



Flink入门到项目实战



文档历史

|  |  |  |  |
| --- | --- | --- | --- |
| 日期 | 版本 | 作者 | 描述说明 |
| 2020-02-12 | 1.0 | 吴波 | 新建 |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
| 文档最后一次修改时间 | | 2020-04-28 16:34:00 | |

目录

1 Flink概述 2

1.1 简介 2

1.2 快速入门 2

2 Flink API 4

2.1 分层API 4

2.2 基本API 4

2.2.1 DataSet API 4

2.2.2 DataStream API 7

2.2.3 Table &SQL API 8

3 常用特性 9

3.1 累加器 9

3.2 广播变量 9

3.3 分布式缓存 10

3.4 DataStream Kafka Source 11

3.5 Event Time与WaterMark 11

3.6 窗口函数（Window Function） 12

3.7 触发器（Trigger） 13

3.8 侧输出（side output） 13

3.9 异步IO 14

3.10 不同数据流join 14

3.11 DataStream Sink 15

4 高级应用 16

4.1 ProcessFunction 16

4.1.1 概念 16

4.1.2 onTimer 16

4.1.3 CoProcessFunctions双流操作 17

4.1.4 KeyedProcessFunction 18

4.1.5 TimerService 18

4.2 状态管理 18

4.2.1 作用 18

4.2.2 基本介绍 19

4.2.3 案例介绍 19

4.2.4 State类型 20

4.2.4.1 Operator State 20

4.2.4.2 Keyed State 21

4.2.5 状态后端(State Backends) 22

4.2.6 扩展有状态的operators 23

4.2.6.1 keyed state 23

4.2.6.2 list state 23

4.2.6.3 union list state 24

4.2.6.4 broadcast state 25

5 项目案例 26

5.1 案例1-实时数据 26

5.1.1 介绍 26

5.1.2 架构 26

5.1.3 代码 26

5.2 案例2-离线数据 27

5.2.1 介绍 27

5.2.2 结构 27

5.2.3 代码 27

6 Flink on Yarn部署和任务提交操作 28

6.1 环境变量配置 28

6.2 任务提交方式 28

6.2.1 同时启动Yarn application和Flink task 28

6.2.2 先启动Yarn application，再在指定的application内运行Flink task 28

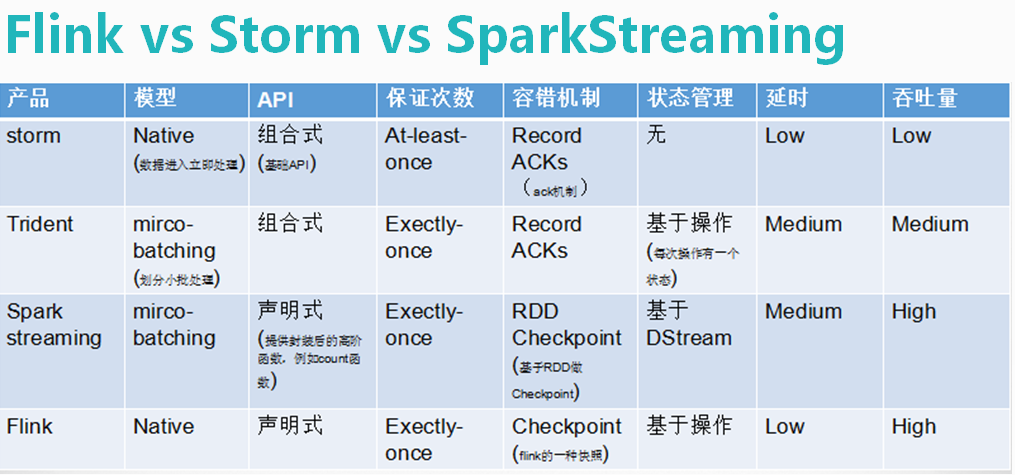
7 异常问题 29

7.1 env.registerCachedFile(cfgfilePath，cacheName) 29

# Flink概述

## 简介

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，提供支持流处理和批处理两种类型应用的功能。本文档主要对Flink中基本API如：DataSet、DataStream、Table、Sql和常用特性如：Time&Window、窗口函数、Watermark、触发器、分布式缓存、异步IO、侧输出、广播和高级应用如：ProcessFunction、状态管理等知识点进行整理



代码地址：<http://172.16.2.18:8690/svn/cloudil-java/trunk/cloudil-p3/flink/flink-local-train>

学习参考：<https://github.com/perkinls/flink-local-train>

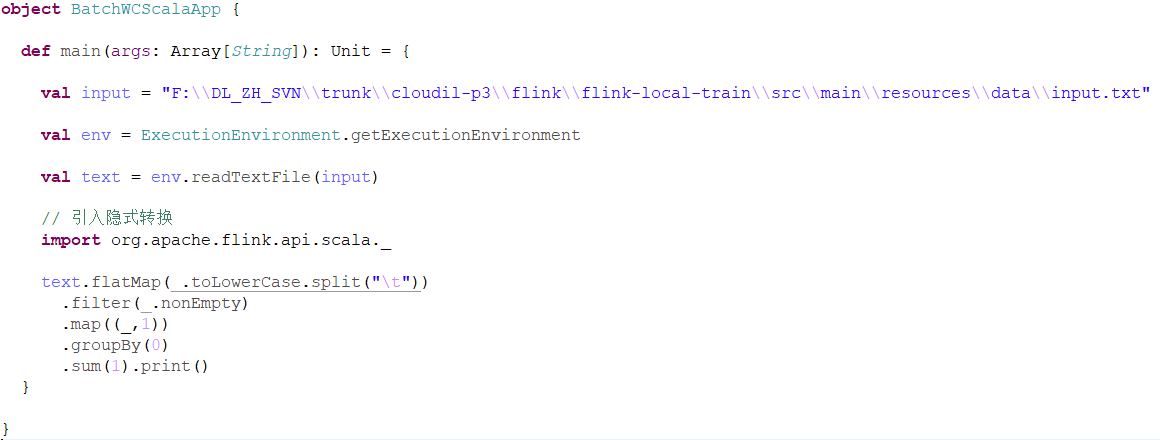
Apache Flink在滴滴的应用与实践

https://blog.csdn.net/w397090770/article/details/105743237/

## 快速入门

在数据处理领域，WordCount就是HelloWorld。Flink自带WordCount例子，它通过socket读取text数据，并且统计每个单词出现的次数。

WordCount（批处理）案例：[BatchWCScalaApp.scala](http://172.16.2.18:8690/svn/cloudil-java/trunk/cloudil-p3/flink/flink-local-train/src/main/scala/com/dinglicom/scala/demo/quickstart/BatchWCScalaApp.scala)



输出



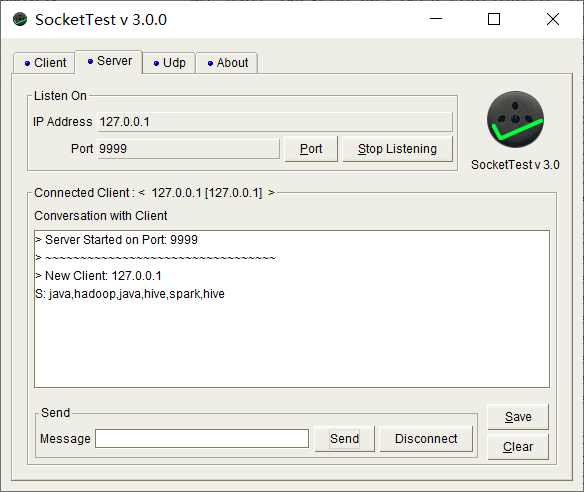
WordCount（流处理，配合Socket\_Test测试）案例：[StreamingWCScalaApp.scala](http://172.16.2.18:8690/svn/cloudil-java/trunk/cloudil-p3/flink/flink-local-train/src/main/scala/com/dinglicom/scala/demo/quickstart/StreamingWCScalaApp.scala)

