Flink Stream DataFlow

1. Flink数据流图

如图1所示，事件从基站等地方生成之后，会以数据流的形式流入Flink系统，通过Source、Transformation以及Sink Operator的计算，最终流入数据库、文件系统等存储介质中。根据事件发生的位置，Flink提出了三种时间概念：EventTime,IngestionTime以及ProcessingTime。EventTime是事件发生时的时间；IngestionTime是事件进入流处理系统的时间；ProcessingTime是数据被计算处理的时间。后续介绍的Window中使用的分配策略就将Window和Time结合到了一起。



图 1 数据流图

Flink的DataFlow中可以有一个或多个Source/Transformation/Sink Operator（算子），这些Operator可以通过调用相应的Function来实现用户逻辑的执行，同时，Operator通过设置不同的paramllelism（并行度），可以实现多个subTask并行执行。例如，图2为并行度为1的数据流逻辑视图，如果设置FlatMap、Window以及Sink的并行度为2，则优化后的数据流并行化视图如图3所示。其中，subTask的数目对应线程数，即有多少个subTask就会有多少个线程。



图 2数据流逻辑视图



图 3 数据流并行化视图

图3中通过优化将Window和Sink合成了一个subTask，这被成为一个OperatorChain。OperatorChain是用于将Operator串联起来，这样被串联起来的Operator之间传递数据就不需要序列化/反序列化，也不需要网络传输等。这两个Operator可以进行合并是由于其合并后不会改变整体的拓扑结构，但是Operator串联在一起是有一定要求的：

（1）上下游的并行度一致；

（2）下游节点的入度为1（也就是说下游节点没有来自其他节点的输入）；

（3）上下游节点都在同一个slot共享组中（Flink 允许subtasks共享slot，条件是它们都来自同一个Job的不同task的subtask。可能一个slot持有该job的整个pipeline。）；

（4）下游节点的chain策略为ALWAYS（可以与上下游链接，map、flatmap、filter等默认是ALWAYS）；

（5）上游节点的chain策略为ALWAYS或HEAD（只能与下游链接，不能与上游链接，Source默认为HEAD）；

（6）两个节点间数据分区方式是forward；

（7）用户没有禁用chain。

数据以Stream的形式在Operator中传递，其传递的模式有以下两种：

1. One-To-One模式：即forward模式，该模式下Operator看到的元素的个数以及顺序与前一个Operator输出的一致。
2. Redistribute模式：即Operator向其下一个Operator传递数据时，会对数据进行重新分布，主要包括rebalance(),shuffle(),broadcast()等模式。其中，keyBy()通过hash的方式分区；rebalance()使用循环分区方式，以循环方式将元素均匀分布到下一个Operator中；shuffle()根据均匀分布方式随机为下一个Operator分配元素；broadcast()将元素广播到所有的分区中。
3. Source

Source可以从文件、Socket、Collection获取数据，也可以使用addSource(SourceFunction)进行用户自定义的数据生成或从外部（如Kafka）获取数据。数据的类型包括：Java Tuple（支持Tuple1到Tuple25）；Java Pojo；Integer、String、Double等基本类型。

目前使用的数据生成是通过实现SourceFunction接口的方式，生成流速变化的数据。同时，通过设置最大执行时间，由TerminateThread来控制数据的生成。

private static class Tuple2SourceFunction extends AbstractSourceFunction implements SourceFunction<Tuple2<Long, Long>> {  
 private static final long *serialVersionUID* = 1L;  
 private TerminatorThread terminator;  
 private long maxexecutiontime;  
 public Tuple2SourceFunction(long maxexecutiontime) {  
 this.maxexecutiontime = maxexecutiontime;  
 }  
  
 public void run(SourceContext<Tuple2<Long, Long>> ctx) throws Exception {  
 terminator = new TerminatorThread(maxexecutiontime, this);  
 terminator.start();  
 System.*out*.println("start generating");  
 while (isRunning) {  
 resetParams();  
 long startTimeNanos = System.*nanoTime*();  
 while (targetDone < targetPerSecond && isRunning) {  
 ctx.collect(new Tuple2<>((long) getRnd().nextInt(10), (long) getRnd().nextGaussian() + 1));  
 throttleNanos(startTimeNanos);  
 }  
 }  
 }  
  
 public void cancel() {  
 isRunning = false;  
 }  
}

1. Window

Window编程分为两种模式，Keyed Window和Non-Keyed Window，唯一的区别是数据流是不是KeyedStream。

1. Keyed Windows：

stream

.keyBy(…)

.window(WindowAssigner)

[.trigger(Trigger)]

[.evictor(Evictor)]

[.allowedLateness(Time)]

.reduce/fold/apply()

1. Non-Keyed Windows:

stream

.windowAll(WindowAssigner)

[.trigger(Trigger)]

[.evictor(Evictor)]

[.allowedLateness(Time)]

.reduce/fold/apply()

使用Keyed Window可以多任务的并行的执行窗口操作，因为每个Keyed Stream可以被单独处理。另外，同一个Key的所有元素将在同一个任务中执行。Non-Keyed Window将不会分割成多个逻辑流，并且所有窗口逻辑都由单个任务执行，即其parallelism为1。

* **windowAll为何只用单个任务执行？**

DataStream提供了windowAll(WindowAssigner)方法，该方法会返回一个AllWindowedStream。AllWindowedStream提供了一系列的方法，其转换关系如下：

