apache.org/projects/flink/flink-docs-release-1.2/dev/datastream_api.html#datastream-transformations)
* [Data Sources](https://ci.apache.org/projects/flink/flink-docs-release-1.2/dev/datastream_api.html#data-sources)
* [Data Sinks](https://ci.apache.org/projects/flink/flink-docs-release-1.2/dev/datastream_api.html#data-sinks)
* [Iterations](https://ci.apache.org/projects/flink/flink-docs-release-1.2/dev/datastream_api.html#iterations)
* [Execution Parameters](https://ci.apache.org/projects/flink/flink-docs-release-1.2/dev/datastream_api.html#execution-parameters)
* [Fault Tolerance](https://ci.apache.org/projects/flink/flink-docs-release-1.2/dev/datastream_api.html#fault-tolerance)
* [Controlling Latency](https://ci.apache.org/projects/flink/flink-docs-release-1.2/dev/datastream_api.html#controlling-latency)
* [Debugging](https://ci.apache.org/projects/flink/flink-docs-release-1.2/dev/datastream_api.html#debugging)

重点特性iterator的使用示例：

StreamExecutionEnvironment env = StreamExecutionEnvironment

.*getExecutionEnvironment*();

DataStream<Long> someIntegers = env.generateSequence(0, 1000);

IterativeStream<Long> iteration = someIntegers.iterate();

DataStream<Long> minusOne = iteration

.map(**new** MapFunction<Long, Long>() {

**private** **static** **final** **long** ***serialVersionUID*** = 1L;

@Override

**public** Long map(Long value) **throws** Exception {

System.***out***.println("-----step 1 map " +value);

**return** value - 1;

}

});

DataStream<Long> stillGreaterThanZero = minusOne

.filter(**new** FilterFunction<Long>() {

**private** **static** **final** **long** ***serialVersionUID*** = 1L;

@Override

**public** **boolean** filter(Long value) **throws** Exception {

System.***out***.println("-----step 2 filter " +value);

**return** (value > 0);

}

});

iteration.closeWith(stillGreaterThanZero);

DataStream<Long> lessThanZero = minusOne

.filter(**new** FilterFunction<Long>() {

**private** **static** **final** **long** ***serialVersionUID*** = 6135931968532483923L;

@Override

**public** **boolean** filter(Long value) **throws** Exception {

System.***out***.println("-----step 3 filter " +value);

**return** (value <= 0);

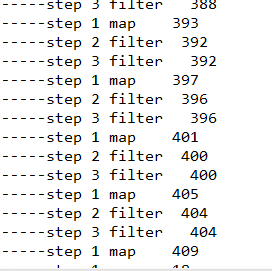
}

});

lessThanZero.print ();

env.execute(" my iteration");

执行结果图：



延迟控制：

默认的元素不会一个个的经过网络传输，而是采用缓冲的形式批量发送，来避免不必要的网络开销。

Advantage: optimizing throughput

Disadvantage: cause latency issues when the incoming stream is not fast enough

Default 100 ms.

LocalStreamEnvironment env **=** StreamExecutionEnvironment**.**createLocalEnvironment**();**

env**.**setBufferTimeout**(**timeoutMillis**);**

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

## 2. Window Lifecycle

In a nutshell, a window is **created** as soon as the first element that should belong to this window arrives, and the window is **completely removed** when the time (event or processing time) passes its end timestamp plus the user-specified allowed lateness.

Flink will create a new window for the interval between 12:00 and 12:05 when the first element with a timestamp that falls into this interval arrives, and it will remove it when the watermark passes the 12:06 timestamp.

Window have a Trigger， Function and Evictor.

## 3. Keyed vs Non-Keyed Windows

1. Having a keyed stream will allow your windowed computation to be performed in parallel by multiple tasks, as each logical keyed stream can be processed independently from the rest.

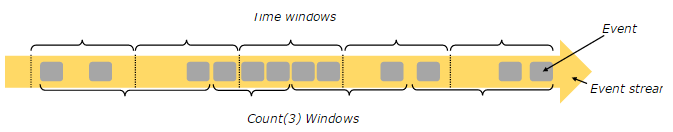
2. In case of non-keyed streams, your original stream will not be split into multiple logical streams and all the windowing logic will be performed by a single task, i.e. with parallelism of 1.

## 4. [Window](https://ci.apache.org/projects/flink/flink-docs-release-1.2/dev/windows.html#window-assigners) [assigner](https://ci.apache.org/projects/flink/flink-docs-release-1.2/dev/windows.html#window-assigners)