Socket工具：<http://172.16.2.18:8690/svn/cloudil-java/trunk/cloudil-p3/flink/Scoket_Test>

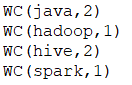
启动socket服务

java -jar F:\DL\_ZH\_SVN\trunk\cloudil-p3\flink\Scoket\_Test\SocketTest.jar





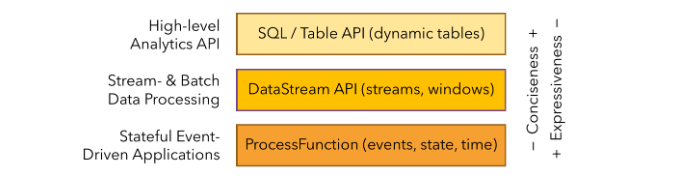
输出



# Flink API

## 分层API

Flink 根据抽象程度分层，提供了三种不同的 API。每一种 API 在简洁性和表达力上有着不同的侧重，并且针对不同的应用场景。



## 基本API



以上为Flink的运行模型（和Spark基本一致），Flink的程序主要由三部分构成，分别为Source、Transformation、Sink。DataSource主要负责数据的读取，Transformation主要负责对属于的转换操作，Sink负责最终数据的输出。

### DataSet API

DataSet API，对静态数据进行批处理操作，将静态数据抽象成分布式的数据集，用户可以方便地使用Flink提供的各种操作符对分布式数据集进行处理。Flink先将接入数据（如可以通过读取文件或从本地集合）来创建转换成DataSet数据集，并行分布在集群的每个节点上；然后将DataSet数据集进行各种转换操作(map，filter，union，group等)，最后通过DataSink操作将结果数据集输出到外部系统。

Flink中每一个的DataSet程序大致包含以下流程：

- step 1 : 获得一个执行环境（ExecutionEnvironment）

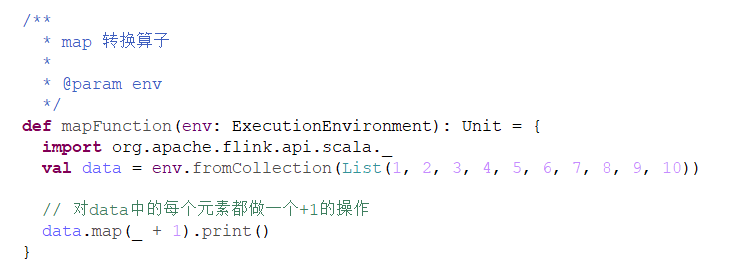
- step 2 : 加载/创建初始数据 （Source）

- step 3 : 指定转换算子操作数据（Transformation）

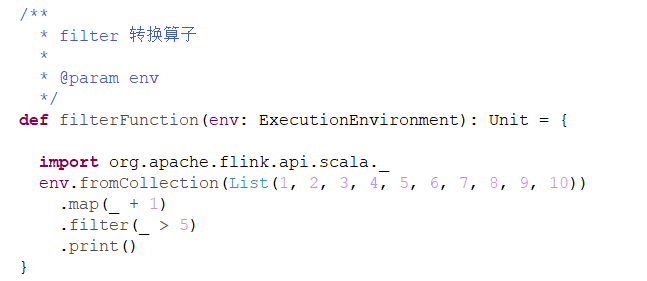
- step 4 : 指定存放结果位置（Sink）

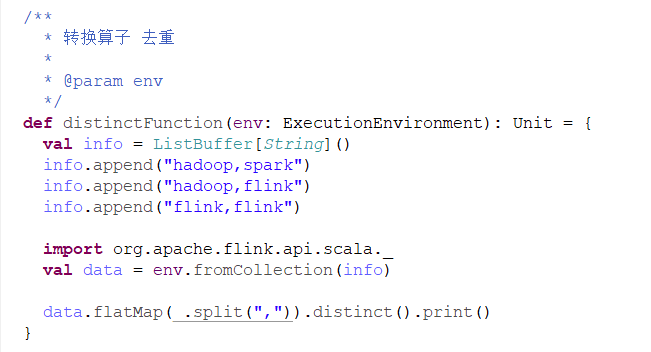
DataSet Source案例：[DataSetDataSourceApp.scala](http://172.16.2.18:8690/svn/cloudil-java/trunk/cloudil-p3/flink/flink-local-train/src/main/scala/com/dinglicom/scala/demo/dataset/DataSetDataSourceApp.scala)

DataSet Transformation案例：[DataSetTransformationApp.scala](http://172.16.2.18:8690/svn/cloudil-java/trunk/cloudil-p3/flink/flink-local-train/src/main/scala/com/dinglicom/scala/demo/dataset/DataSetTransformationApp.scala)











DataSet Sink案例：[DataSetSinkApp.scala](http://172.16.2.18:8690/svn/cloudil-java/trunk/cloudil-p3/flink/flink-local-train/src/main/scala/com/dinglicom/scala/demo/dataset/DataSetSinkApp.scala)

### DataStream API

DataStreamAPI，是Flink API中最核心的数据结构，对数据流进行流处理操作，将流式的数据抽象成分布式的数据流，用户可以方便地对分布式数据流进行各种操作。Flink先将流式数据（如可以通过消息队列，套接字流，文件等）来创建DataStream，并行分布在集群的每个节点上；然后对DataStream数据流进行转换（filter,join, update state, windows, aggregat等），最后通过DataSink操作将DataStream输出到外部文件或存储系统中。

F link中每一个DataStream程序大致包含以下流程：

- step 1 : 获得一个执行环境（StreamExecutionEnvironment）

- step 2 : 加载/创建初始数据 （Source）

- step 3 : 指定转换算子操作数据（Transformation）

- step 4 : 指定存放结果位置（Sink）

- step 5 : 手动触发执行

注意：

因为flink是lazy加载的，所以必须调用execute方法，上面的代码才会执行。

在DataSet和DataStrean中transformation 都是懒执行，需要最后使用env.execute()触发执行或者使用 print(),count(),collect() 触发执行。

代码案例：Scala

DataStream Source案例：[DataStreamSourceApp.scala](http://172.16.2.18:8690/svn/cloudil-java/trunk/cloudil-p3/flink/flink-local-train/src/main/scala/com/dinglicom/scala/demo/datastream/DataStreamSourceApp.scala)

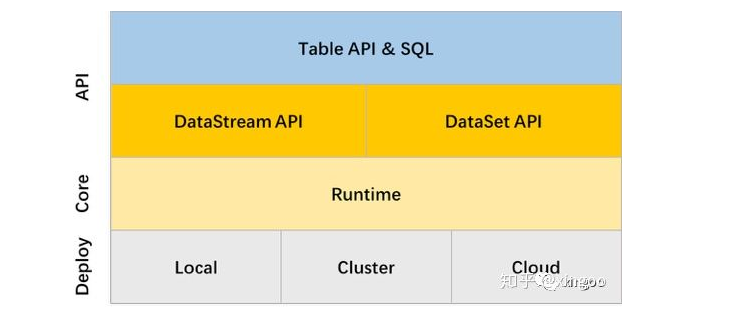
DataStream Transformation案例：[DataStreamTransformationApp.scala](http://172.16.2.18:8690/svn/cloudil-java/trunk/cloudil-p3/flink/flink-local-train/src/main/scala/com/dinglicom/scala/demo/datastream/DataStreamTransformationApp.scala)

DataStream Sink案例：[DataStreamSinkToMysqlApp.scala](http://172.16.2.18:8690/svn/cloudil-java/trunk/cloudil-p3/flink/flink-local-train/src/main/scala/com/dinglicom/scala/demo/datastream/DataStreamSinkToMysqlApp.scala)

