

★ Get unlimited access to the best of Medium for less than \$1/week. [Become a member](#)



Basic Apache Flink Tutorial: DataStream API Programming

 Alibaba Cloud [Follow](#) 14 min read · Jan 2, 2020

 43     

By Cui Xingcan, an external committer and collated by Gao Yun

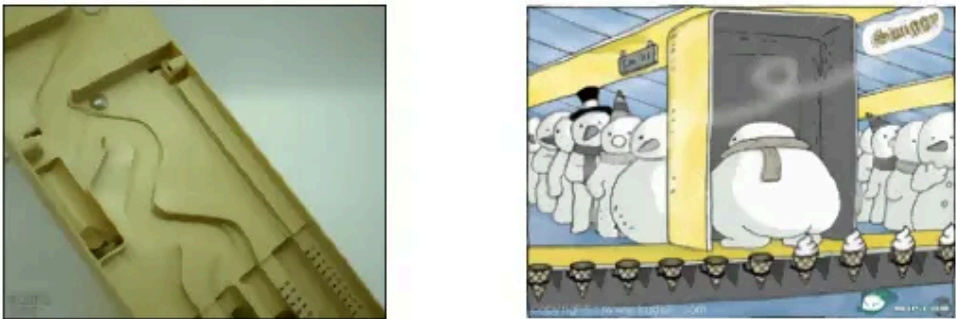
This article explains the basic concepts, installation, and deployment process of Flink. This article focuses on Flink development and describes the DataStream API, which is the core of Flink development.

Basic Concepts of Stream Processing

The definition of stream processing may vary. Conceptually, stream processing and batch processing are two sides of the same coin. Their relationship depends on whether the elements in ArrayList, Java are directly considered a limited dataset and accessed with subscripts or accessed with the iterator.



What is the stream processing?



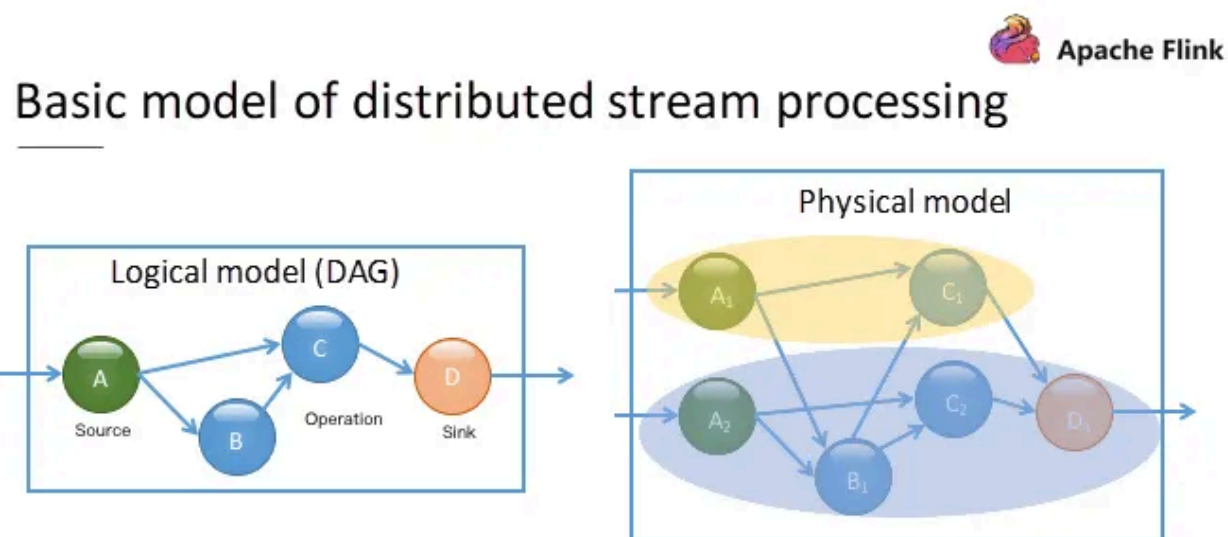
© Apache Flink Community China. Commercial use is strictly prohibited.

