精通Apache Flink读书笔记—1、2

1、Apache Flink介绍

既然有了Apache Spark，为什么还要使用Apache Flink?

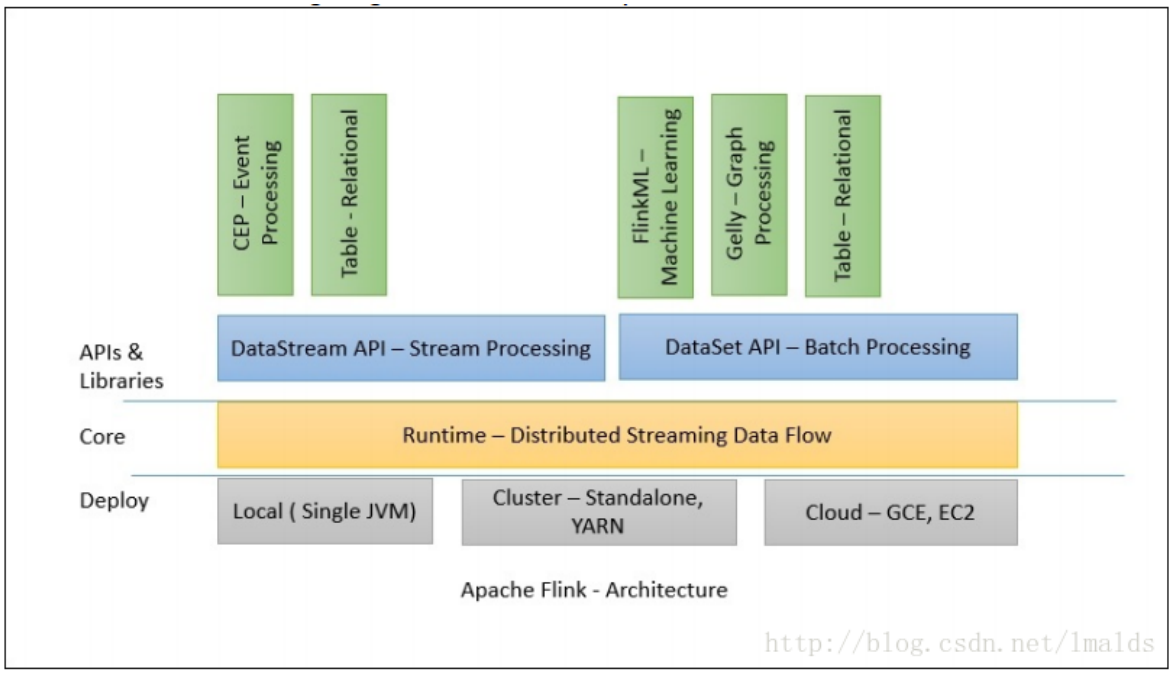
因为Flink是一个纯流式计算引擎，而类似于Spark这种微批的引擎，只是Flink流式引擎的一个特例。其他的不同点之后会陆续谈到。

1.1 历史

Flink起源于一个叫做Stratosphere的研究项目，目标是建立下一代大数据分析引擎，其在2014年4月16日成为Apache的孵化项目，从Stratosphere 0.6开始，正式更名为Flink。Flink 0.7中介绍了最重要的特性：Streaming API。最初只支持Java API，后来增加了Scala API。