### Table &SQL API

Flink Table & SQL API是在DataStream和DataSet之上封装的一层高级API。由于DataStream和DataSet有各自的API，开发起来又有些困难，如果只是应对一些相对通用的需求会有点麻烦。而Flink Table & SQL API，通过关系型的API简化了内部的复杂实现。

通过SQL开发人员可以只关注业务逻辑，学习成本低，容易理解，而且内置了很多的优化规则，可以简化开发复杂度，通过SQL还能在高层应用上实现真正的批流一体。



Flink中每一个Table & Sql程序大致包含以下流程：

- step 1 : 获得一个执行环境（ExecutionEnvironment/StreamExecutionEnvironment）

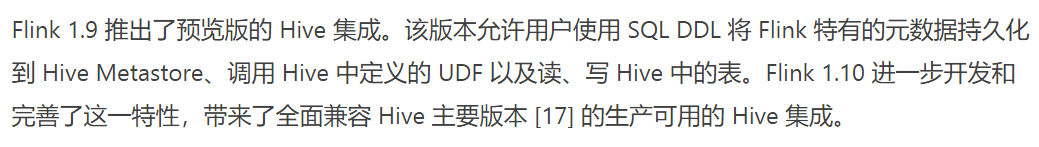
- step 2 : 根据执行环境获取Table & Sql运行环境（TableEnvironment）

- step 3 : 注册输入表（Input table）

- step 4 : 执行Table & Sql查询

- step 5 : 输出表（Output table）结果发送到外部系统

代码案例：[StreamTableApiApp.scala](http://172.16.2.18:8690/svn/cloudil-java/trunk/cloudil-p3/flink/flink-local-train/src/main/scala/com/dinglicom/scala/demo/table/StreamTableApiApp.scala)



# 常用特性

## 累加器

Flink中累加器(Accumulators)是非常的简单，通过一个add操作累加最终的结果，在job执行后可以获取最终结果。

Flink最直接的累加器是一个计数器(counter)：你可以使用Accumulator.add()方法对其进行累加。在作业结束时，Flink将合并所有部分结果并将最终结果发送给客户端。在调试过程中，或者你快速想要了解有关数据的更多信息，累加器很有用。

目前Flink拥有以下内置累加器。它们中的每一个都实现了累加器接口：

(1) IntCounter, LongCounter 以及 DoubleCounter: 可参考案例中的计数器。

(2) Histogram：为离散数据的直方图(A histogram implementation for a discrete number of bins.)。内部它只是一个整数到整数的映射。你可以用它来计算值的分布，例如 单词计数程序的每行单词分配。

Flink中累加器的开发步骤大致如下：

- step 1 : 在你要使用的用户自定义转换函数中创建一个累加器(accumulator)对象

- step 2 : 注册累加器(accumulator)对象，通常在rich函数的open()方法中注册。在这里你也可以自定义累加器的名字。

- step 3 : 算子函数中的任何位置使用累加器

- step 4 : 最后结果将存储在JobExecutionResult对象中，该对象从执行环境的execute()方法返回(当前仅当执行等待作业完成时才起作用)

注意：

每个作业的所有累加器共享一个命名空间。因此，你可以在作业的不同算子函数中使用同一个累加器。Flink在内部合并所有具有相同名称的累加器。

目前累加器的结果只有在整个工作结束之后才可以使用。我们还计划在下一次迭代中可以使用前一次迭代的结果。你可以使用聚合器来计算每次迭代的统计信息，并基于此类统计信息来终止迭代。

代码案例：CounterAndAccumulator.scala

## 广播变量

可以理解为是一个公共的共享变量，就是将数据广播到具体的taskmanager上。

1、dataStream当中的广播分区

将数据广播给所有的分区，数据可能会被重复处理，一般用于某些公共的配置信息读取，不会涉及到更改的数据

2、dataSet当中的广播变量

广播变量允许编程人员在每台机器上保持1个只读的缓存变量，而不是传送变量的副本给tasks

用法如下：

- step 1 : 初始化数据 val toBroadcast = env.fromElements(1, 2, 3)

- step 2 : 广播数据 .withBroadcastSet(toBroadcast, "broadcastSetName")

- step 3 : 获取数据 val broadcastSet = getRuntimeContext().getBroadcastVariable("broadcastSetName")

注意：

1：广播出去的变量存在于每个节点的内存中，所以这个数据集不能太大。因为广播出去的数据，会常驻内存，除非程序执行结束

2：广播变量在初始化广播出去以后不支持修改，这样才能保证每个节点的数据都是一致的。

代码案例：DataSetBroadCast.scala

## 分布式缓存

Flink提供了一个分布式缓存，类似于hadoop，可以使用户在并行函数中很方便的读取本地文件，并把它放在taskmanager节点中，防止task重复拉取。此缓存的工作机制如下：程序注册一个文件或者目录(本地或者远程文件系统，例如hdfs或者s3)，通过ExecutionEnvironment注册缓存文件并为它起一个名称。

当程序执行，Flink自动将文件或者目录复制到所有taskmanager节点的本地文件系统，仅会执行一次。用户可以通过这个指定的名称查找文件或者目录，然后从taskmanager节点的本地文件系统访问它。其实分布式缓存就相当于spark的广播,把一个变量广播到所有的executor上,也可以看做是Flink的广播流,只不过这里广播的是一个文件。

Flink中分布式缓存开发步骤大致如下：

- step 1 : 注册一个文件,可以使用hdfs上的文件 也可以是本地文件进行测试

- step 2 : 通过RichFunction函数使用文件

注意：

在用户函数中访问缓存文件或者目录。这个函数必须继承RichFunction,因为它需要使用RuntimeContext读取数据。

代码案例：[DistributedCacheApp.scala](http://172.16.2.18:8690/svn/cloudil-java/trunk/cloudil-p3/flink/flink-local-train/src/main/scala/com/dinglicom/scala/demo/dataset/DistributedCacheApp.scala)

## DataStream Kafka Source

Flink提供了特殊的Kafka connector，用于从Kafka主题读写数据。 Flink Kafka Consumer与Flink的检查点(checkpoint)机制集成在一起，以提供有且仅有一次的语义。为此，Flink不仅仅依赖于Kafka的消费者群体偏移量跟踪，还内部跟踪和检查这些偏移量。

在真实的生产环境上，我们都需要保证系统的高可用。即需要保证系统的各个组件不能出现问题，或者提供一系列的容错机制。启用Flink的检查点后，Flink Kafka Consumer将在一个topic消费记录的时候，并以一致的方式定期记录Kafka偏移量和其它操作者的操作到检查点。万一作业失败，Flink将把流式程序恢复到最新检查点的状态，并从检查点中存储的偏移量开始重新使用Kafka的记录。



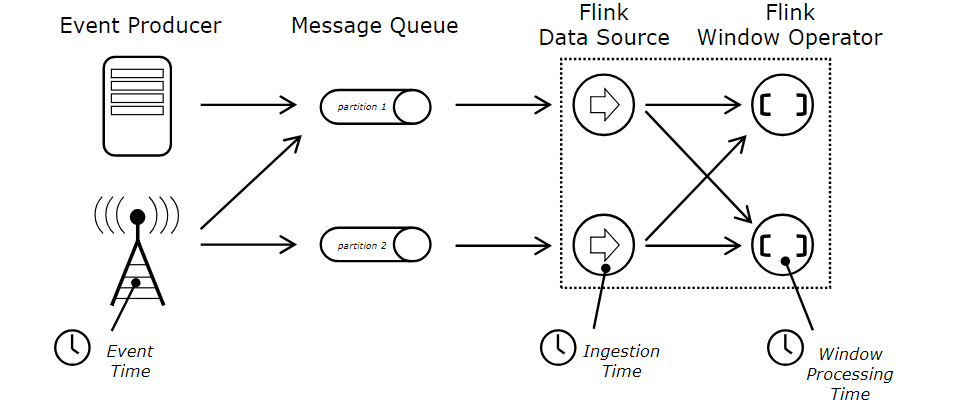
可参考文章：<https://blog.csdn.net/u013076044/article/details/102651473>