Figure 1. On the left is a coin classifier.

We can describe a coin classifier as a stream processing system. In advance, all components used for coin classification connect in series. Coins continuously enter the system and output to different queues for future use. The same is true for the picture on the right.

The stream processing system has many features. Generally, a stream processing system uses a data-driven processing method to support the processing of infinite datasets. It sets operators in advance and then processes the data. To express complex computational logic, distributed stream processing engines, including Flink, generally use DAG graphs to represent the entire computational logic.

Each point in DAG represents a basic logical unit — the operator mentioned earlier. Organize the computational logic into a directed graph where the data flows into the system from special source nodes from the edge. The data is transmitted and processed between operators through different data transmission methods, such as network transmissions and local transmissions. Finally, the data result are sent to an external system, or database through other special sink nodes.



© Apache Flink Community China. Commercial use is strictly prohibited.

Figure 2. A DAG computing logic graph and an actual runtime physical model.

Each operator in the logic graph has multiple concurrent threads in the physical graph. For distributed stream processing engines, their actual runtime physical models are more complex because each operator may have multiple instances. As shown in Figure 2, the source operator A has two instances; intermediate operator C has two instances, too.

In the logical model, A and B are the upstream nodes of C. In the corresponding physical model, data exchange may exist between all instances of C, A, and B. In the physical model, we use automatic system optimization or manual designated methods to distribute computing jobs to different instances based on the computing logic.

Data is transmitted over the network when operator instances are distributed to different processes. Data transmission between multiple instances in the same process usually does not need to be carried out through the network.

Table 1 A DAG computational graph is constructed with Apache Storm. The API definition of Apache Storm is “operations-oriented,” so it is lower-level.

```
TopologyBuilder builder = new TopologyBuilder();

builder.setSpout("spout", new RandomSentenceSpout(), 5);
builder.setBolt("split", new SplitSentence(),
8).shuffleGrouping("spout");
builder.setBolt("count", new WordCount(), 12).fieldsGrouping("split",
new Fields("word"));
```

Table 2 A DAG computational graph is constructed with Apache Flink. The API definition of Apache Flink is more “data-oriented,” so it is higher-level.

```
StreamExecutionEnvironment env =
StreamExecutionEnvironment.getExecutionEnvironment();

DataStream<String> text = env.readTextFile ("input");
DataStream<Tuple2<String, Integer>> counts = text.flatMap(new
Tokenizer()).keyBy(0).sum(1);
counts.writeAsText("output");
```

The DAG graphs represent the computational logic of the stream processing, so most of the APIs are designed around building this computational logic graph. Table 1 shows the WordCount example for Apache Storm, which was very popular a few years ago.

In Apache Storm, add Spout or Bolt operators in the graph and specify the connection method between the operators. This way, after building the entire graph, submit it to run in a remote or local cluster.

In contrast, although the Apache Flink API is also building a computational logic graph, the API definition of Flink is more oriented to the data processing logic. It abstracts data streams into an infinite set, defines a group of operations on the set, and then automatically builds the corresponding DAG graph at the bottom layer.

Therefore, the Flink API is more high-level. Many researchers may prefer the high flexibility of Storm for their experiments because it is easier to ensure the expected graph structure. However, industry-wide preference is given to advanced APIs, such as the Flink API, because it is easier to use.

Flink DataStream API Overview

Based on the previous basic concepts of stream processing, this section describes how to use the Flink DataStream API in detail. Let’s start with a simple example. Table 3 is an example of a streaming WordCount. Although it only has five lines of code, it provides the basic structure for developing programs based on the Flink DataStream API.