### *4.1 Feature：*

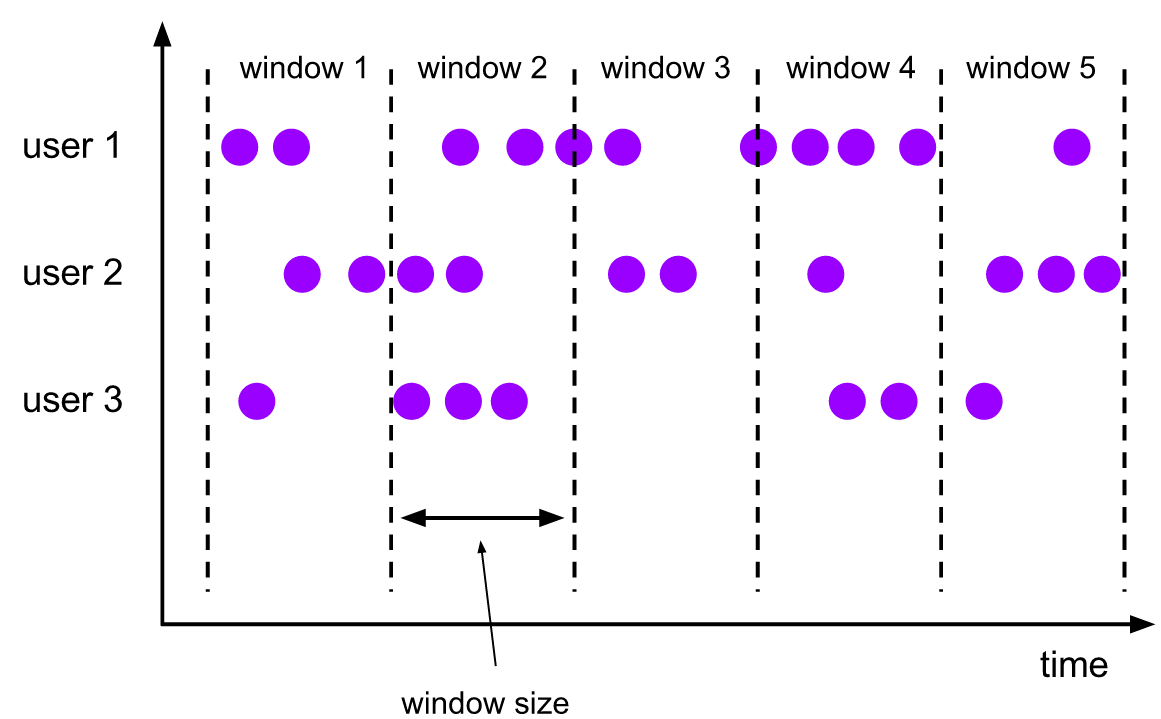
基于时间驱动 example: every 30 seconds

基于数据驱动 example: every 100 elements



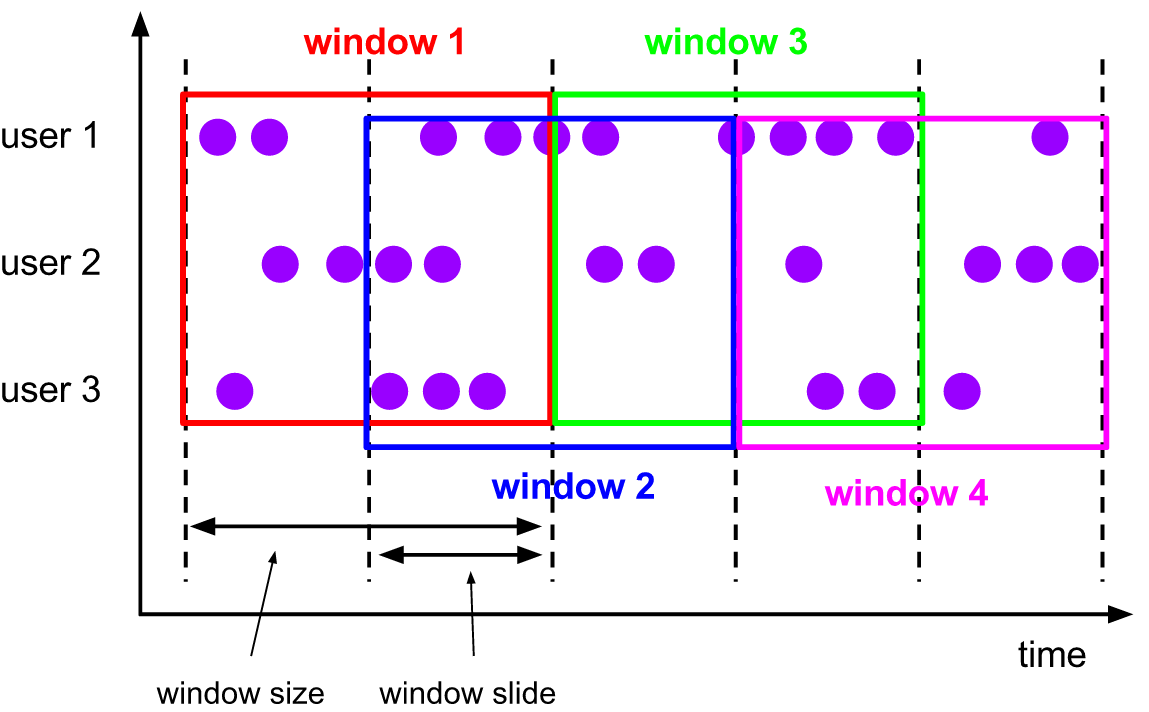
### *4.2 Window Types:*

### *4.2.1 tumbling windows* (no overlapping 无重叠的窗口)

[](https://ci.apache.org/projects/flink/flink-docs-release-1.2/dev/windows.html#window-lifecycle)

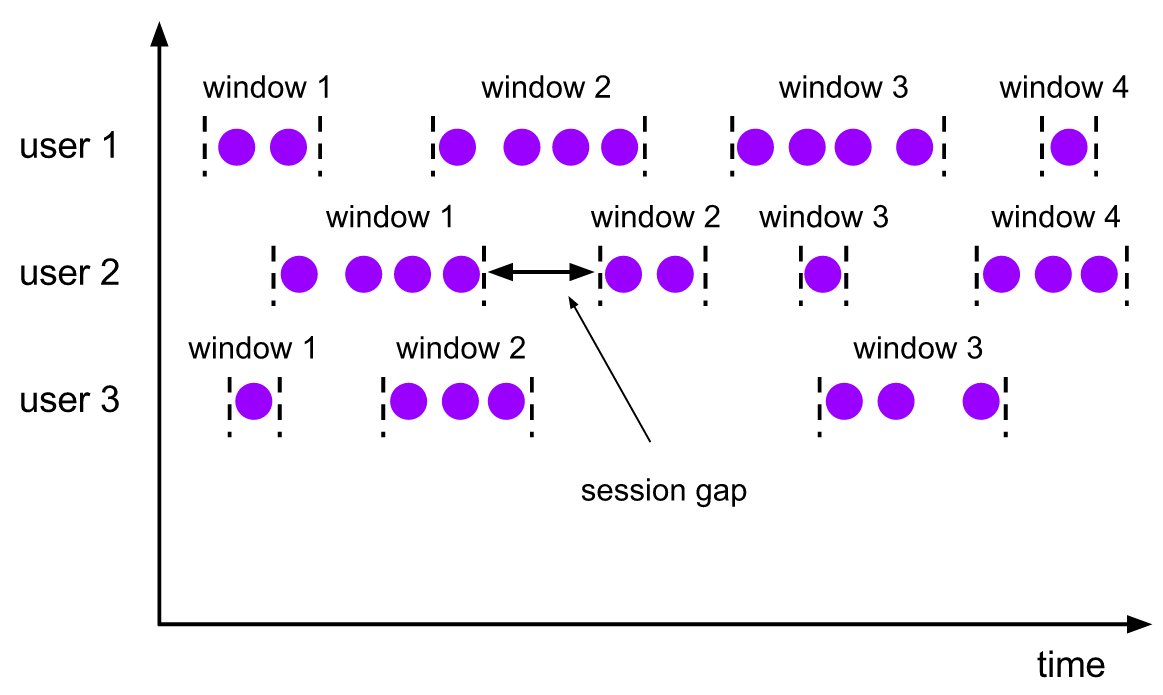
TumblingEventTimeWindows**.**of**(**Time**.**days**(**1**),** Time**.**hours**(-**8**))**

### *4.2.2 sliding windows* (可以重叠的滑动窗口)

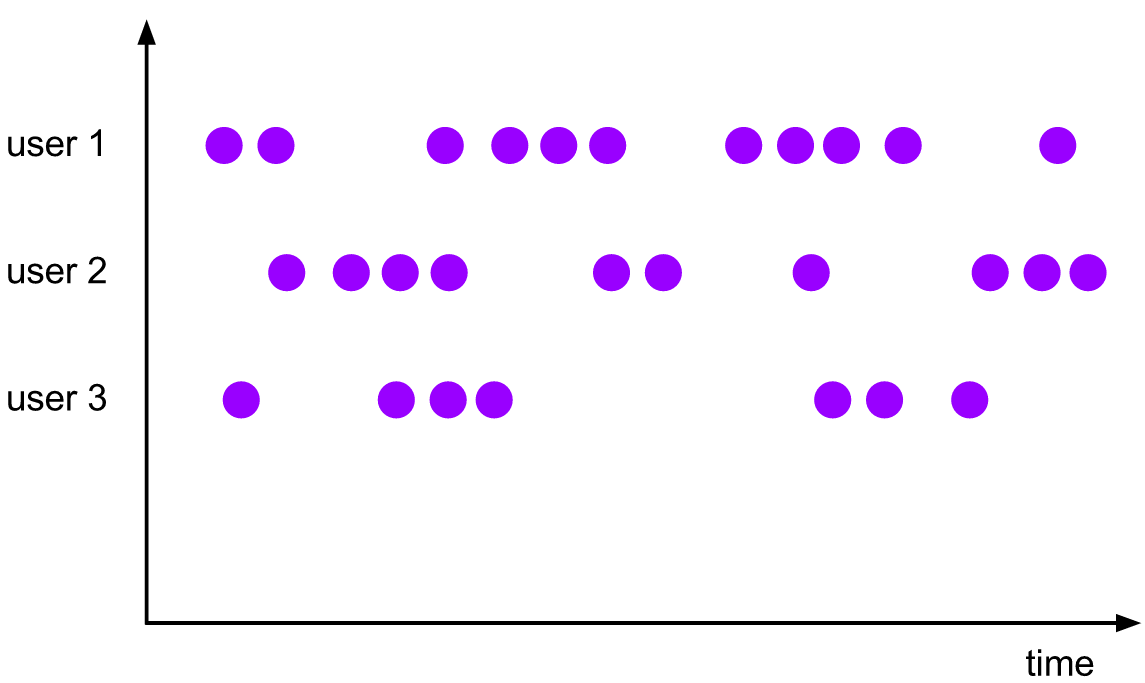


**SlidingEventTimeWindows.of(Time.seconds(10), Time.seconds(5))**

### *4.2.3 session windows* (会话超时窗口，窗口宽度取决于session gap”)

ProcessingTimeSessionWindows**.**withGap**(**Time**.**minutes**(**10**))**

### 4.2.4 global windows

 **GlobalWindows.create()**

**Recipes for Building Windows**

**window 分配器，trigger，evictor的机制都功能强大。这些机制让你可以定义各种不同类型的window。Flink’s的基本window其实是在这三个机制之上包了一层的，.下面是一些通用的端口是如何通过这三种机制来构造的 。**

| **Window type** | **Definition** |
| --- | --- |
| **Tumbling count window**  **stream.countWindow(1000)** | **stream.window(GlobalWindows.create())**  **.trigger(CountTrigger.of(1000)**  **.evict(CountEvictor.of(1000)))** |
| **Sliding count window**  **stream.countWindow(1000, 100)** | **stream.window(GlobalWindows.create())**  **.trigger(CountTrigger.of(1000)**  **.evict(CountEvictor.of(100)))** |
| **Tumbling event time window**  **stream.timeWindow(Time.of(5, TimeUnit.SECONDS))** | **stream.window(TumblingTimeWindows.of((Time.of(5,**  **TimeUnit.SECONDS)))**  **.trigger(EventTimeTrigger.create())** |
| **Sliding event time window**  **stream.timeWindow(Time.of(5, TimeUnit.SECONDS),**  **Time.of(1, TimeUnit.SECONDS))** | **stream.window(SlidingTimeWindows.of(Time.of(5,**  **TimeUnit.SECONDS), Time.of(1, TimeUnit.SECONDS)))**  **.trigger(EventTimeTrigger.create())** |
| **Tumbling processing time window**  **stream.timeWindow(Time.of(5, TimeUnit.SECONDS))** | **stream.window(TumblingTimeWindows.of((Time.of(5,**  **TimeUnit.SECONDS)))**  **.trigger(ProcessingTimeTrigger.create())** |
| **Sliding processing time window**  **stream.timeWindow(Time.of(5, TimeUnit.SECONDS), Time.of(1, TimeUnit.SECONDS))** | **stream.window(SlidingTimeWindows.of(Time.of(5,**  **TimeUnit.SECONDS), Time.of(1, TimeUnit.SECONDS)))**  **.trigger(ProcessingTimeTrigger.create())** |

## Window Functions

ReduceFunction, FoldFunction : Flink can incrementally aggregate the elements for each window as they arrive.