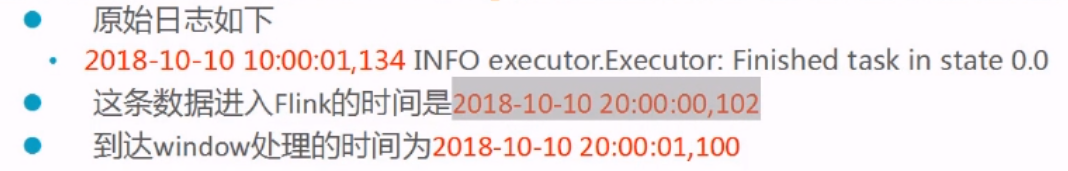
代码案例：[KafkaSourceApp.scala](http://172.16.2.18:8690/svn/cloudil-java/trunk/cloudil-p3/flink/flink-local-train/src/main/scala/com/dinglicom/scala/demo/datastream/source/KafkaSourceApp)

## Event Time与WaterMark

针对stream数据中的时间，可以分为以下三种：

* Event time：事件产生的时间，它通常由事件中的时间戳描述。
* Ingestion time：事件进入Flink的时间。
* Processing time：事件被处理时当前系统的时间。





Time例子分析：如果我们想要统计每分钟内接口调用失败的错误日志个数，使用哪个时间才有意义？（Event Time）

Watermark是一种衡量Event Time进展的机制，它是数据本身的一个隐藏属性。通常watermark是用于处理乱序事件的，而正确的处理乱序事件，通常用watermark机制结合window来实现。

我们知道，流处理从事件产生，到流经source，再到operator，中间是有一个过程和时间的。虽然大部分情况下，流到operator的数据都是按照事件产生的时间顺序来的，但是也不排除由于网络、背压等原因，导致乱序的产生（out-of-order或者说late element）。但是对于late element，我们又不能无限期的等下去，必须要有个机制来保证一个特定的时间后，必须触发window去进行计算了。这个特别的机制，就是watermark。

触发机制：

- 1.watermark时间>=window\_end\_time。

- 2.在【window\_start\_time,window\_end\_time】中有数据存在, 注意左闭右开区间。

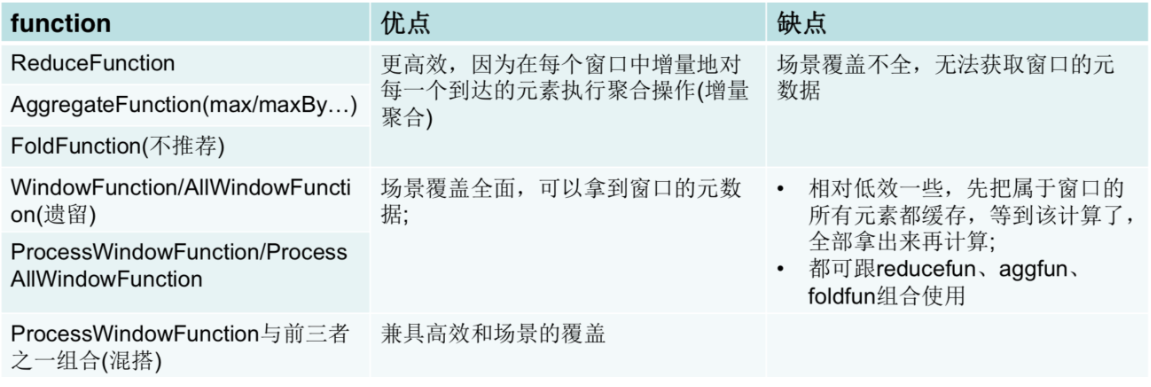
有关WaterMark详解请参考文章：[一文读懂WaterMark机制](https://blog.csdn.net/lmalds/article/details/52704170)

代码案例：[KafkaSourceWatermark.scala](http://172.16.2.18:8690/svn/cloudil-java/trunk/cloudil-p3/flink/flink-local-train/src/main/scala/com/dinglicom/scala/demo/datastream/watermark/KafkaSourceWatermark.scala)

## 窗口函数（Window Function）

窗口函数（Window Function）作用于窗口分配器之后，指定要在每个窗口上执行的计算。一旦系统确定某个窗口已准备好进行处理，就可以使用该窗口函数来处理每个（按key分组）窗口的元素。

窗口函数可以是ReduceFunction，AggregateFunction，FoldFunction或ProcessWindowFunction之一。前两个可以更有效地执行，因为Flink可以在每个窗口到达时逐步地聚合它们。ProcessWindowFunction为窗口中包含的所有元素以及该元素所属的窗口的其他元信息获取Iterable。使用ProcessWindowFunction进行窗口转换不能像其他情况一样有效地执行，因为Flink必须在调用函数之前会在内部缓冲窗口的所有元素。可以通过将ProcessWindowFunction与ReduceFunction，AggregateFunction或FoldFunction组合使用来获得窗口元素的增量聚合以及ProcessWindowFunction接收的其他窗口元数据，从而缓解这种情况。



关于窗口可参考：https://www.cnblogs.com/bjwu/p/10393146.html

关于窗口函数更多可参考:<http://www.lllpan.top/article/43>

代码案例：[TumblingWindowsProcessFunction.scala](http://172.16.2.18:8690/svn/cloudil-java/trunk/cloudil-p3/flink/flink-local-train/src/main/scala/com/dinglicom/scala/demo/datastream/windows)

## 触发器（Trigger）

触发器（Trigger）确定窗口何时准备好由窗口功能处理。每个WindowAssigner都带有一个默认触发器。如果默认触发器不适合您的需求，则可以使用trigger（...）指定自定义触发器。

Trigger触发器接口有五个方法允许trigger对不同的事件做出反应：

- onElement()进入窗口的每个元素都会调用该方法。

- onEventTime()事件时间timer触发的时候被调用。

- onProcessingTime()处理时间timer触发的时候会被调用。

- onMerge()有状态的触发器相关，并在它们相应的窗口合并时合并两个触发器的状态，例如使用会话窗口。

- clear()该方法主要是执行窗口的删除操作。

代码案例：[CustomTriggerApp.scala](http://172.16.2.18:8690/svn/cloudil-java/trunk/cloudil-p3/flink/flink-local-train/src/main/scala/com/dinglicom/scala/demo/datastream/trigger/CustomTriggerApp.scala)

## 侧输出（side output）

在Flink处理数据流时，我们经常会遇到这样的情况：在处理一个数据源时，往往需要将该源中的不同类型的数据做分割处理，如果使用 filter算子对数据源进行筛选分割的话，势必会造成数据流的多次复制，造成不必要的性能浪费；flink中的侧输出就是将数据流进行分割，而不对流进行复制的一种分流机制。flink的侧输出的另一个作用就是对延时迟到的数据进行处理，这样就可以不必丢弃迟到的数据。 此外，侧输出（side output）能有效的解决算子spilt不能进行连续分流的问题。

<https://issues.apache.org/jira/browse/FLINK-11084>

代码案例：[SideOutPutTest.scala](http://172.16.2.18:8690/svn/cloudil-java/trunk/cloudil-p3/flink/flink-local-train/src/main/scala/com/dinglicom/scala/demo/datastream/sideoutputs)

## 异步IO

Flink在有些流式应用的场合中，我们都会去与外部系统进行交互，比如连接数据库等。当我们需要向外部系统发送一个请求a的时候，我们需要等待它返回结果，这是同步的模式。考虑到吞吐量和延迟，我们可以并发的发送请求a,b,c，当哪个请求先回复的时候先处理哪个，这样可以在连续的请求中避免不必要的等待，这就是Async IO。它是由阿里巴巴捐献给Flink社区的，于1.2版本后引入。