Table 3 Example of WordCount based on the Flink DataStream API.

```
// 1. Set the runtime environment
StreamExecutionEnvironment env =
StreamExecutionEnvironment.getExecutionEnvironment();
// 2. Configure the data source to read data
DataStream<String> text = env.readTextFile ("input");
// 3. Perform a series of transformations
DataStream<Tuple2<String, Integer>> counts = text.flatMap(new
Tokenizer()).keyBy(0).sum(1);
// 4. Configure the Sink to write data out
counts.writeAsText("output");
// 5. Submit for execution
env.execute("Streaming WordCount");
```

To implement a streaming WordCount, first, obtain a `StreamExecutionEnvironment` object. It is the context object of building the graph. Add operators based on this object. For stream processing levels, create a data source to access the data. In this example, we use the built-in data source for reading files in the `Environment` object.

Then, obtain a `DataStream` object, which is an infinite dataset. Perform a sequence of operations on this dataset. For example, in the WordCount example, each record (that is, one row in the file) is first separated into words and implemented through the `FlatMap` operation.

Upon calling `FlatMap`, an operator adds to the underlying DAG graph. Then, to obtain a stream of words, group the words in the stream (`KeyBy`) and cumulatively compute the data of each word (`sum(1)`). The computed word data forms a new stream and is written into the output file.

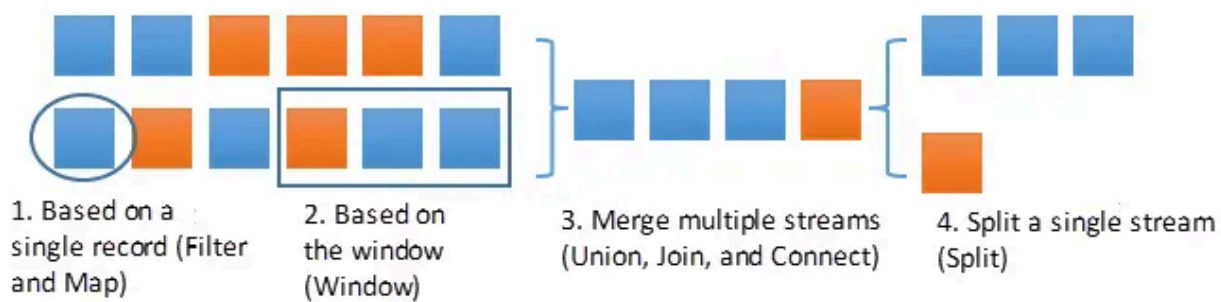
Finally, call the `env#execute` method to start the program execution. Ensure that none of the methods called earlier are processing data, rather build the DAG graph to express the computational logic.

Only after you build the entire graph and explicitly call the `Execute` method. The framework provides the computational graph to the cluster and accesses the data to execute the logic.

The example based on the streaming WordCount shows that it generally takes three steps to compile a stream processing program based on the Flink DataStream API: accessing, processing, and writing out the data.

Finally, explicitly call the `Execute` method; otherwise the logic will not be executed.

Operation overview



© Apache Flink Community China. Commercial use is strictly prohibited.

Figure 3. Flink DataStream operation overview.

As seen from the previous example, the core of the Flink DataStream API is the DataStream object that represents streaming data. The entire computational logic graph is built on the basis of calling different operations on the DataStream object to generate new DataStream objects.

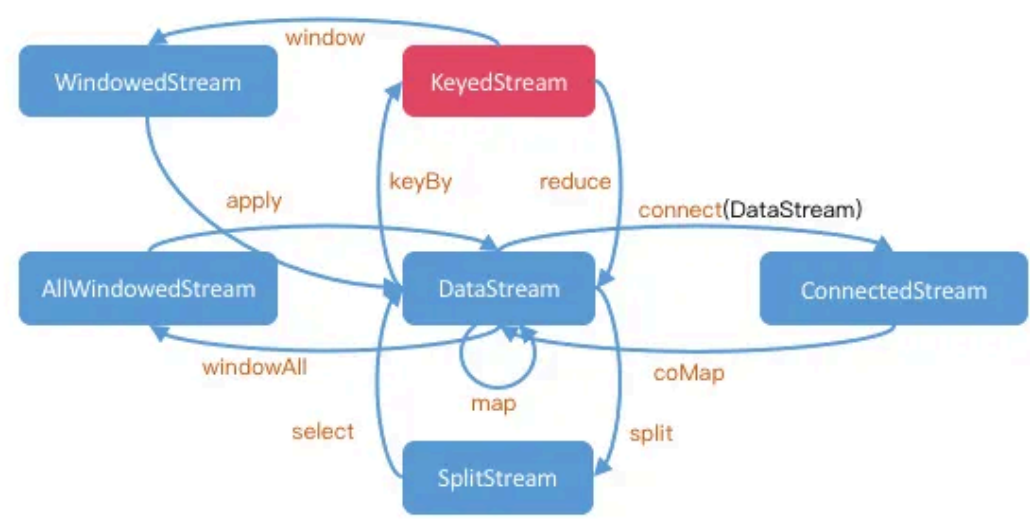
In general, operations on the DataStream are of four types. The first type is a single record operation, such as filtering out undesirable records (Filter operation) or converting each record (Map operation). The second type is the multiple records operation. For example, to count the total volume of orders within an hour, add all order records within an hour. To support this type of operation, one must join the required records through the Window for processing.