WindowFunction:Flink has to buffer all elements for a window internally before invoking the function

### ReduceFunction

DataStream**<**Tuple2**<**String**,** Long**>>** input **=** **...;**

input

**.**keyBy**(<**key selector**>)**

**.**window**(<**window assigner**>)**

**.**reduce**(new** ReduceFunction**<**Tuple2**<**String**,** Long**>>** **{**

**public** Tuple2**<**String**,** Long**>** **reduce(**Tuple2**<**String**,** Long**>** v1**,** Tuple2**<**String**,** Long**>** v2**)** **{**

**return** **new** Tuple2**<>(**v1**.**f0**,** v1**.**f1 **+** v2**.**f1**);**

**}**

**});**

### FoldFunction

DataStream**<**Tuple2**<**String**,** Long**>>** input **=** **...;**

input

**.**keyBy**(<**key selector**>)**

**.**window**(<**window assigner**>)**

**.**fold**(**""**,** **new** FoldFunction**<**Tuple2**<**String**,** Long**>,** String**>>** **{**

**public** String **fold(**String acc**,** Tuple2**<**String**,** Long**>** value**)** **{**

**return** acc **+** value**.**f1**;**

**}**

**});**

#### Window Function :Incremental Window Aggregation with FoldFunction

DataStream**<**SensorReading**>** input **=** **...;**

input

**.**keyBy**(<**key selector**>)**

**.**timeWindow**(<**window assigner**>)**

**.**reduce**(new** **MyReduceFunction(),** **new** **MyWindowFunction());**

*// Function definitions*

**private** **static** **class** **MyReduceFunction** **implements** ReduceFunction**<**SensorReading**>** **{**

**public** SensorReading **reduce(**SensorReading r1**,** SensorReading r2**)** **{**

**return** r1**.**value**()** **>** r2**.**value**()** **?** r2 **:** r1**;**

**}**

**}**

**private** **static** **class** **MyWindowFunction**

**implements** WindowFunction**<**SensorReading**,** Tuple2**<**Long**,** SensorReading**>,** String**,** TimeWindow**>** **{**

**public** **void** **apply(**String key**,**

TimeWindow window**,**

Iterable**<**SensorReading**>** minReadings**,**

Collector**<**Tuple2**<**Long**,** SensorReading**>>** out**)** **{**

SensorReading min **=** minReadings**.**iterator**().**next**();**

out**.**collect**(new** Tuple2**<**Long**,** SensorReading**>(**window**.**getStart**(),** min**));**

**}**

**}**

## Triggers

每个WindowAssigner 都有默认的Trigger：

Trigger提供了五中方法来处理不同的事件:

* onElement() 窗口中的每个元素都会被执行.
* onEventTime()当注册的事件触发器触发
* onProcessingTime() 当注册的时间触发器触发
* onMerge() 针对于有状态的Trigger，当相对应的window merge的时候合并两者的状态，比如session window
* clear()当对应的窗口需要执行移除操作的时候

注意:

1.前三种决定了怎么处理对应的触发事件，并返回一个TriggerResult：

* CONTINUE: 什么也不做
* FIRE: 触发计算
* PURGE: 清除窗口的元素
* FIRE\_AND\_PURGE: 触发计算然后清除元素

2.任何一种方法都能被注册为时间或者事件触发器

Trigger定义了，跟在每一个的window后面的函数（sum，count），什么时候evaluated (“fires”)。如果没有指定trigger，就使用默认的trigger. Flink 自带了一组trigger，如果默认的trigger都没法满足你的应用，可以通过实现Trigger接口实现自己的trigger. 注意，如果使用自定义trigger后，会覆盖默认的trigger.

| **Transformation** | **Description** |
| --- | --- |
| **Processing time trigger** | 当前的处理时间超过他的end-value时，则发射一个window，  从此之后，被跟踪的window上的元素就会被丢弃。  windowedStream**.**trigger**(**ProcessingTimeTrigger**.**create**());** |
| **Watermark trigger** | 当接收到一个超过end value的watermark时，则发射一个window。  被跟踪的window上的元素就会被丢弃。  windowedStream**.**trigger**(**EventTimeTrigger**.**create**());** |
| **Continuous processing**  **time trigger** | 每个being fire 的 window会定期的考虑if（）。当当前时间超过他的  end-value的时候，才会真正发射，被触发的窗口里的函数将会保留。  windowedStream**.**trigger**(**ContinuousProcessingTimeTrigger**.**of**(**Time**.**of**(**5**,**  TimeUnit**.**SECONDS**)));** |
| **Continuous watermark**  **time trigger** | 每个being fire 的 window会定期的考虑if（）。当watermark时间超过他的  end-value的时候，才会真正发射，被触发的窗口里的函数将会保留。  windowedStream**.**trigger**(**ContinuousEventTimeTrigger**.**of**(**Time**.**of**(**5**,**  TimeUnit**.**SECONDS**)));** |
| **Count trigger** | 超过1000个元素后，这个窗口就会被发射，处于准备发射状态的窗口里的  元素，将会被保留。  windowedStream**.**trigger**(**CountTrigger**.**of**(**1000**));** |
| **Purging trigger** | Takes any trigger as an argument and forces the triggered window  elements to be "purged" (discarded) after triggering.  windowedStream**.**trigger**(**PurgingTrigger**.**of**(**CountTrigger**.**of**(**1000**)));** |
| **Delta trigger** | 每个being fire 的 window会定期的考虑if（）。当最后一个元素和第一个  插入的元素运算后满足true的时候，才会真正发射。  windowedStream**.**trigger**(new** DeltaTrigger**.**of**(**5000.0**,**  **new** DeltaFunction**<**Double**>()** **{**  @Override  **public** **double** **getDelta** **(**Double old**,** Double **new)** **{**  **return** **(new** **-** old **>** 0.01**);**  **}**  **}));** |

## Evictors

The evictor has the ability to remove elements from a window after the trigger fires and before and/or after the window function is applied.

Optionally evicts elements. Called before windowing function.

void evictBefore(Iterable<TimestampedValue<T>> elements, int size, W window, EvictorContext evictorContext);

Optionally evicts elements. Called after windowing function

void evictAfter(Iterable<TimestampedValue<T>> elements, int size, W window, EvictorContext evictorContext);

注意点：

evictor阻止任何的pre-aggregation，因为元素在计算之前必须传递到evictor函数

window的数据不能保证数据的顺序，evictor同样也不保证窗口开始数据的顺序

当trigger进行了fire之后, 并且执行sum和count之前, 有一个可选的逐出器可以移除保留元素。. Flink 自带了一组evictors ，你还可以通过实现Evictor接口，实现自定义的逐出器。.