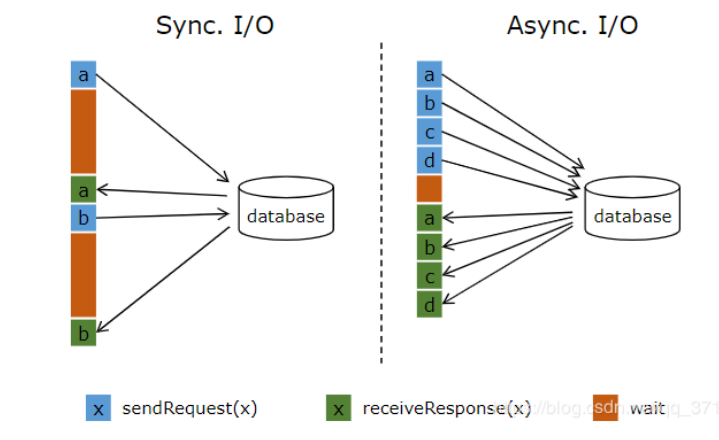
Flink中异步I/O的开发步骤大致如下：

- step 1 : 实现AsyncFunction，该函数实现了请求分发的功能。

- step 2 : Callback回调，该函数取回操作的结果，然后传递给ResultFuture。

- step 3 : 对DataStream使用异步IO操作。

异步I/O详细介绍或者更多相关文章可参考：<http://www.lllpan.top/article/45>



代码案例：[AsyncIOSideTableJoinMysql.scala](http://172.16.2.18:8690/svn/cloudil-java/trunk/cloudil-p3/flink/flink-local-train/src/main/scala/com/dinglicom/scala/demo/datastream/asyncio)

## 不同数据流join

window join将共享相同key并位于同一窗口中的两个流的元素联接在一起。可以使用窗口分配器定义这些窗口，并根据两个流中的元素对其进行评估。然后将双方的元素传递到用户定义的JoinFunction或FlatJoinFunction，在此用户可以发出满足联接条件的结果。

Interval Join使用公共key连接两个流（现在将它们分别称为A和B）的元素，并且流B的元素具有与流A的元素时间戳相对时间间隔的时间戳。

并且流B的元素具有与流A的元素时间戳**相对时间间隔的时间戳**。

* Tumbling Window Join
* Sliding Window Join
* Session Window Join
* Interval Join

代码案例：[TumblingWindowJoin.scala](http://172.16.2.18:8690/svn/cloudil-java/trunk/cloudil-p3/flink/flink-local-train/src/main/scala/com/dinglicom/scala/demo/datastream/join)

## DataStream Sink

Sink是将数据源最终写入文件或者数据库或者其他中间件当中。 writeAsText()：将元素以字符串形式逐行写入，这些字符串通过调用每个元素的toString()方法来获取。print() / printToErr()：打印每个元素的toString()方法的值到标准输出或者标准错误输出流中。 同时Flink DataStream也支持自定义输出addSink【kafka、redis、Mysql、Hbase等】外部数据源。

内置的Connectors可参考官网：

<https://ci.apache.org/projects/flink/flink-docs-release-1.9/dev/connectors/>

针对于Mysql、Hbase、Redis可参考代码案例：[Kafka2Mysql.scala](http://172.16.2.18:8690/svn/cloudil-java/trunk/cloudil-p3/flink/flink-local-train/src/main/scala/com/dinglicom/scala/demo/datastream/sink)