The third type is to operate multiple streams and transform them into a single stream. For example, multiple streams can be merged through operations, such as Union, Join, or Connect. These operations merge streams using different logics, but they eventually generate a new unified stream, thus allowing some cross-stream operations.

The fourth type is 'split operations,' which is supported by DataStream and opposed to Merge operations. These operations split a stream into multiple ones by rule with each split stream being a subset of the previous one, so that different streams are processed differently.



Basic transformation of DataStream



© Apache Flink Community China. Commercial use is strictly prohibited.

Figure 4. Different types of DataStream sub-types. Different sub-types support different sets of operations.

Medium

Search

Write



stream types to indicate the intermediate stream dataset types. Figure 4 shows the complete type of conversion relationship.

For single record operations such as Map, the results are the DataStream type. The Split operation generates a SplitStream. Based on the SplitStream, use the Select method to filter the desirable records and obtain a basic stream.

Similarly, for the Connect operation, obtain a dedicated ConnectedStream after calling StreamA.connect(StreamB). The operations supported by ConnectedStream are different from those supported by common DataStream.

This is the result of merging two different streams; it allows us to specify different processing logics of the two-stream records, and the processed results form a new DataStream stream. Process different records in the same operator so that they share state information during the process. Some Join operations in the upper layer need to be implemented through the Connect operations in the lower layer.

Also, we can split the stream by time or number into smaller groups through the Window operation. Select the specific splitting logic. When all records in a group arrive, obtain all records and perform Traverse or Sum operations. Therefore, you get a set of output data by processing each group, and all the output data forms a new basic stream.

For a common DataStream, use the allWindow operation, which represents a unified Window processing for the entire stream. Therefore, you can't use multiple operator instances for simultaneous computation. To solve this problem, use the KeyBy method to group records by Key first. Then perform

separate Window operations on records corresponding to different keys in parallel.

The KeyBy operation is one of the most important and commonly used operations. It is described in more detail below,