1.2 架构

Flink 1.X版本的包含了各种各样的组件，包括部署、flink core（runtime）以及API和各种库。



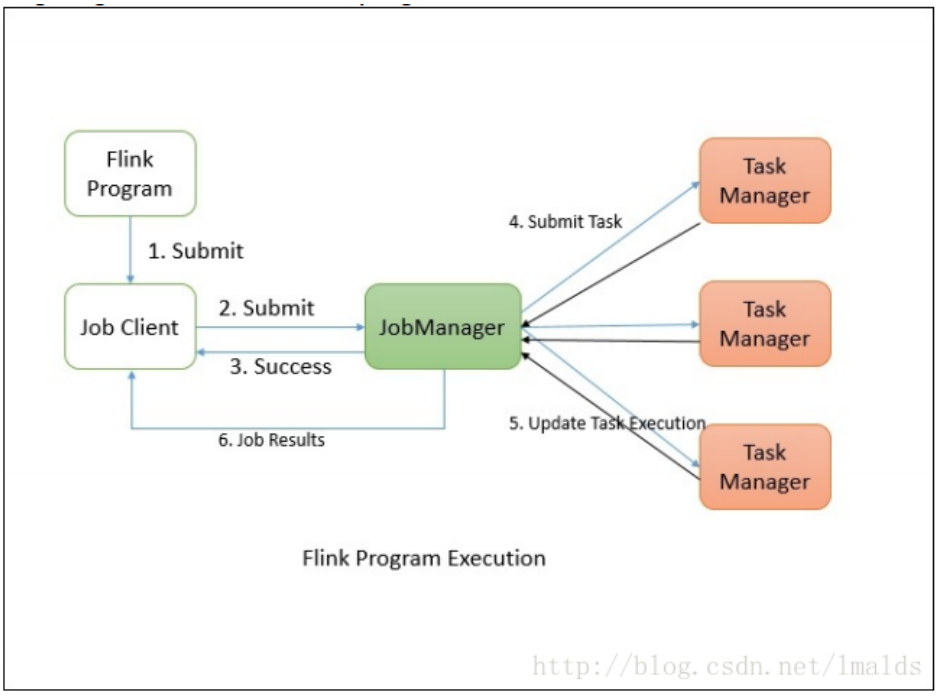
从部署上讲，Flink支持local模式、集群模式（standalone集群或者Yarn集群）、云端部署。Runtime是主要的数据处理引擎，它以JobGraph形式的API接收程序，JobGraph是一个简单的并行数据流，包含一系列的tasks，每个task包含了输入和输出（source和sink例外）。

DataStream API和DataSet API是流处理和批处理的应用程序接口，当程序在编译时，生成

JobGraph。编译完成后，根据API的不同，优化器（批或流）会生成不同的执行计划。根据部署方式的不同，优化后的JobGraph被提交给了executors去执行。

1.3 分布式执行

Flink分布式程序包含2个主要的进程：JobManager和TaskManager.当程序运行时，不同的进程就会参与其中，包括Jobmanager、TaskManager和JobClient。



首先，Flink程序提交给JobClient，JobClient再提交到JobManager，JobManager负责资源的协调和Job的执行。一旦资源分配完成，task就会分配到不同的TaskManager，TaskManager会初始化线程去执行task，并根据程序的执行状态向JobManager反馈，执行的状态包括starting、in progress、finished以及canceled和failing等。当Job执行完成，结果会返回给客户端。

1.3.1 JobManager

Master进程，负责Job的管理和资源的协调。包括任务调度，检查点管理，失败恢复等。

当然，对于集群HA模式，可以同时多个master进程，其中一个作为leader，其他作为standby。当leader失败时，会选出一个standby的master作为新的leader（通过zookeeper实现leader选举）。JobManager包含了3个重要的组件：

```
(1) Actor系统
(2) 调度
(3) 检查点
```

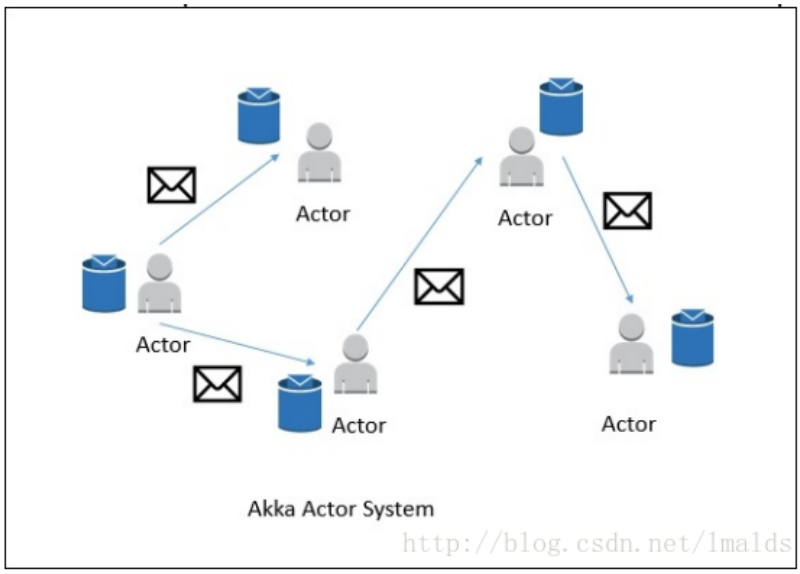
1.3.1.1 Actor系统

Flink内部使用Akka模型作为JobManager和TaskManager之间的通信机制。

Actor系统是个容器，包含许多不同的Actor，这些Actor扮演者不同的角色。Actor系统提供类似于调度、配置、日志等服务，同时包含了所有actors初始化时的线程池。