# 高级应用

## ProcessFunction

### 概念

- 低阶API Process function，越底层越丰富复杂，提供了对状态的管理。

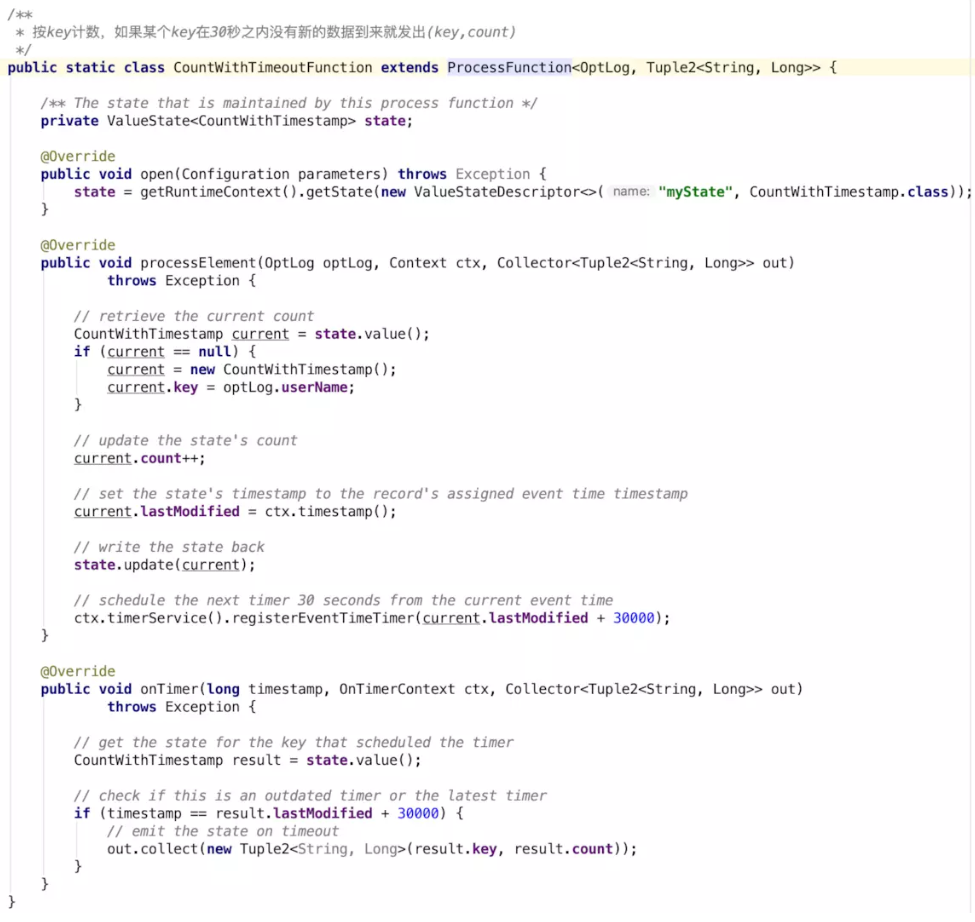
- 中阶API DataStream API，一般都是使用这个。

- 高阶API SQL/Table API，高度抽象。

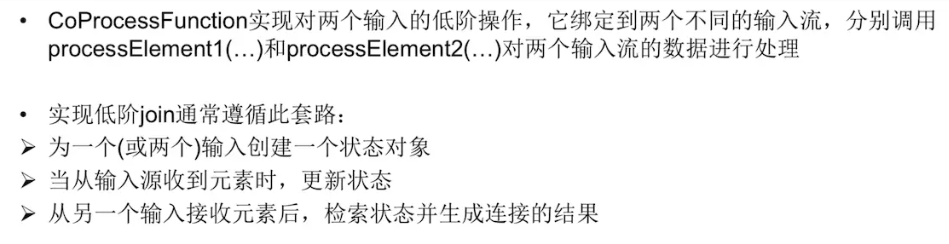


### onTimer

Process Function ,注册定时器，维护状态，定时到时触发调用onTimer个默认触发器。

****

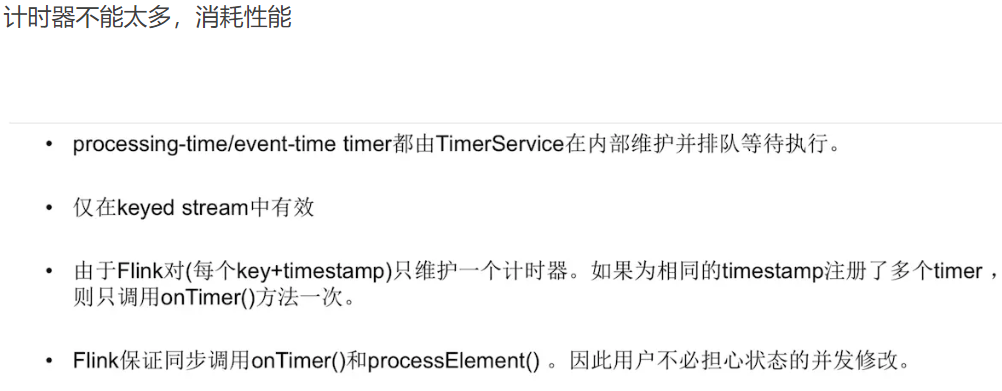
### CoProcessFunctions双流操作



### KeyedProcessFunction



### TimerService



## 状态管理

### 作用

- 增量计算：聚合操作、机器学习训练模型迭代运算时保存当前模型等等。

- 容错：Job故障重启、升级。

### 基本介绍

定义：某Task或者Operator在某一时刻的在内存中的状态。而checkpoint是对于这个中间结果进行的一次快照。

作用：State是可以被记录的，在失败的情况下可以恢复。Checkpoint则表示了一个Flink Job，在一个特定时刻的一份全局状态快照，即包含了一个job下所有task/operator某时刻的状态。

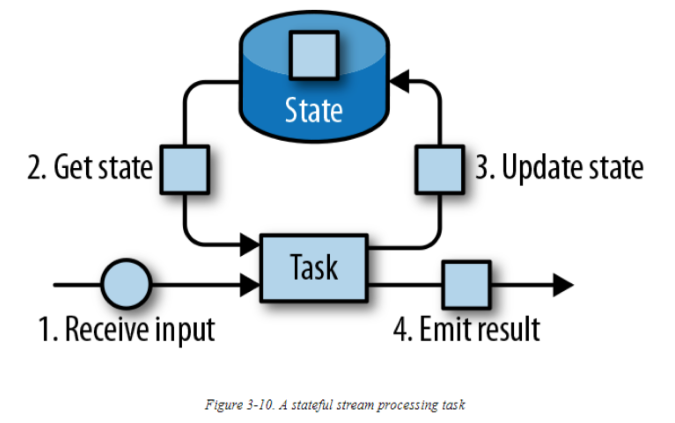
比如任务挂掉的时候或被手动停止的时候，可以从挂掉的点重新继续消费。基本类型：Operater state、Keyed state；特殊的Broadcast State。使用状态，必须使用RichFunction，因为状态是使用RuntimeContext访问的，只能在RichFunction中访问。

### 案例介绍

假设现在存在输入源数据格式为（EventId，Value），输出数据，直接flatMap即可，无状态，如果要输出某EventId最大值/最小值等，HashMap是否可以？程序一旦Crash，如何恢复？

答案：Flink提供了一套状态保存的方法，不需要借助第三方存储系统来解决状态存储的问题。

下图展示了一个task与它的state的常规交互过程：



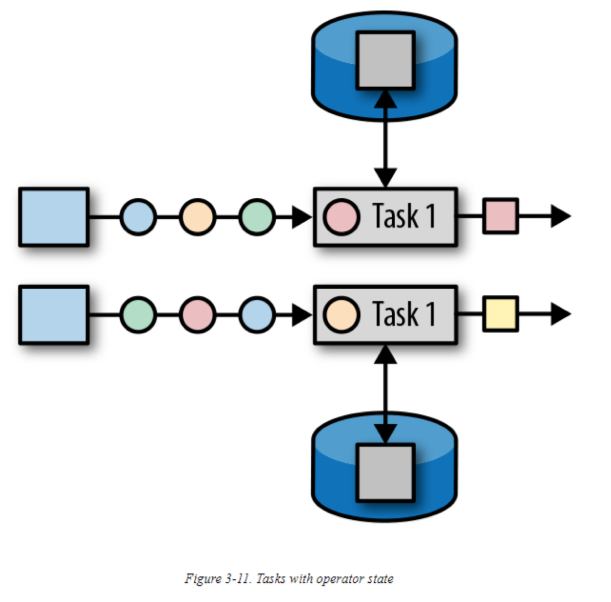
一个task接收一些输入数据。当处理数据时，task会访问state，并根据输入数据和state的信息更新它的state。一个简单的例子如：一个task持续计算迄今它接收到了多少条记录。当task接收到一个新的记录时，它会访问state获取当前的count数，增加count，更新state，并释放新的count作为结果输出。

### State类型

在Flink中，state一定是与一个特定的operator关联的。为了让Flink的runtime可以意识到一个operator的state，operator需要注册它的state。在Flink中有两种类型的state：operator state和keyed state。下面我们对它们做详细介绍。

#### Operator State

Operator state 被限定到一个operator task中，这个意思是：各个并行的task都有它自己的state，Operator state无法被其他task（无论是同一个还是不同的operator的task）访问。下图是tasks如何访问operator state



Flink为operator state提供了三种原型：

List state

· 以list的方式表示state

Union list state

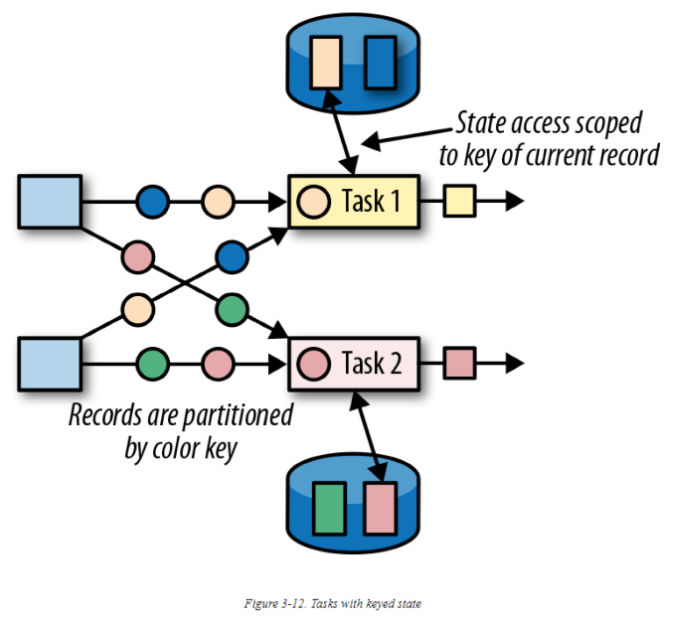
· 同样以list的方式表示state。但是它与常规list state的不同点在于：发生故障时恢复的方式、或一个application从检查点开始的方式。

Broadcast state

· 被用于特殊场景，当一个operator的每个task的state都是相同时。这个属性可以被用于检查点，或是rescaling 一个 operator时。

#### Keyed State

Keyed state 的维护与访问是根据对应记录中的key决定的。Flink对每个key value 维护了一个state 实例，并将所有同样key的记录，分区到维护这些key的state的operator task中。当一个task处理一条记录时，它会自动归类当前record的key所要访问的state。最终，所有具有相同key的records会访问同一个state。下图展示了tasks与keyed state 的交互：



可以将keyed state看做是：对一个operator所有并行tasks上的所有key做分区后的key-value map。Flink为 keyed state 提供了不同的原语，用于决定在分布式的key-value map中，每个key里存储的value类型。

Value state

·为每个key存一个单值(可以是任意类型)。复杂的数据结构也可以作为value state 存储

List state

·为每个key存一个列表值。这个列表可以是任意类型

Map state

·为每个key存一个key-value 映射。映射中的key和value可以是任意类型

State 原语为Flink提供了state的结构，并可以更高效的对state做访问。

### 状态后端(State Backends)

在有状态的operator中，它的task在每接受到一条记录时，一般都会访问、并更新state。因为高效地访问state 对于低延时处理records至关重要，所以每个并行的task都会在本地维护它的state，以确保快速访问state。State是如何准确的存储、访问、以及维护是由一个可插拔的组件决定的，这个组件成为状态后端（State backend）。一个state backend负责两件事：本地state管理，以及为state做检查点并存储到外部地址。