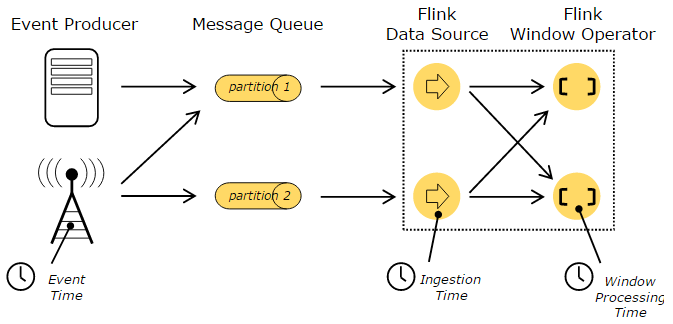
| **Transformation** | **Description** |
| --- | --- |
| **Time evictor** | 从window的begin处开始移除元素，知道最后剩下 end value -1秒到  end value的元素。  triggeredStream**.**evict**(**TimeEvictor**.**of**(**Time**.**of**(**1**,** TimeUnit**.**SECONDS**)));** |
| **Count evictor** | 保留倒数的最后1000 元素，其他的丢掉。  triggeredStream**.**evict**(**CountEvictor**.**of**(**1000**));** |
| **Delta evictor** | 从window的begin开始，一直丢元素，知道某个元素，8比最后一个元素15小5。  (通过一个阈值5 ，和一个函数).  triggeredStream**.**evict**(**DeltaEvictor**.**of**(**5000**,**  **new** DeltaFunction**<**Double**>()** **{**  **public** **double** **(**Double oldValue**,** Double newValue**)** **{**  **return** newValue **-** oldValue**;**  **}**  **}));** |

## Time

**Event Time**: Event time is the time that each individual event occurred on its producing device. 时间产生的时间点,能够保证无序的数据或者是延迟的数据以及失败后的从备份或者持久化的文件恢复时得到正确的结果。通常需要结合process time使用，需要手动指定如何获取event time 以及watermark。

**Ingestion time**: Flink source operator接收到消息的时间点，不能处理任何无序事件数据,但这些项目没有指定如何生成水印。

**Processing Time**:每个operator执行一个基于时间操作的时间点



**final** StreamExecutionEnvironment env **=** StreamExecutionEnvironment**.**getExecutionEnvironment**();**

env**.**setStreamTimeCharacteristic**(**TimeCharacteristic**.**ProcessingTime**);**

*// alternatively:*

*// env.setStreamTimeCharacteristic(TimeCharacteristic.IngestionTime);*

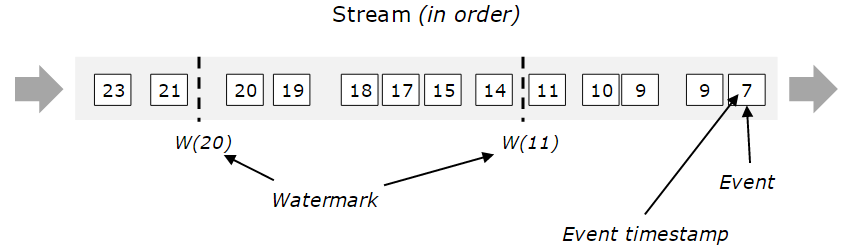
*// env.setStreamTimeCharacteristic(TimeCharacteristic.EventTime);*

## Event Time and Watermarks

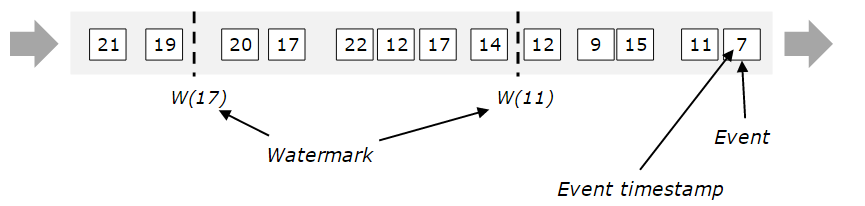
A stream processor that supports event time needs a way to measure the progress of event time.

Event time can progress independently of processing time (measured by wall clocks).

The mechanism in Flink to measure progress in event time is **watermarks**.

Watermarks flow as part of the data stream and carry a timestamp t. A Watermark(t) declares that event time has reached time t in that stream, meaning that there should be no more elements from the stream with a timestamp t’ <= t (i.e. events with timestamps older or equal to the watermark，带有时间戳的时间必须早于或者是等于watermark). 

Watermarks are crucial for out-of-order streams, as illustrated below, where the events are not ordered by their timestamps. In general a watermark is a declaration that by that point in the stream, all events up to a certain timestamp should have arrived. Once a watermark reaches an operator, the operator can advance its internal event time clock to the value of the watermark.



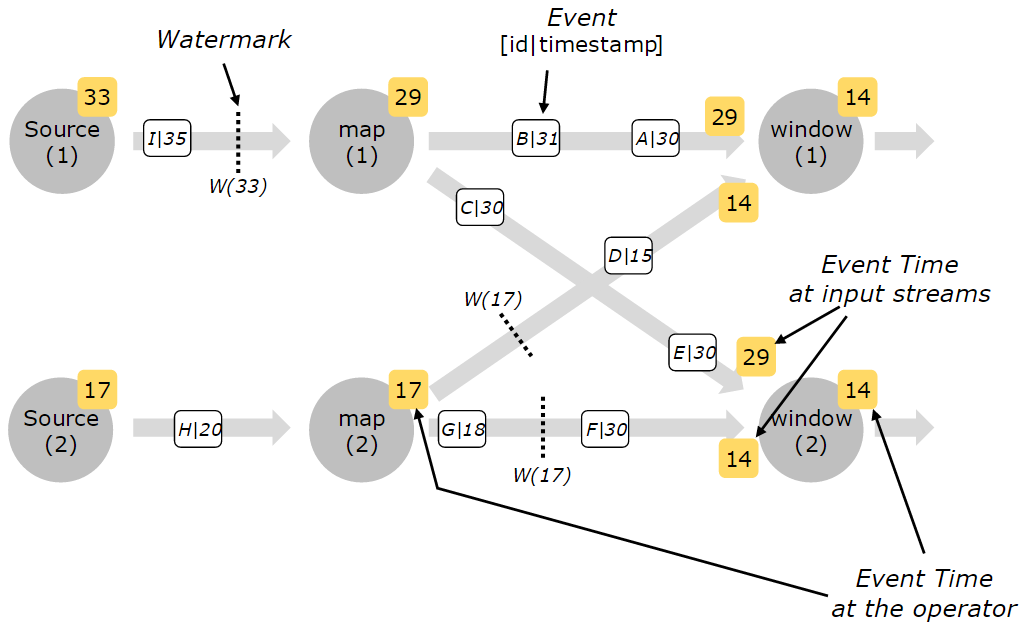
图片的理解：watermark就是一个时间印记，记录已经来的抵达事件中的数据的时间戳的最大值来标识已经收到数据的时间，在operator操作中更新该标识。

## Watermarks in Parallel Streams

Each parallel subtask of a source function usually generates its watermarks independently.

As the watermarks flow through the streaming program, they advance the event time at the operators where they arrive. Whenever an operator advances its event time, it generates a new watermark downstream for its successor operators.

Such an operator’s current event time is the minimum of its input streams’ event times. As its input streams update their event times, so does the operator.



## Allowed Lateness

Late elements are dropped when the watermark is past the end of the window.

Allowed lateness specifies by how much time elements can be late before they are dropped, and its default value is 0. Elements that arrive after the watermark has passed the end of the window but before it passes the end of the window plus the allowed lateness, are still added to the window.

DataStream**<**T**>** input **=** **...;**

input

**.**keyBy**(<**key selector**>)**

**.**window**(<**window assigner**>)**

**.**allowedLateness**(<**time**>)**

**.<**windowed transformation**>(<**window function**>);**

### Late elements considerations

These firings are called late firings, as they are triggered by late events and in contrast to the main firing which is the first firing of the window. In case of session windows, late firings can further lead to merging of windows, as they may “bridge” the gap between two pre-existing, unmerged windows.

**Attention**:

Depending on your application, you need to take these duplicated results into account or deduplicate them.

## Generating Timestamps/Watermarks

There are two ways to assign timestamps and generate watermarks:

1. Directly in the data stream source
2. Via a timestamp assigner / watermark generator: in Flink timestamp assigners also define the watermarks to be emitted

### Source Functions with Timestamps and Watermarks

@Override

**public** **void** **run(**SourceContext**<**MyType**>** ctx**)** **throws** Exception **{**

**while** **(***/\* condition \*/***)** **{**

MyType next **=** getNext**();**

ctx**.**collectWithTimestamp**(**next**,** next**.**getEventTimestamp**());**

**if** **(**next**.**hasWatermarkTime**())** **{**

ctx**.**emitWatermark**(new** **Watermark(**next**.**getWatermarkTime**()));**

**}**

**}**

**}**

### Timestamp Assigners / Watermark Generators

If the original stream had timestamps and/or watermarks already, the timestamp assigner overwrites them.