所有的Actors存在着层级的关系。新加入的Actor会被分配一个父类的Actor。Actors之间的通信采用一个消息系统，每个Actor都有一个“邮箱”，用于读取消息。如果Actors是本地的，则消息在共享内存中共享；如果Actors是远程的，则消息通过RPC远程调用。

每个父类的Actor都负责监控其子类Actor，当子类Actor出现错误时，自己先尝试重启并修复错误；如果子类Actor不能修复，则将问题升级并由父类Actor处理。
在Flink中，actor是一个有状态和行为的容器。Actor的线程持续的处理从“邮箱”中接收到的消息。Actor中的状态和行为则由收到的消息决定。



1.3.1.2 调度器

Flink中的Executors被定义为task slots（线程槽位）。每个Task Manager需要管理一个或多个task slots。

Flink通过SlotSharingGroup和CoLocationGroup来决定哪些task需要被共享，哪些task需要被单独的slot使用。

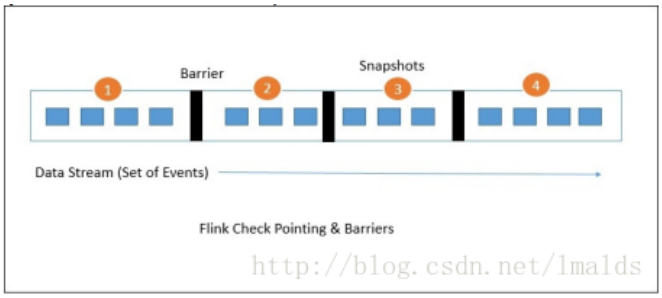
1.3.1.3 检查点

Flink的检查点机制是保证其一致性容错功能的骨架。它持续的为分布式的数据流和有状态的operator生成一致性的快照。其改良自Chandy-Lamport算法，叫做ABS（轻量级异步Barrier快照），具体参见论文：
Lightweight Asynchronous Snapshots for Distributed Dataflows

Flink的容错机制持续的构建轻量级的分布式快照，因此负载非常低。通常这些有状态的快照都被放

在HDFS中存储（state backend）。程序一旦失败，Flink将停止executor并从最近的完成了的检查点开始恢复（依赖可重发的数据源+快照）。

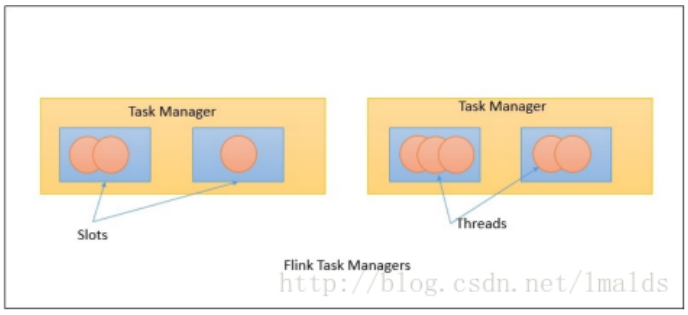
Barrier作为一种Event，是Flink快照中最主要的元素。它会随着data record一起被注入到流数据中，而且不会超越data record。每个barrier都有一个唯一的ID，将data record分到不同的检查点的范围中。下图展示了barrier是如何被注入到data record中的：



每个快照中的状态都会报告给Job Manager的检查点协调器；快照发生时，flink会在某些有状态的operator上对data record进行对齐操作（alignment），目的是避免失败恢复时重复消费数据。这个过程也是exactly once的保证。通常对齐操作的时间仅是毫秒级的。但是对于某些极端的应用，在每个operator上产生的毫秒级延迟也不能允许的话，则可以选择降级到at least once，即跳过对齐操作，当失败恢复时可能发生重复消费数据的情况。Flink默认采用exactly once意义的处理。

1.3.2 TaskManager

Task Managers是具体执行tasks的worker节点，执行发生在一个JVM中的一个或多个线程中。Task的并行度是由运行在Task Manager中的task slots的数量决定。如果一个Task Manager有4个slots，那么JVM的内存将分配给每个task slot 25%的内存。一个Task slot中可以运行1个或多个线程，同一个slot中的线程又可以共享相同的JVM。在相同的JVM中的tasks，会共享TCP连接和心跳消息：