对于本地state 管理，state backend存储所有keyed state，并确保所有对keyed state的访问都符合当前key的条件。Flink提供的了state backend 将keyed state作为对象存储管理，并将它存储在JVM的堆内存中。另一个state backend 将state 对象序列化，并放入RocksDB中。RocksDB会将它们写入本地磁盘。第一个state backend 提供了快速访问state的选择，但是它会受到内存大小的限制。访问由RocksDB state backend存储的state会相对较慢，但是state可以增长到非常大。

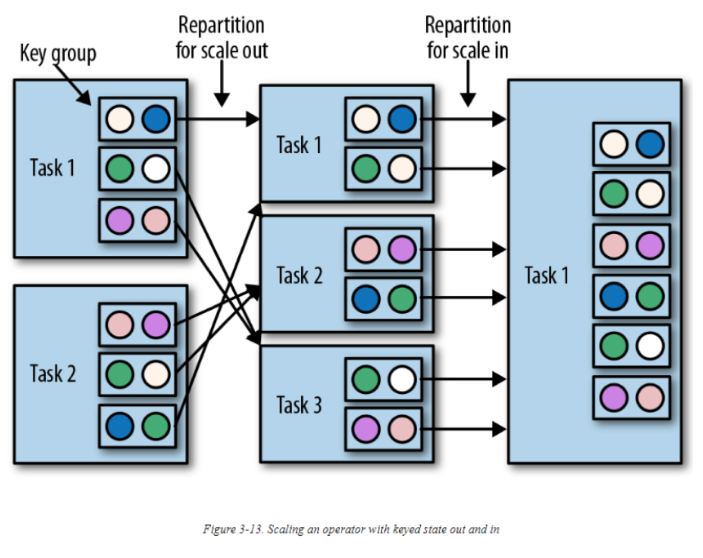
对state做检查点非常重要，因为Flink是一个分布式系统，并且state仅仅是本地维护的。一个TaskManager进程（包括里面所有运行的task）可能会在任何时候出现故障。所以它的存储必须被认为是不稳定的。一个state backend 会对一个task的state做检查点，并存储到远端的持久性存储中。存储检查点的远端存储可以是一个分布式文件系统，或是一个数据库系统。不同的state backend会有不同的为state做检查点的方式。例如，RocksDB state backend 支持增量检查点，此方法可以大量减少对超大state做检查点时的开销。

### 扩展有状态的operators

对于流处理程序来说，一个常见的需求是：根据输入数据的速率，调节operators的并行度。对于无状态的operators 来说，扩展是很简单的。但是对于有状态的operators，会更具挑战性，因为他们的state需要被重新分区，并分配给更多或是更少的并行tasks。Flink支持四种模式，用于扩展不同类型的state。

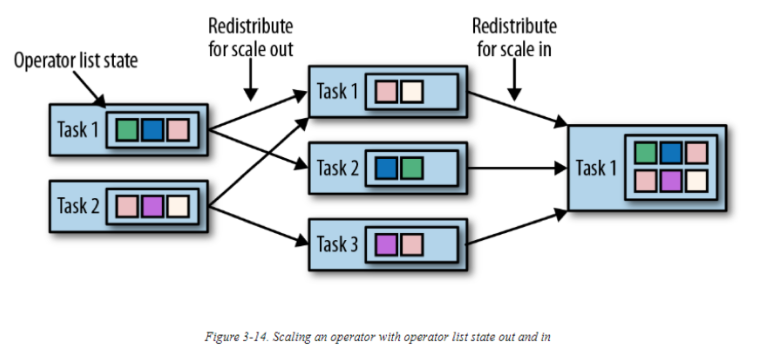
#### keyed state

对于keyed state的operators，扩展的实现方式是将keys重新分区到更少或是更多的tasks中。然而，为了提高tasks之间传递state的效率，Flink不会重新分布keys。它会将keys组织在一个或多个key groups中。一个key group不仅是keys的一个分区，也是Flink分配keys给tasks的方式。下图显示了keyed state 是如何在key groups中重新分区的：



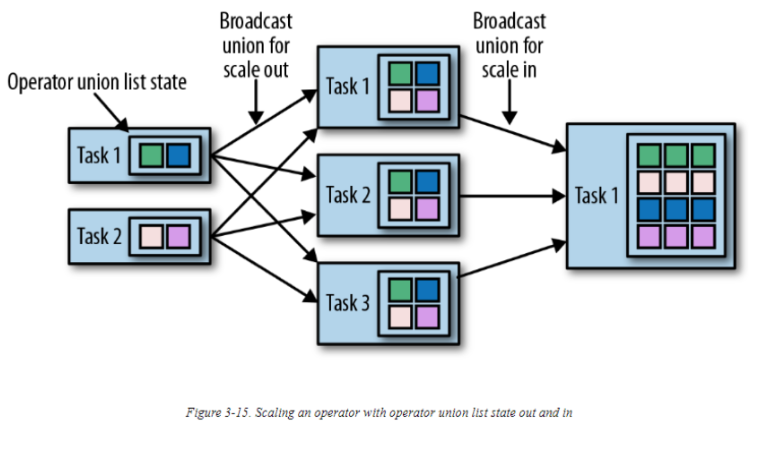
#### list state

在扩展state为list state 的operators时，列表里的条目会被重新分配。概念上，所有并行tasks的列表里的条目被收集并均分的重新分布到更少或是更多的tasks中。如果列表条目数小于operator的新并行数，则一些task会从空state开始。下图显示了operator list state的重新分布：



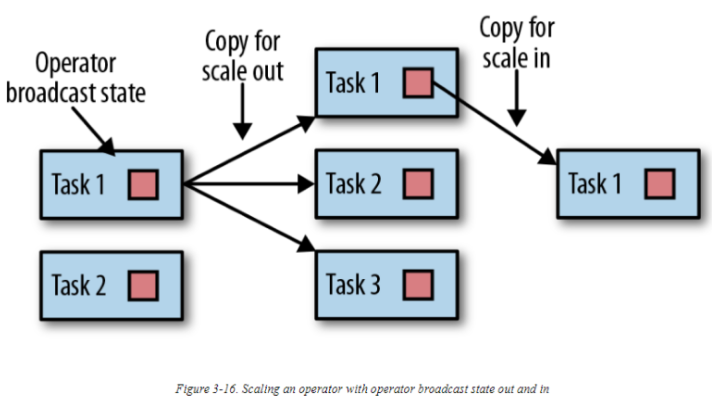
#### union list state

在扩展state 为union list state的operator时，列表中所有的state条目会被广播到每个task。Task之后可以选择使用哪些条目，丢弃哪些条目。下图显示了operator union list state 是如何重新分布的：



#### broadcast state

在扩展state为broadcast state 的operator时，state会被复制到新的task中。这里适用于这个操作是因为：广播state可以确保所有task有相同的state。在缩容时，多余的tasks会被简单地取消，因为state已经被复制了并且不会被丢失。下图显示的是operator broadcast state 的重新分布：