For example, is to parse (MapFunction) and filter (FilterFunction) before the timestamp assigner

**final** StreamExecutionEnvironment env **=** StreamExecutionEnvironment**.**getExecutionEnvironment**();**

env**.**setStreamTimeCharacteristic**(**TimeCharacteristic**.**EventTime**);**

DataStream**<**MyEvent**>** stream **=** env**.**readFile**(**

myFormat**,** myFilePath**,** FileProcessingMode**.**PROCESS\_CONTINUOUSLY**,** 100**,**

FilePathFilter**.**createDefaultFilter**(),** typeInfo**);**

DataStream**<**MyEvent**>** withTimestampsAndWatermarks **=** stream

**.**filter**(** event **->** event**.**severity**()** **==** WARNING **)**

**.**assignTimestampsAndWatermarks**(new** **MyTimestampsAndWatermarks());**

withTimestampsAndWatermarks

**.**keyBy**(** **(**event**)** **->** event**.**getGroup**()** **)**

**.**timeWindow**(**Time**.**seconds**(**10**))**

**.**reduce**(** **(**a**,** b**)** **->** a**.**add**(**b**)** **)**

**.**addSink**(...);**

#### ****With Periodic Watermarks****

AssignerWithPeriodicWatermarks assigns timestamps and generates watermarks periodically (possibly depending on the stream elements, or purely based on processing time).

Emit watermarks periodically.

#### ****With Punctuated Watermarks****

Emit watermarks based on some property of the incoming records, e.g. whenever a special element is encountered in the stream.

To generate watermarks whenever a certain event indicates that a new watermark might be generated, useAssignerWithPunctuatedWatermarks.

The checkAndGetNextWatermark(...) method is passed the timestamp that was assigned in the extractTimestamp(...) method, and can decide whether it wants to generate a watermark. Whenever the checkAndGetNextWatermark(...) method returns a non-null watermark, and that watermark is larger than the latest previous watermark, that new watermark will be emitted.

**public** **class** **PunctuatedAssigner** **extends**

AssignerWithPunctuatedWatermarks**<**MyEvent**>** **{**

@Override // first call

**public** **long** **extractTimestamp(**MyEvent element**,** **long** previousElementTimestamp**)** **{**

**return** element**.**getCreationTime**();**

**}**

@Override // second

**public** Watermark **checkAndGetNextWatermark(**MyEvent lastElement**,** **long** extractedTimestamp**)** **{**

**return** lastElement**.**hasWatermarkMarker**()** **?** **new** **Watermark(**extractedTimestamp**)** **:** **null;**

**}**

**}**

**Attention**:每个事件都可能触发产生一个watermark，每个watermark 导致下游的计算，因此大量的watermarks会导致性能降低。

### Timestamps per Kafka Partition

However, when consuming streams from Kafka, multiple partitions often get consumed in parallel, interleaving the events from the partitions and destroying the per-partition patterns (this is inherent in how Kafka’s consumer clients work).

Feature : Flink’s Kafka-partition-aware watermark generation.

FlinkKafkaConsumer09**<**MyType**>** kafkaSource **=** **new** FlinkKafkaConsumer09**<>(**"myTopic"**,** schema**,** props**);**

kafkaSource**.**assignTimestampsAndWatermarks**(new** AscendingTimestampExtractor**<**MyType**>()** **{**

@Override

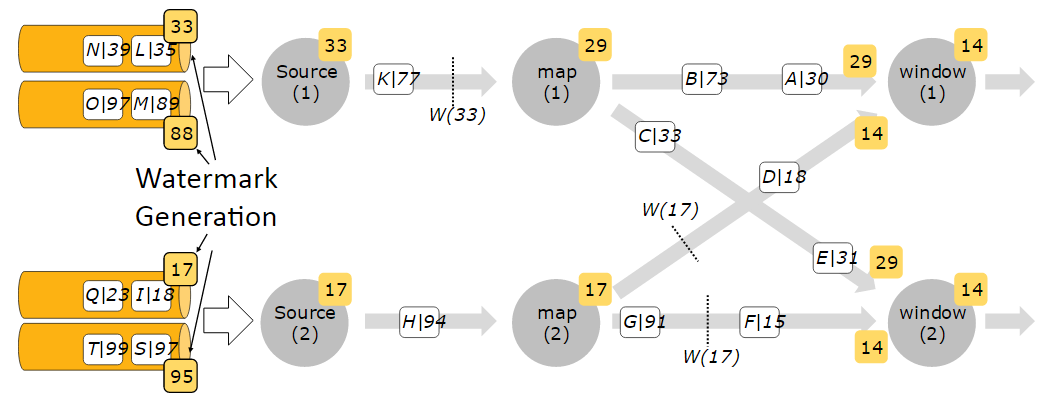
**public** **long** **extractAscendingTimestamp(**MyType element**)** **{**

**return** element**.**eventTimestamp**();**

**}**

**});**

DataStream**<**MyType**>** stream **=** env**.**addSource**(**kafkaSource**);**



## Controlling Latency

默认的,接收到的元素不是一个的经过网络进行传输，而是采用buffer缓冲，缓冲的大小是可以进行配置的。虽然采用缓冲可以优化程序提高吞吐能力，但是无疑将会增加延迟，尤其是当数据源来的很缓慢的时候，为了控制延迟，可以采用env.setBufferTimeout(timeoutMillis)方法，即是数据量没有达到缓冲的大小，仍然会被发送出去。默认值是100ms。

LocalStreamEnvironment env **=** StreamExecutionEnvironment**.**createLocalEnvironment**();**

env**.**setBufferTimeout**(**timeoutMillis**);**

env**.**genereateSequence**(**1**,**10**).**map**(new** **MyMapper()).**setBufferTimeout**(**timeoutMillis**);**

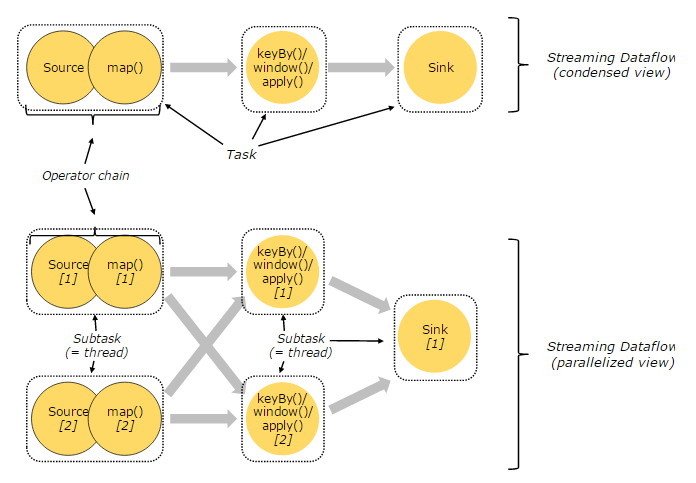
1. 为了达到最大化的吞吐量，可以将缓冲超时时间设置为-1，这样将会关闭超时，只有当缓冲满的时候才会flush。
2. 为了达到最小延迟，设置超时时间接近于0（5ms or 10 ms）,但是不能设置为0，因为将会导致性能服务器急剧下降。

# Distributed Runtime Environment

## Tasks and Operator Chains

Flink 将子任务操作chain在一起，形成task。每一个Task由一个Thread执行。

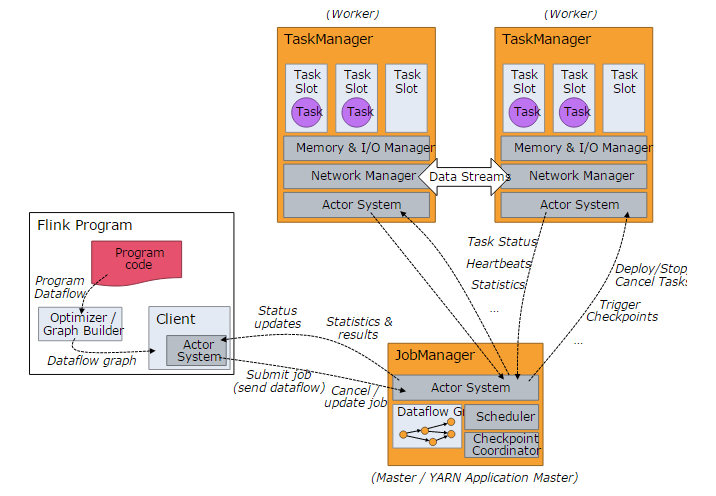
好处：减少线程切换所需要的overhead以及buffering,提升处理能力，降低延迟,chain操作也可以进行配置。



## Job Managers, Task Managers

运行时组件：

* **Job Managers** ( *masters*) 协调分布任务执行. 比如调度task, checkpoints, 失败重试等。至少一个**Job Manager**，支持HA( One leader, and the others are standby)
* **Task Managers** (*workers*) 执行*tasks* 数据流(更具体点执行subtasks), 缓存以及交换数据。至少一个Task Manager.。



## Task Slots and Resources

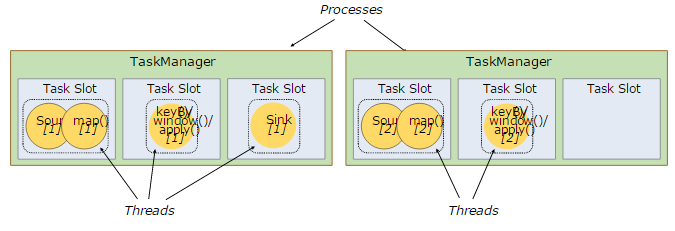
Worker是一个JVM进程，并且在每个单独的线程中尽可能多的处理subtask.