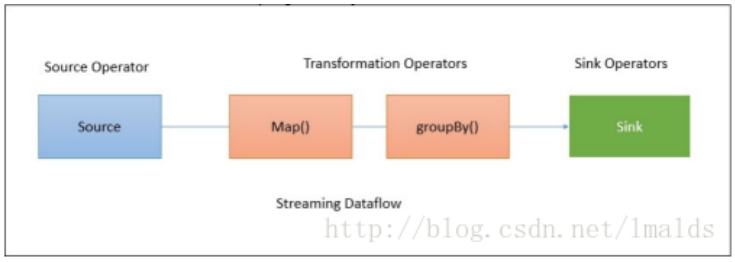
1.3.3 Job Client

Job Client并不是Flink程序执行中的内部组件，而是程序执行的入口。Job Client负责接收用户提交的程序，并创建一个data flow，然后将生成的data flow提交给Job Manager。一旦执行完成，Job Client将返回给用户结果。

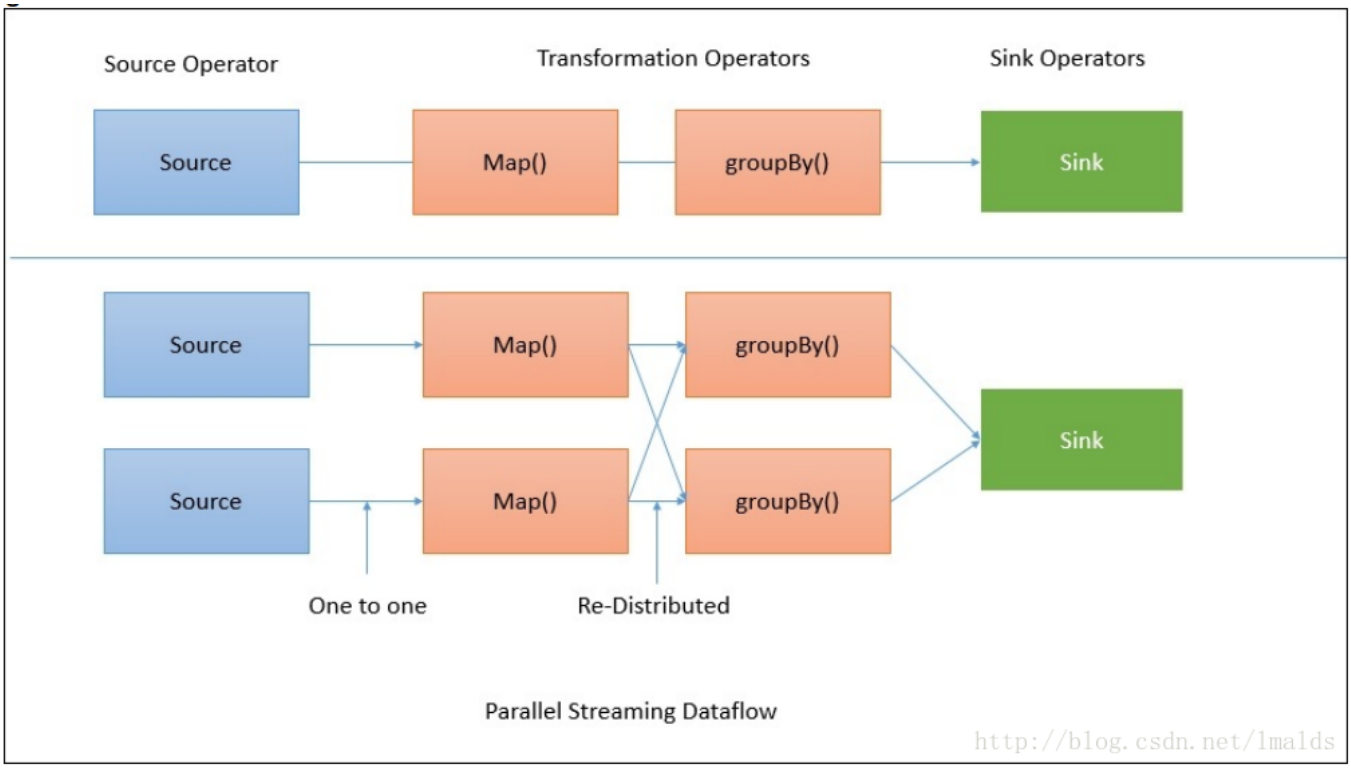
Data flow就是执行计划，比如下面一个简单的word count的程序：

```
val text =  env.readTextFile("input.txt"))        // Source

val counts = text.flatMap { _.toLowerCase.split("\\W+") filter { _.nonEmpty } }
      .map { (_, 1) }
      .groupBy(0)
      .sum(1)                                     // Transformation

counts.writeAsCsv("output.txt", "\n", " ")        // Sink
```