Sum()/min()/minBy()/max()/maxBy()🡪aggregate()🡪reduce(aggregator);

reduce()🡪apply(reduceFunction,function,resultType);

fold()🡪apply(initialValue,foldFunction,function,resultType);

apply()🡪apply(function,resultType);

reduce()/fold()/apply()最后转换成相应的apply()方法，且这三种类型的apply()方法最后都会返回以下信息：

return input.transform(opName, resultType, operator).setParallelism(1);

从源码中得出，使用DataStream的windowAll()方法进行的操作，最终都会被设置并行度为1，而在外部通过env.setParamllelism()方法设置的并行度将不再起作用。

数据流经Operator会按照定义的Function（如FlatMapFunction、WindowFunction等）进行相应的操作。下面主要介绍数据流经Window时的操作。图4展示了数据流经一个Window时的操作。

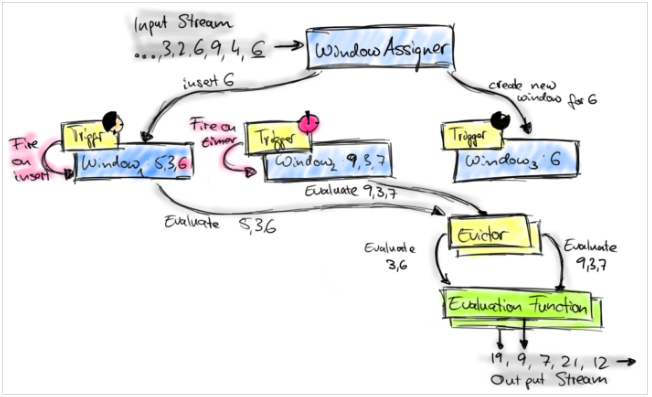


图 4 Window数据处理流程

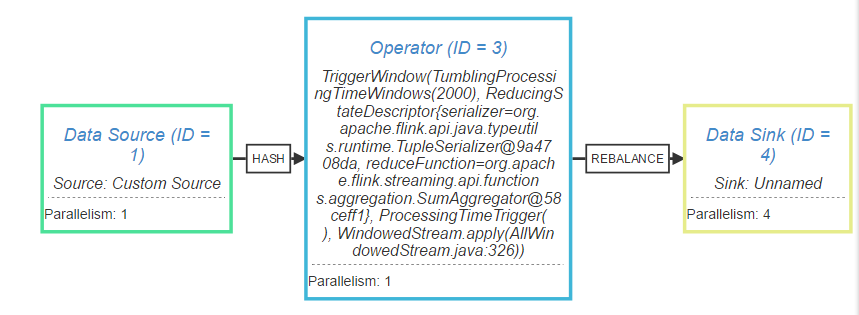
其中,WindowAssigner用于决定元素被分配到哪个窗口中；Trigger决定了一个窗口何时能够被计算或清除，每个窗口都会拥有一个自己的Trigger；在窗口被处理之前，Evictor（如果有Evictor的话）会用来剔除窗口中不需要的元素，相当于一个filter，但性能会显着降低，因为窗口结果的增量聚合不能使用。allowedLateness为允许的延迟时间，只用于Event-time中。

Trigger被触发后会返回TriggerResult，共有四种类型：

1. Continue：对该窗口不做任何操作。
2. Fire：窗口中的数据被计算，并且将结果传递到下一个Operator。
3. Fire and Purge:窗口中的数据被计算，并将结果传递出去，同时将窗口中的数据清除掉。
4. Purge：窗口中的所有元素都被清除，窗口也会被丢弃，不执行窗口函数，也不向外传递数据。

默认情况下，使用的Trigger是返回Fire，即如果窗口中的数据是存储在内存中，则这些数据不会被清除。

windowAll()所有窗口逻辑都由单个任务执行，即其parallelism为1。



1. State

见文档Flink State。

// RichFlatMapFunction  
private static class CountWindowAverage extends RichFlatMapFunction<Tuple2<Long, Long>, Tuple2<Long, Long>> {  
 */\*\*  
 \* The ValueState handle. The first field is the count, the second field a running sum.  
 \*/* private transient ValueState<Tuple2<Long, Long>> sum;  
  
 @Override  
 public void flatMap(Tuple2<Long, Long> input, Collector<Tuple2<Long, Long>> out) throws Exception {  
  
 // access the state value  
 Tuple2<Long, Long> currentSum = sum.value();  
  
 // update the count  
 currentSum.f0 += 1;  
  
 // add the second field of the input value  
 currentSum.f1 += input.f1;  
  
 // update the state  
 sum.update(currentSum);  
  
 // if the count reaches 2, emit the average and clear the state  
 if (currentSum.f0 >= 2) {  
 out.collect(new Tuple2<Long, Long>(input.f0, currentSum.f1 / currentSum.f0));  
 sum.clear();  
 }  
 }  
  
 @Override  
 public void open(Configuration config) {  
 ValueStateDescriptor<Tuple2<Long, Long>> descriptor =  
 new ValueStateDescriptor<>(  
 "average", // the state name  
 TypeInformation.*of*(new TypeHint<Tuple2<Long, Long>>() {  
 }), // type information  
 Tuple2.*of*(0L, 0L)); // default value of the state, if nothing was set  
 sum = getRuntimeContext().getState(descriptor);  
 }  
}

1. Sink

Sink可以以Text或CSV格式写入文件，print出来，也可以通过addSink(SinkFunction)向特定的文件系统等写入数据。

写入文件或打印出来的方式的语义保障比较低，如果想要确保“恰好一次”的语义，需要将其写入到文件系统中。

参考文献：

[1] http://www.tuicool.com/articles/7FjqQjb