为了控制worker中执行的task的数量，所以worker就有task slots角色。

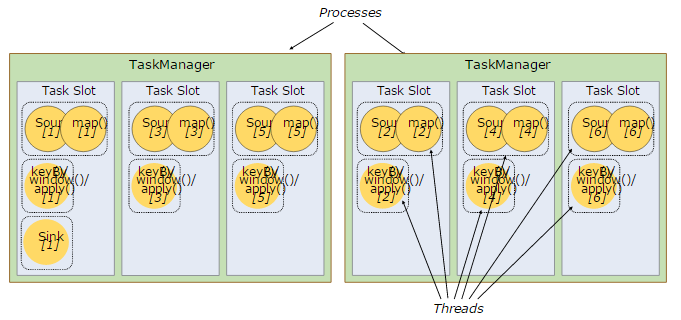
每个task slot代表了一个拥有固定资源子集的TaskManager。Task slot将会致力于将内存资源平均分配给每个slot.。Slot the resources 意味着subtask将不会参与其他job的subtasks的内存的竞争，而是使用固定的预留资源。这里CPU资源是不独立的。

同一个程序中资源隔离与资源共享模型：

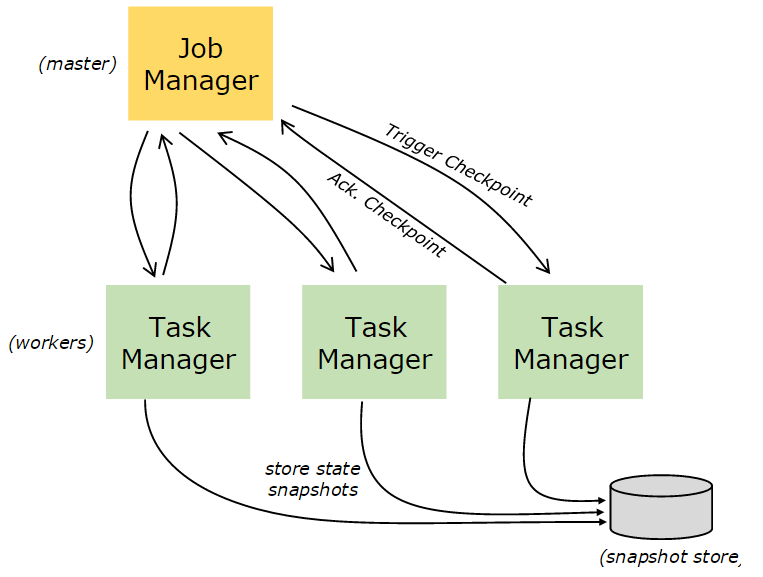
不同slot之间隔离资源



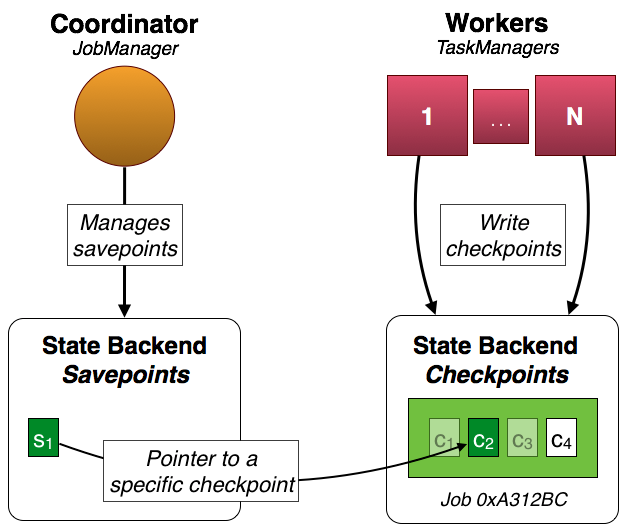
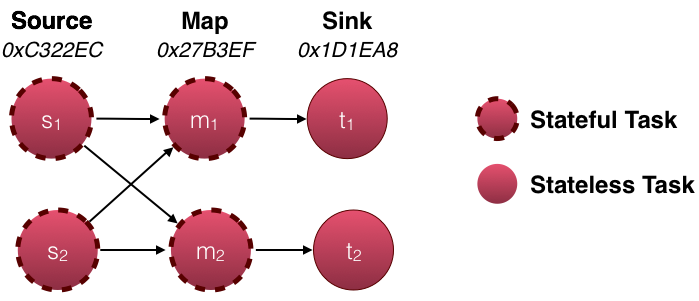
同一个slot中共享资源



## State Backends



# Savepoints

**DataStream<String>** stream **=** env**.**

*// Stateful source (e.g. Kafka) with ID*

**.**addSource**(new** **StatefulSource())**

**.**uid**(**"source-id"**)**

**.**shuffle**()**

*// The stateful mapper with ID*

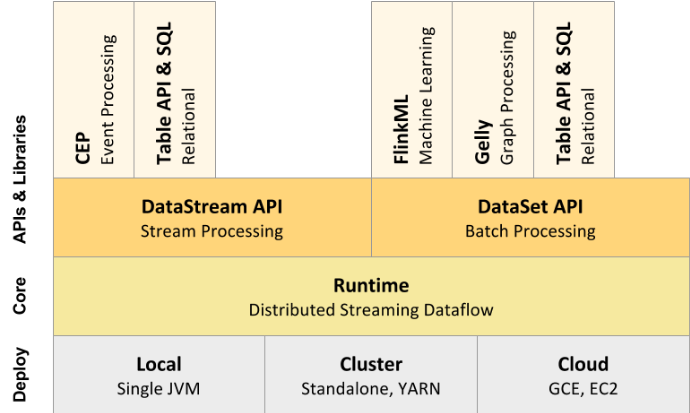
**.**map**(new** **StatefulMapper())**

**.**uid**(**"mapper-id"**)**

*// Stateless sink (no specific ID required)*

stream**.**print**()**

# Component Stack

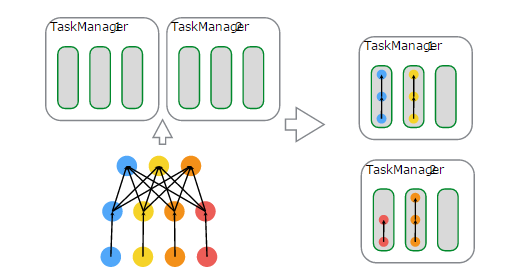


# Fault Tolerance Data Streaming

# the data stream source (such as message queue or broker) needs to be able to rewind the stream to a defined recent point. [Apache Kafka](http://kafka.apache.org/) has this ability and Flink’s connector to Kafka exploits this ability.

# how to generate Event Time Watermarks

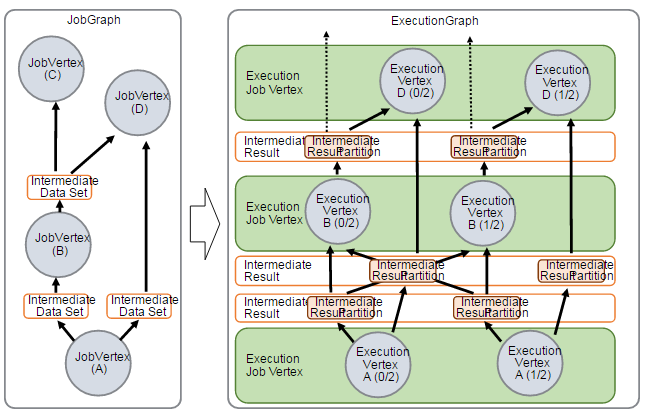
# Jobs and Scheduling



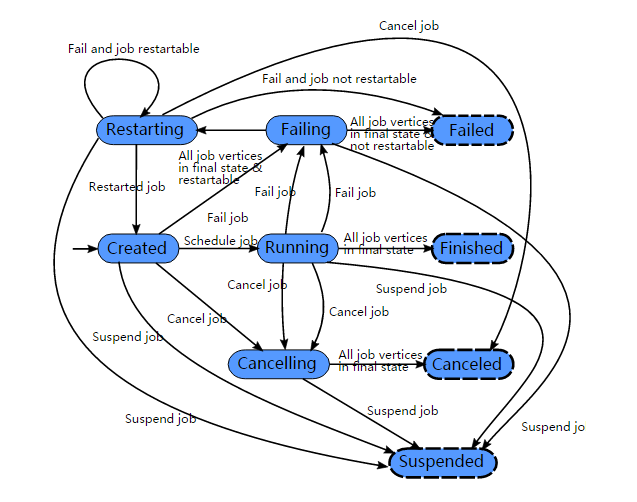
Get JobGraphs through separate compilation processes，uses an optimizer to determine the optimal plan for the program

Consider a program with a data source, a MapFunction, and a ReduceFunction. The source and MapFunction are executed with a parallelism of 4, while the ReduceFunction is executed with a parallism of 3. A pipeline consists of the sequence Source - Map - Reduce. On a cluster with 2 TaskManagers with 3 slots each, the program will be executed as described above.