当用户将这段程序提交时，Job Client负责接收此程序，并根据operator生成一个data flow，那么这个程序生成的data flow也许看起来像是这个样子：



默认情况下，Flink的data flow都是分布式并行处理的，对于数据的并行处理，flink将operators和数据流进行partition。Operator partitions叫做sub-tasks。数据流又可以分为一对一的传输与重分布的情况。



我们看到，从source到map的data flow，是一个一对一的关系，没必要产生shuffle操作；而从map到groupBy操作，flink会根据key将数据重分布，即shuffle操作，目的是聚合数据，产生正确的结

果。

1.4 特性

1.4.1 高性能

Flink本身就被设计为高性能和低延迟的引擎。不像Spark这种框架，你没有必要做许多手动的配置，用以获得最佳性能，Flink管道式（pipeline）的数据处理方式已经给了你最佳的性能。

1.4.2 有状态的支持Exactly once的计算

通过检查点+可重发的数据源，使得Flink对于stateful的operator，支持exactly once的计算。当然你可以选择降级到at least once。

1.4.3 灵活的流处理窗口

Flink支持数据驱动的窗口，这意味着我们可以基于时间（event time或processing time）、count和session来构建窗口；窗口同时可以定制化，通过特定的pattern实现。

1.4.4 容错机制

通过轻量级、分布式快照实现。

1.4.5 内存管理

Flink在JVM内部进行内存的自我管理，使得其独立于java本身的垃圾回收机制。当处理hash、index、caching和sorting时，Flink自我的内存管理方式使得这些操作很高效。但是，目前自我的内存管理只在批处理中实现，流处理程序并未使用。