# 项目案例

## 案例1-实时数据

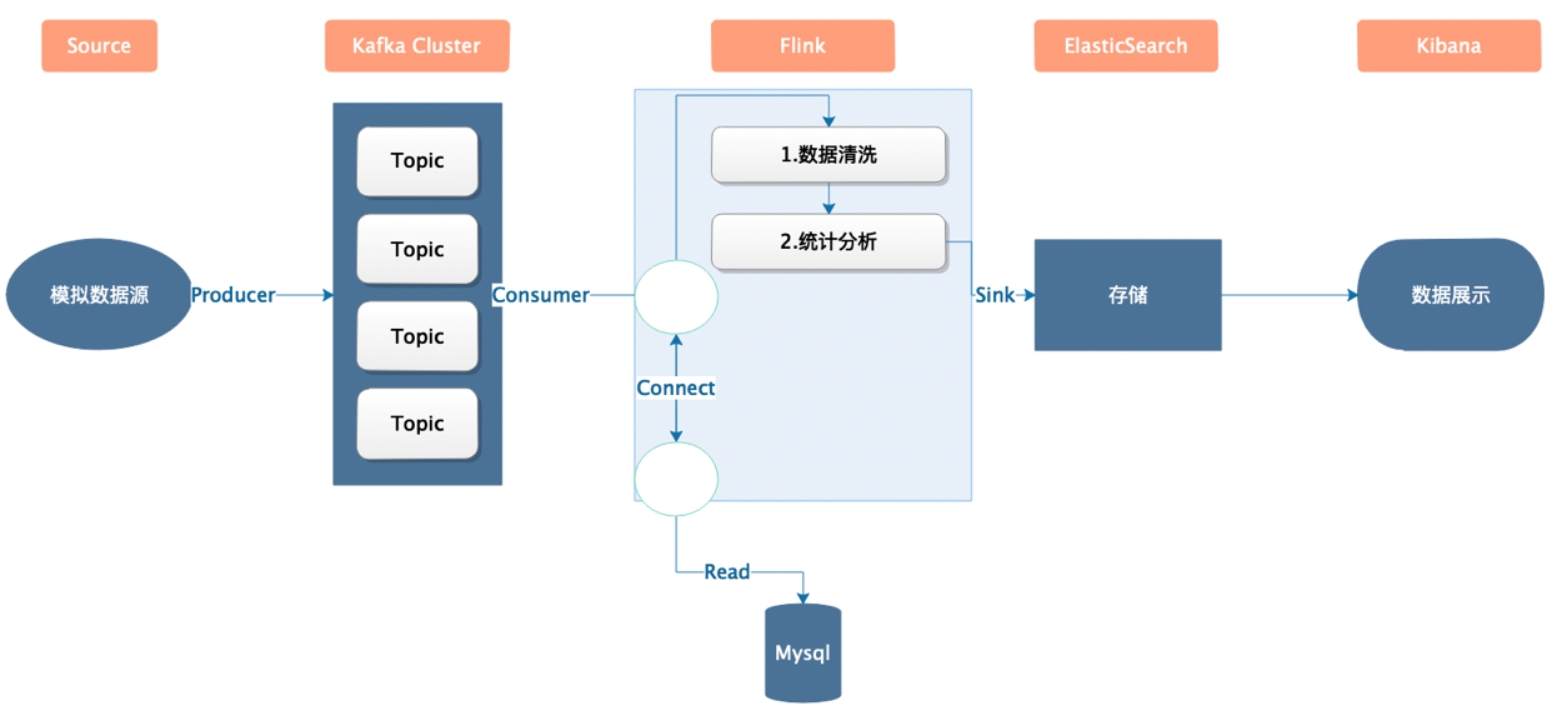
### 介绍

使用Flink对原始日志数据进行清洗、加工、计算后分别统计：

- 功能1 最近一分钟每个域名产生的流量。

- 功能2 一分钟内每个用户产生的流量（其中域名和用户有对应关系，数据存放于HDFS）

### 架构



### 代码

功能1 [TrafficAnalysisApp.scala](http://172.16.2.18:8690/svn/cloudil-java/trunk/cloudil-p3/flink/flink-local-train/src/main/scala/com/dinglicom/scala/example/TrafficAnalysisApp.scala)

功能2 [TrafficAnalysisApp2.scala](http://172.16.2.18:8690/svn/cloudil-java/trunk/cloudil-p3/flink/flink-local-train/src/main/scala/com/dinglicom/scala/example/TrafficAnalysisApp2.scala)

## 案例2-离线数据

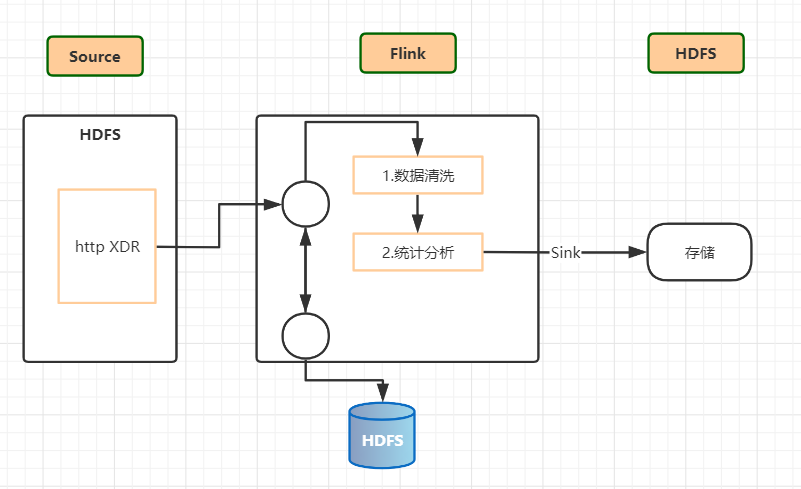
### 介绍

使用Flink对原始日志数据进行清洗、加工、计算后分别统计：

- 功能1 最近一小时物联网指标数据分析。

- 功能2 最近一小时物联网指标数据分析（过滤特定tac8数据，数据存放于HDFS）

### 结构



### 代码

功能1 [IOTKpiAnalysisApp.scala](http://172.16.2.18:8690/svn/cloudil-java/trunk/cloudil-p3/flink/flink-local-train/src/main/scala/com/dinglicom/scala/example/IOTKpiAnalysisApp.scala)

功能2 [IOTKpiAnalysisApp2.scala](http://172.16.2.18:8690/svn/cloudil-java/trunk/cloudil-p3/flink/flink-local-train/src/main/scala/com/dinglicom/scala/example/IOTKpiAnalysisApp2.scala)

# Flink on Yarn部署和任务提交操作

## 环境变量配置

提交Flink任务的client端必须要设置YARN\_CONF\_DIR或者HADOOP\_CONF\_DIR环境变量，通过这个环境变量来读取YARN和HDFS的配置信息，否则提交任务会失败。

## 任务提交方式

### 同时启动Yarn application和Flink task

flink run -m yarn-cluster -ys 8 -ynm myapp -yn 4 -yjm 1024 -ytm 4096 -d -c com.paultech.MyApp ./myapp.jar

参数说明：

-m 运行模式，这里使用yarn-cluster，即yarn集群模式。

-ys slot个数。

-ynm Yarn application的名字。

-yn task manager 数量。

-yjm job manager 的堆内存大小。

-ytm task manager 的堆内存大小。

-d detach模式。可以运行任务后无需再控制台保持连接。

-c 指定jar包中class全名。

### 先启动Yarn application，再在指定的application内运行Flink task

启动yarn session的命令如下：

yarn-session.sh -d -n 4 -nm riskmanater -jm 1024 -tm 4096 -t relative/path/to/file

相关参数解释：

-d: Detach模式

-nm: Application名称

-jm: Job Manager 容器的内存

-tm: Task Manager 容器的内存

-t: 传送文件至集群，使用相对路径。程序中读取文件仍使用相对路径

这里的参数和上面flink的类似，只不过少了前缀y

在特定Flink yarn session上提交Flink任务：

flink run -yid application\_12345678 --class MainClassFullPath riskmanager.jar

这里使用-yid参数来指定任务运行于那个Flink yarn session之上。

# 异常问题

## env.registerCachedFile(cfgfilePath，cacheName)

cfgfilePath只能为节点机本地地址，无法使用HDFS地址，原因还未查明。