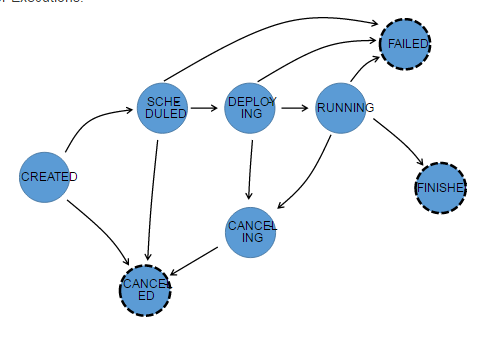
JobManager keeps track of distributed tasks, decides when to schedule the next task (or set of tasks), and reacts to finished tasks or execution failures.



**每个job可能经历的状态**



**每个task的可能经历的状态**



**延迟控制**

默认情况下，数据并不是一个接着一个在网络上传输的（这会导致不必要的网络流量），而是被缓冲的（buffered）。缓冲（实际上是机器之间的传输） 的大小可以在 Flink 配置文件中设置。虽然这种方法有利于优化吞吐量，但当输入的数据流不够快时，它可能会导致延迟问题时。要控制吞吐量和延迟，你可以在 StreamExecutionEnvironment 上使用env.setBufferTimeout(timeoutMillis)（或者单独的 operator 上）设置等待缓冲区被填满的最长等待时间。超过了这个时间，即时缓冲区还没有满也会被自动发送出去。默认的超时时间是 100 ms。

用法:

LocalStreamEnvironment env **=** StreamExecutionEnvironment**.**createLocalEnvironment**();**

env**.**setBufferTimeout**(**timeoutMillis**);**

env**.**genereateSequence**(**1**,**10**).**map**(new** **MyMapper()).**setBufferTimeout**(**timeoutMillis**);**

为了最大化吞吐量，可以设置 setBufferTimeout(-1)，这会移除超时等待时间而缓冲区只有被填满后才会被发送出去。为了最小化延时，可以设置一个接近 0 的超时时间（如 5 或 10 毫秒）。建议避免缓冲超时时间为 0 ，因为这会降低服务性能。

# [Task Lifecycle](https://ci.apache.org/projects/flink/flink-docs-release-1.3/internals/task_lifecycle.html)

TASK::setInitialState

TASK::invoke

create basic utils (config, etc) and load the chain of operators

setup-operators

task-specific-init

initialize-operator-states

open-operators

run

close-operators

dispose-operators

task-specific-cleanup

common-cleanup

As shown above, after recovering the tas

# Flink DataSet API Programming Guide

## Example Program

**public** **class** **WordCountExample** **{**

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

**final** ExecutionEnvironment env **=** ExecutionEnvironment**.**getExecutionEnvironment**();**

DataSet**<**String**>** text **=** env**.**fromElements**(**

"Who's there?"**,**

"I think I hear them. Stand, ho! Who's there?"**);**

DataSet**<**Tuple2**<**String**,** Integer**>>** wordCounts **=** text

**.**flatMap**(new** **LineSplitter())**

**.**groupBy**(**0**)**

**.**sum**(**1**);**

wordCounts**.**print**();**

**}**

**public** **static** **class** **LineSplitter** **implements** FlatMapFunction**<**String**,** Tuple2**<**String**,** Integer**>>** **{**

@Override

**public** **void** **flatMap(**String line**,** Collector**<**Tuple2**<**String**,** Integer**>>** out**)** **{**

**for** **(**String word **:** line**.**split**(**" "**))** **{**

out**.**collect**(new** Tuple2**<**String**,** Integer**>(**word**,** 1**));**

**}**

**}**

**}**

**}**

## DataSet Transformations

**Map,** **FlatMap,** **MapPartition,** **Filter,** **Reduce,** **ReduceGroup,** **Aggregate,** **Distinct,Join,** **CoGroup,** **Hash-Partition,** **Rebalance,** **Sort Partition**

**Project :** Selects a subset of fields from the tuples

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

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

**MinBy / MaxBy**

## Data Sources

* readTextFile(path) / TextInputFormat - Reads files line wise and returns them as Strings.
* readTextFileWithValue(path) / TextValueInputFormat - Reads files line wise and returns them as StringValues. StringValues are mutable strings.
* readCsvFile(path) / CsvInputFormat - Parses files of comma (or another char) delimited fields. Returns a DataSet of tuples or POJOs. Supports the basic java types and their Value counterparts as field types.
* readFileOfPrimitives(path, Class) / PrimitiveInputFormat - Parses files of new-line (or another char sequence) delimited primitive data types such as String or Integer.
* readFileOfPrimitives(path, delimiter, Class) / PrimitiveInputFormat - Parses files of new-line (or another char sequence) delimited primitive data types such as String or Integer using the given delimiter.
* readHadoopFile(FileInputFormat, Key, Value, path) / FileInputFormat - Creates a JobConf and reads file from the specified path with the specified FileInputFormat, Key class and Value class and returns them as Tuple2<Key, Value>.
* readSequenceFile(Key, Value, path) / SequenceFileInputFormat - Creates a JobConf and reads file from the specified path with type SequenceFileInputFormat, Key class and Value class and returns them as Tuple2<Key, Value>.

附：[Maven Flink项目构建：](https://ci.apache.org/projects/flink/flink-docs-release-1.2/quickstart/scala_api_quickstart.html#create-project-1)

Java

mvn archetype:generate -DarchetypeGroupId=org.apache.flink -DarchetypeArtifactId=flink-quickstart-java -DarchetypeVersion=1.2.0

-DgroupId=wiki-edits -DartifactId=wiki-edits -Dversion=0.1 -Dpackage=wikiedits -DinteractiveMode=false

Scala

mvn archetype:generate -DarchetypeGroupId=org.apache.flink -DarchetypeArtifactId=flink-quickstart-scala -DarchetypeVersion=1.2.0 -DgroupId=flink-scala -DartifactId=flink-scala -Dversion=0.1 -Dpackage=flink.scala -DinteractiveMode=false

**Flink Java Project**

mvn archetype:generate \

-DarchetypeGroupId**=**org.apache.flink \

-DarchetypeArtifactId**=**flink-quickstart-java \

-DarchetypeVersion**=**1.2.0 \

-DgroupId**=**org.apache.flink.quickstart \

-DartifactId**=**flink-java-project \

-Dversion**=**0.1 \

-Dpackage**=**org.apache.flink.quickstart \

-DinteractiveMode**=**false

**Flink Scala project**

mvn archetype:generate \

-DarchetypeGroupId**=**org.apache.flink \

-DarchetypeArtifactId**=**flink-quickstart-scala \

-DarchetypeVersion**=**1.2.0 \

-DgroupId**=**org.apache.flink.quickstart \

-DartifactId**=**flink-scala-project \

-Dversion**=**0.1 \

-Dpackage**=**org.apache.flink.quickstart \

-DinteractiveMode**=**false

**Savepoint 可恢复的**

* **启用检查点功能**：在每种情况下，我们都推荐在构建Flink程序的同时，把检查点功能打开，事实上在你的Flink程序中加上检查点只是需要增加几行代码而已。
* **可以重置的数据源**（即Apache Kafka、Amazon Kinesis，或者文件系统等）：数据源必须能按照你想要重新处理的点开始，重放数据。
* **所有的状态都通过Flink的管理状态接口保存**：所有具体的操作者的状态都必须保存在Flink的容错状态数据结构中，这让它可以按照某个之前的保存点位置被重置。
* **配置一个合适的状态后台**：Flink提供了不同的[状态后台](https://ci.apache.org/projects/flink/flink-docs-release-1.1/apis/streaming/state_backends.html)来将检查点和保存点持久化。默认地，保存点都保存在JobManager中，但你要为你的程序配置一个适当的后台状态程序，比如[RocksDB](https://ci.apache.org/projects/flink/flink-docs-release-1.1/apis/streaming/state_backends.html#the-rocksdbstatebackend)等。