1.4.6 优化器

为了避免shuffle、sort等操作，Flink的批处理API进行了优化，它可以确保避免过度的磁盘IO而尽可能使用缓存。

1.4.7 流和批的统一

Flink中批和流有各自的API，你既可以开发批程序，也可以开发流处理程序。事实上，Flink中的流处理优先原则，认为批处理是流处理的一种特殊情况。

1.4.8 Libraries库

Flink提供了用于机器学习、图计算、Table API等库，同时Flink也支持复杂的CEP处理和警告。

1.4.9 Event Time语义

Flink支持Event Time语义的处理，这有助于处理流计算中的乱序问题，有些数据也许会迟到，我们

可以通过基于event time、count、session的窗口来处于这样的场景。

1.5 快速安装

直接参见官方文档：QuickStart

1.6 Standalone 集群安装

直接参见官方文档：Standalone Cluster

1.7 例子

略去，可参见官方文档：Examples

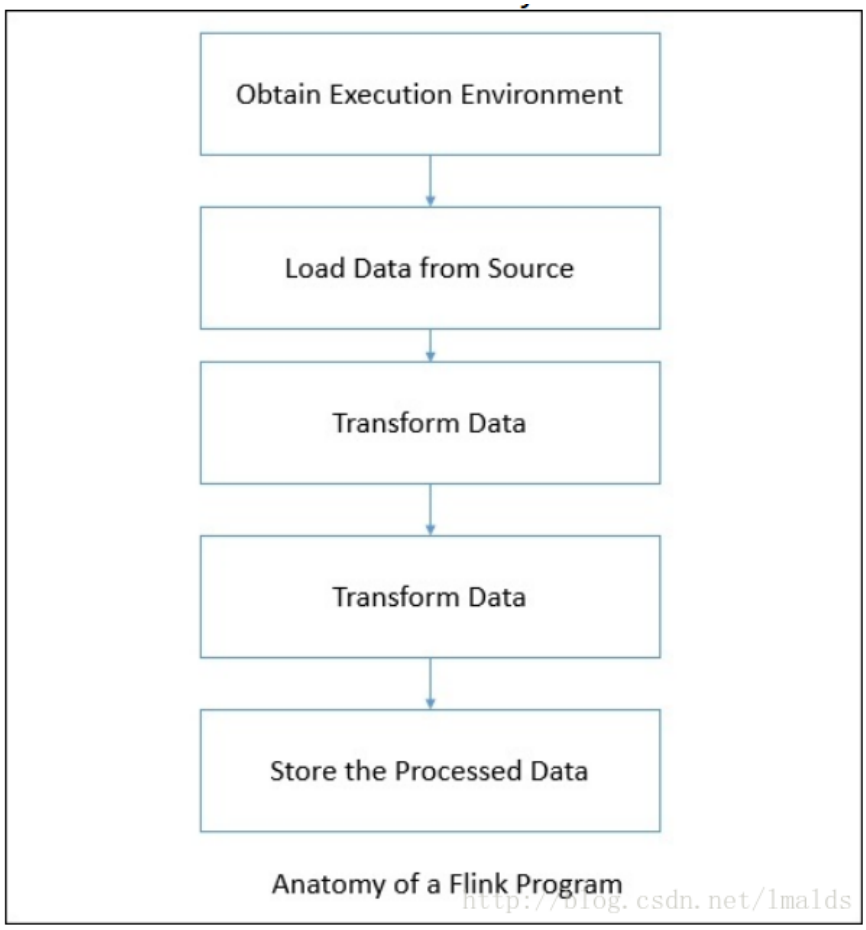
1.8 总结

Flink细节上的讨论和处理模型。下一章将介绍Flink Streaming API。

2、用DataStream API处理数据

许多领域需要数据的实时处理，物联网驱动的应用程序在数据的存储、处理和分析上需要实时或准实时的进行。

Flink提供流处理的API叫做DataStream API，每个Flink程序都可以按照下面的步骤进行开发：



2.1 运行环境

我们首先要获得已经存在的运行环境或者创建它。有3种方法得到运行环境：

```
(1) 通过getExecutionEnvironment()获得；这将根据上下文得到运行环境，假如local模式，
(2) 通过createLocalEnvironment() 创建一个本地的运行环境；
(3) 通过createRemoteEnvironment (String host, int port, String, and .jar fi
```

2.2 数据源

Flink支持许多预定义的数据源，同时也支持自定义数据源。下面我们看看有哪些预定义的数据源。

2.2.1 基于socket

DataStream API支持从socket读取数据，有如下3个方法：

```
socketTextStream(hostName, port);
socketTextStream(hostName,port,delimiter)
socketTextStream(hostName,port,delimiter, maxRetry)
```

2.2.2 基于文件

你可以使用readTextFile(String path)来消费文件中的数据作为流数据的来源，默认情况下的格式是TextInputFormat。当然你也可以通过readFile(FileInputFormat inputFormat, String path)来指定FileInputFormat的格式。

Flink同样支持读取文件流：

```
readFileStream(String filePath, long intervalMillis,
FileMonitoringFunction.WatchType watchType)

readFile(fileInputFormat, path, watchType, interval, pathFilter,
typeInfo)。
```

关于基于文件的数据流，这里不再过多介绍。

2.2.3 Transformation

Transformation允许将数据从一种形式转换为另一种形式，输入可以是1个源也可以是多个，输出则可以是0个、1个或者多个。下面我们一一介绍这些Transformations。

2.2.3.1 Map

输入1个元素，输出一个元素，Java API如下：

```
inputStream.map(new MapFunction<Integer, Integer>() {
@Override
public Integer map(Integer value) throws Exception {
return 5 * value;
}
});
```

2.2.3.2 FlatMap

输入1个元素，输出0个、1个或多个元素，Java API如下：

```
inputStream.flatMap(new FlatMapFunction<String, String>() {
@Override
public void flatMap(String value, Collector<String> out)
throws Exception {
for(String word: value.split(" ")){
out.collect(word);
}
}
});
```

2.2.3.3 Filter

条件过滤时使用，当结果为true时，输出记录；

```
inputStream.filter(new FilterFunction<Integer>() {
@Override
public boolean filter(Integer value) throws Exception {
return value != 1;
}
});
```

2.2.3.4 keyBy

逻辑上按照key分组，内部使用hash函数进行分组，返回keyedDataStream：

```
inputStream.keyBy("someKey");
```

2.2.3.5 Reduce

keyedStream流上，将上一次reduce的结果和本次的进行操作，例如sum reduce的例子：

```
keyedInputStream. reduce(new ReduceFunction<Integer>() {
@Override
public Integer reduce(Integer value1, Integer value2)
throws Exception {
return value1 + value2;
}
});
```

2.2.3.6 Fold

在keyedStream流上的记录进行连接操作，例如：

```
keyedInputStream keyedStream.fold("Start", new FoldFunction<Integer,
String>() {
@Override
public String fold(String current, Integer value) {
return current + "=" + value;
}
});
```