**第一步：创建一个保存点**

首先，获得所有运行中的Flink任务的列表：

user$ flink list

------------Running/Restarting Jobs------------

10.10.2016 16:20:33 : job\_id : Sample Job (RUNNING)

（运行上面的命令时，你的真实任务ID会是一个包括字母和数字的字符串。）

然后，用相应的任务ID创建一个保存点：

user$ flink savepoint job\_id

现在你的保存点就已经可用了。

如果你准备马上根据你的保存点来重新启动任务，你通常会想要把现在正在运行的任务先停掉。你已经有了相应任务的ID，那把它停掉只要几秒钟就够了：

user$ flink cancel job\_id

**第二步：从一个保存点开始启动任务**

当你更新完程序之后，就可以从你的保存点开始启动任务了。

user$ flink run -d -s hdfs://savepoints/1 directory/your-updated-application.jar

如果你想在一个示例程序中自己重做这些步骤，我们推荐你看看一篇[之前的博客文章](http://data-artisans.com/robust-stream-processing-flink-walkthrough/#savepoints)，我们在那里讲了怎么做这件事。

#### 如果我想升级我的程序，该怎样做？

如果你想从一个保存点开始启动一个修改过的程序，有几件事是要考虑的。我们可以区别下面这两种情况：

1. 改变一个用户定义的函数的逻辑，比如MapFunction；
2. 改变一个程序的架构，也就是增加或减少操作者等；

第一种情况很简单，不需要什么特别的准备。你可以按你的需要去修改函数代码。不过，如果你用一个修改了的架构从保存点开始启动程序，那么为了能够恢复操作者的状态，Flink必须能够将保存点程序的操作者与使用了新架构的新程序的操作者对应起来。

在这种情况下，你就要手动地将操作者ID分配给最初的和更新了的程序。因为如果没有操作者ID的话，是没办法修改程序的架构的。所以最佳实践经验就要求一定要分配操作者ID。

下面的代码段显示了如何为操作者们分配ID。

DataStream stream = env.

// Stateful source (e.g. Kafka) with ID

.addSource(new StatefulSource())

.uid(“source-id”)

.shuffle()

// The stateful mapper with ID

.map(new StatefulMapper())

.uid(“mapper-id”)

// Stateless sink (no specific ID required)

stream.print()

请查阅[文档](https://ci.apache.org/projects/flink/flink-docs-release-1.1/apis/streaming/savepoints.html#changes-to-your-program)，了解更多关于升级程序和保存点的细节。

## 关于保存点的最佳实践

要更好的利用上文中描述的Flink的重新处理功能，你应该经常触发，生成新的保存点。我们建议要根据某些时刻表（比如每天一次，每周一次，等等）自动地生成保存点，而且每当你关闭某个任务或发布程序的新版本时，也最好先生成保存点。

依据你想用Flink做的事件不同，生成保存点的最佳方法也会不同，但总的来说，在构建你的程序时你应该花些时间考虑如何使用这些保存点。

## 这些东西是怎么工作的呢？

保存点事实上只是检查点的一个延伸，这就是Flink的容错机制。如果开启了检查点功能，Flink就会周期性地为所有的操作者状态生成一个一致的检查点。在[文档](https://ci.apache.org/projects/flink/flink-docs-release-1.1/internals/stream_checkpointing.html)中详细的描述了检查点的细节，如果你是个Flink新手，花些时间去读读是非常值得的。

你可能会以为要生成一个一致的检查点，就得暂停数据处理，因为Flink必须要等着，直到所有没处理完的记录全被处理掉了，然后做个镜像，镜像生成之后再回去继续处理数据。事实并非如此！Flink是持续处理数据的，即使在生成检查点的时候也是这样。文档中的[“Barriers”一节](https://ci.apache.org/projects/flink/flink-docs-release-1.1/internals/stream_checkpointing.html#barriers)讲了实现这个功能的原理。

两者之间的关键区别：检查点是基于某些规定的时间间隔自动生成的，而保存点是由用户显式地触发生成的，而且不会象检查点那样[过了一定的时间之后就会被删掉](http://data-artisans.com/how-apache-flink-enables-new-streaming-applications/)。

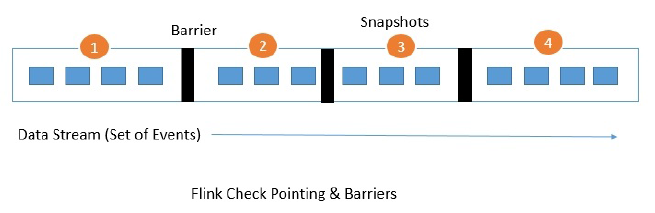
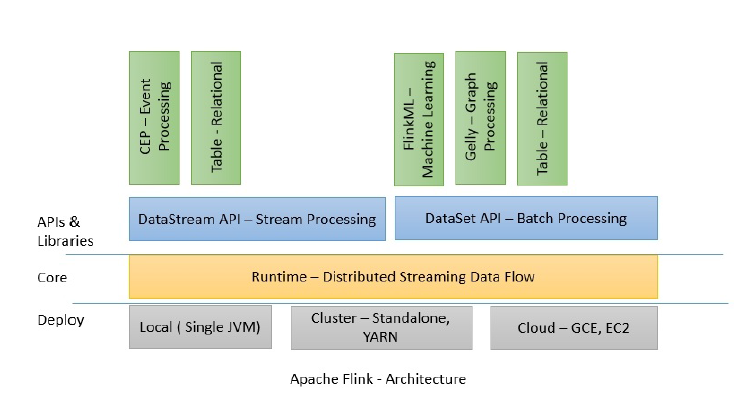
## 总结

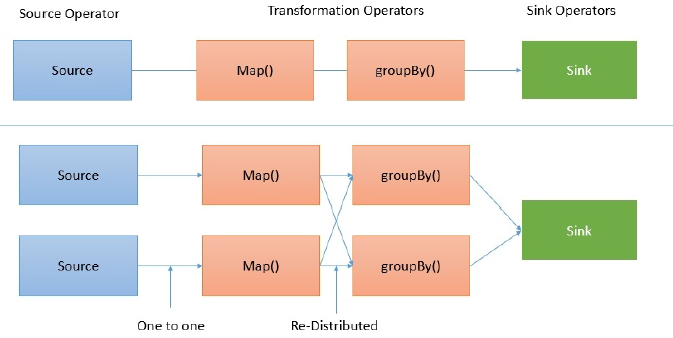
我们讨论了Apache Flink的保存点和数据重处理功能，因为我们相信这就是Flink与开源世界中其它流处理器之间的重要区别之一。而且最重要的，在容错的Flink程序中获得重处理功能几乎是不需要任何代价的，只需要很少的改动。

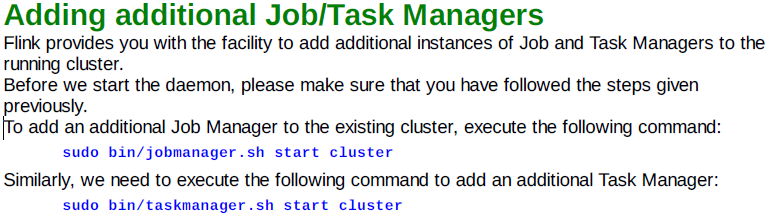
Flink社区现在还在积极地工作着，要把保存点功能做得更好，包括在改变并发度的情况下保存状态的解决方案等。有些相应的功能（比如[Flink-3755](https://issues.apache.org/jira/browse/FLINK-3755)）已经发布到主分支上了，而且会被包含到下一个小版本Flink 1.2.0中。

所以，当你需要把程序多部署一份，或者上个新版本，或者要做A/B测试，或者要让多个程序从同一个点开始处理数据时，你可以这么做了，而且不会丢失那些宝贵的状态数据。

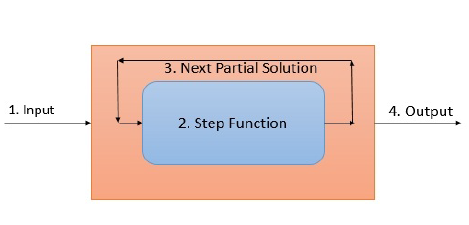
当有真实的需求时，流处理基于实时的特性不应该阻挡你把时间调回过去的动作。







**Iterator operator**



Delta iterator works on updating the solution set rather than fully re-computing it every iteration.

Delta iterator 采用更新结果集，而不是完全的重新计算一遍。