假如是一个（1,2,3,4,5）的流，那么结果将是：Start=1=2=3=4=5

2.2.3.7 Aggregation

在keyedStream上应用类似min、max等聚合操作：

```
keyedInputStream.sum(0)
```

```
keyedInputStream.sum("key")
keyedInputStream.min(0)
keyedInputStream.min("key")
keyedInputStream.max(0)
keyedInputStream.max("key")
keyedInputStream.minBy(0)
keyedInputStream.minBy("key")
keyedInputStream.maxBy(0)
keyedInputStream.maxBy("key")
```

2.2.3.8 Window

窗口功能允许在keyedStream上应用时间或者其他条件（count或session），根据key分组做聚合操作。

流是无界的，为了处理无界的流，我们可以将流切分到有界的窗口中去处理，根据指定的key，切分为不同的窗口。我们可以使用Flink预定义的窗口分配器。当然你也可以通过继承WindowAssginer自定义分配器。

下面看看有哪些预定义的分配器。

2.2.3.8.1 Global windows

Global window的范围是无限的，你需要指定触发器来触发窗口。通常来讲，每个数据按照指定的key分配到不同的窗口中，如果不指定触发器，则窗口永远不会触发。

2.2.3.8.2 Tumbling Windows

Tumbling窗口是基于特定时间创建的，他们的大小固定，窗口间不会发生重合。例如你想基于event timen每隔10分钟计算一次，这个窗口就很适合。

2.2.3.8.3 Sliding Windows

Sliding窗口的大小也是固定的，但窗口之间会发生重合，例如你想基于event time每隔1分钟，统一过去10分钟的数据时，这个窗口就很适合。

2.2.3.8.4 Session Windows

Session窗口允许我们设置一个gap时间，来决定在关闭一个session之前，我们要等待多长时间，是衡量用户活跃与否的标志。

2.2.3.9 WindowAll

WindowAll操作不是基于key的，是对全局数据进行的计算。由于不基于key，因此是非并行的，即并行度是1.在使用时性能会受些影响。

```
inputStream.windowAll(TumblingEventTimeWindows.of(Time.seconds(10)));
```

2.2.3.10 Union

Union功能就是在2个或多个DataStream上进行连接，成为一个新的DataStream。

```
inputStream. union(inputStream1, inputStream2, ...)
```

2.2.3.11 Join

Join允许在2个DataStream上基于相同的key进行连接操作，计算的范围也是要基于一个window进行。

```
inputStream. join(inputStream1)
.where(0).equalTo(1)
.window(TumblingEventTimeWindows.of(Time.seconds(5)))
.apply (new JoinFunction () {...})
```

2.2.3.12 Split

Split的功能是根据某些条件将一个流切分为2个或多个流。例如你有一个混合数据的流，根据数据自身的某些特征，将其划分到多个不同的流单独处理。

```
SplitStream<Integer> split = inputStream.split(new
OutputSelector<Integer>() {
@Override
public Iterable<String> select(Integer value) {
List<String> output = new ArrayList<String>();
if (value % 2 == 0) {
output.add("even");
}else {
output.add("odd");}
return output;
}
})
```

2.2.3.13 select

DataStream根据选择的字段，将流转换为新的流。

```
SplitStream<Integer> split;
DataStream<Integer> even = split.select("even");
DataStream<Integer> odd = split.select("odd");
DataStream<Integer> all = split.select("even","odd");
```

2.2.3.14 project

Project功能允许你选择流中的一部分元素作为新的数据流中的字段，相当于做个映射。

```
DataStream<Tuple4<Integer, Double, String, String>> in = // [...]
DataStream<Tuple2<String, String>> out = in.project(3,2);
```

2.2.4 物理分片

Flink允许我们在流上执行物理分片，当然你可以选择自定义partitioning。

2.2.4.1 自定义partitioning

根据某个具体的key，将DataStream中的元素按照key重新进行分片，将相同key的元素聚合到一个线程中执行。

```
inputStream.partitionCustom(partitioner, "someKey");
inputStream.partitionCustom(partitioner, 0);
```

2.2.4.2 随机partitioning

不根据具体的key，而是随机将数据打散。

```
inputStream.shuffle();
```

2.2.4.3 Rebalancing partitioning

内部使用round robin方法将数据均匀打散。这对于数据倾斜时是很好的选择。

```
inputStream.rebalance();
```

2.2.4.4 Rescaling

Rescaling是通过执行oepration算子来实现的。由于这种方式仅发生在一个单一的节点，因此没有跨网络的数据传输。

```
inputStream.rescale();
```

2.2.4.5 广播

广播用于将dataStream所有数据发到每一个partition。

```
inputStream.broadcast();
```

2.2.5 数据Sink

我们最终需要将结果保存在某个地方，Flink提供了一些选项：

```
(1) writeAsText()：将结果以字符串的形式一行一行写到文本文件中。

(2) writeAsCsV()：保存为csv格式。

(3) print()/printErr()：标准输出或错误输出。输出到Terminal或者out文件。

(4) writeUsingOutputFormat()：自定义输出格式，要考虑序列化与反序列化。

(5) writeUsingOutputFormat()：也可以输出到socket，但是你需要定义SerializationScl
```

对于Flink中的connector以及自定义输出，后续的章节会讲到。

2.2.6 Event Time和watermark

Flink Streaming API受到了Google DataFlow模型的启发，支持3种不同类型的时间概念：

```
(1)  Event Time
(2)  Processing Time
(3)  Ingestion Time
```

（1）Event Time
事件发生的时间，一般数据中自带时间戳。这就可能导致乱序的发生。

（2）Processing Time
Processing Time是机器的时间，这种时间跟数据本身没有关系，完全依赖于机器的时间。

（3）Ingestion Time
是数据进入到Flink的时间。注入时间比processing time更加昂贵（多了一个assign timestamp的步骤），但是其准确性相比processing time的处理更好。由于是进入Flink才分配时间戳，因此无法处理乱序。

我们通过在env中设置时间属性来选择不同的时间概念：

```
final StreamExecutionEnvironment env =
StreamExecutionEnvironment.getExecutionEnvironment();
env.setStreamTimeCharacteristic(TimeCharacteristic.ProcessingTime);
//or
env.setStreamTimeCharacteristic(TimeCharacteristic.IngestionTime);
//or
env.setStreamTimeCharacteristic(TimeCharacteristic.EventTime);
```

Flink提供了预定义的时间戳抽取器和水位线生成器。参考：

Pre-defined Timestamp Extractors / Watermark Emitters

2.2.7 connectors连接器

2.2.7.1 Kafka connector

kafka是一个基于发布、订阅的分布式消息系统。Flink定义了kafka consumer作为数据源。我们只需要引入特定的依赖即可（这里以kafka 0.9为例）：

```
<dependency>
    <groupId>org.apache.flink</groupId>
    <artifactId>flink-connector-kafka-0.9_2.11/artifactId>
    <version>1.1.4</version>
</dependency
```

在使用时，我们需要指定topic name以及反序列化器：

```
Properties properties = new Properties();
properties.setProperty("bootstrap.servers", "localhost:9092");
properties.setProperty("group.id", "test");
DataStream<String> input = env.addSource(new
FlinkKafkaConsumer09<String>("mytopic", new SimpleStringSchema(),
properties));
```

Flink默认支持String和Json的反序列化。

Kafka consumer在实现时实现了检查点功能，因此失败恢复时可以重发。

Kafka除了consumer外，我们也可以将结果输出到kafka。即kafka producer。例如：

```
stream.addSink(new FlinkKafkaProducer09[String]("localhost:9092",
"mytopic", new SimpleStringSchema()))
```

2.2.7.2 Twitter connector

用twitter作为数据源，首先你需要用于twitter账号。之后你需要创建twitter应用并认证。

这里有个帮助文档：https://dev.twitter.com/oauth/overview/application-owner-access-tokens

Pom中添加依赖：

```
<dependency>
    <groupId>org.apache.flink</groupId>
    <artifactId>flink-connector-twitter_2.11/artifactId>
    <version>1.1.4</version>
```

```
</dependency>
```

API：

```
Properties props = new Properties();
props.setProperty(TwitterSource.CONSUMER_KEY, "");
props.setProperty(TwitterSource.CONSUMER_SECRET, "");
props.setProperty(TwitterSource.TOKEN, "");
props.setProperty(TwitterSource.TOKEN_SECRET, "");
DataStream<String> streamSource = env.addSource(new
TwitterSource(props));
```

2.2.7.3 RabbitMQ connector

2.2.7.4 ElasticSearch connector

2.2.7.5 Cassandra connector

这3个connetor略过，壳参考官方文档：

https://flink.apache.org/ecosystem.html

2.2.8 例子

这里可以参考OSCON的例子：

https://github.com/dataArtisans/oscon。

2.2.9 总结

本章介绍了Flink的DataStream API，下一章将介绍DataSet API。