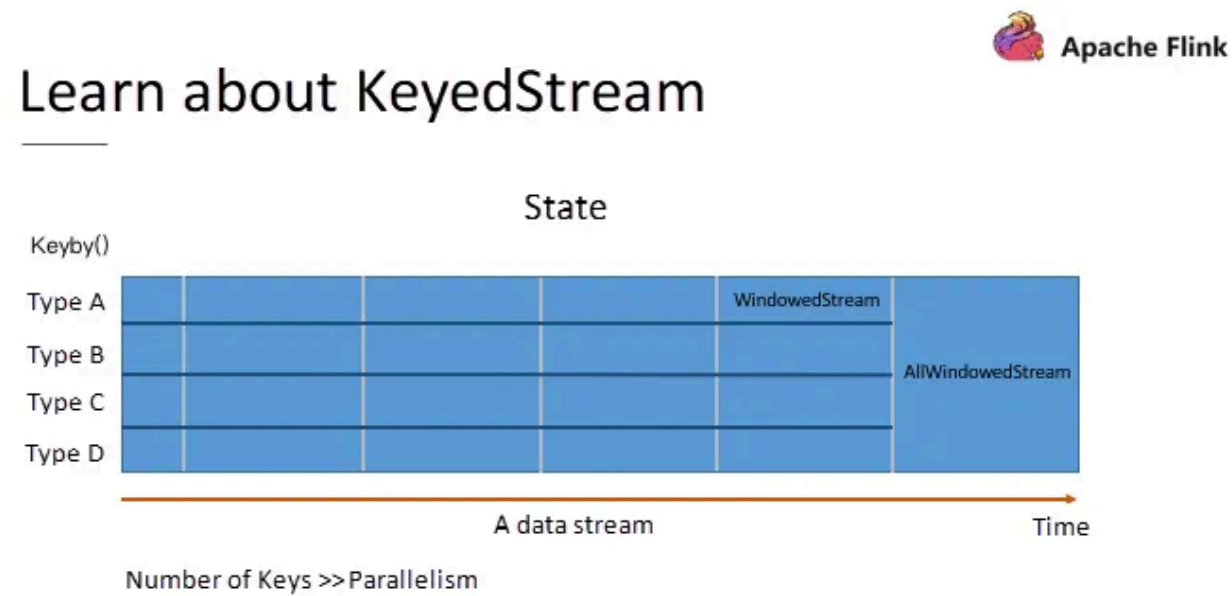


Figure 5. The comparison between the Window operation on the basic stream and the KeyedStream.

The Window operation on the KeyedStream makes it possible to use multiple instances for concurrent processing. Figure 5 shows the comparison between the allWindow operation on the basic DataStream object and the Window operation on the KeyedStream object. To process data simultaneously on multiple concurrent instances, group the data through the KeyBy operation.

Both KeyBy and Window Operations group data, but KeyBy splits the stream in a horizontal direction, while Window splits the stream in a vertical direction.

After splitting data with KeyBy, each subsequent operator instance can process the data corresponding to a specific Key set. Besides, Flink allows operators to maintain certain states. The states of operators on the KeyedStream are stored in a distributed manner.

KeyBy is a definite data allocation method (the following section introduces other allocation methods). If you restart a failed job, and the parallelism is changed, Flink reallocates Key groups and ensures that the group processing a certain Key must contain the state of the Key, thus ensuring consistency.

Finally, note that the KeyBy operation works only when the number of Keys exceeds the number of concurrent instances of the operator. All the data corresponding to the same Key is sent to the same instance, so if the number of Keys is less than the number of instances, some instances may not receive data, resulting in underutilized computing power.

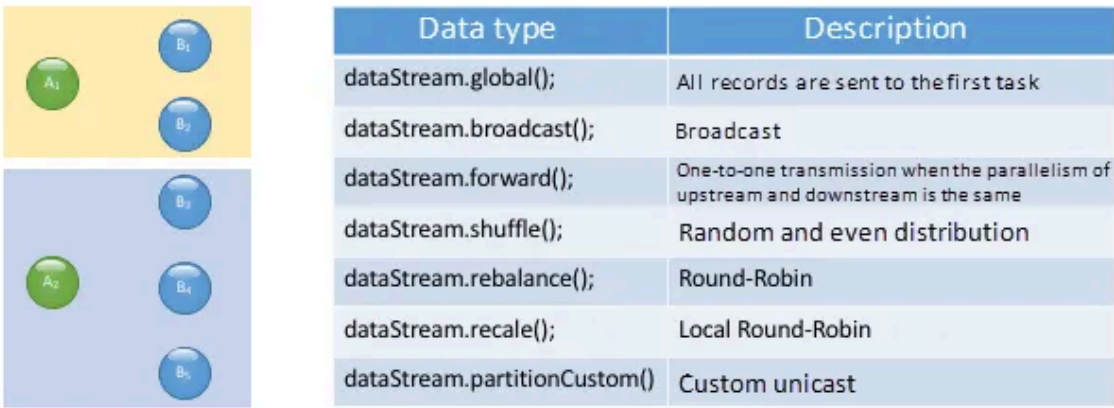
Additional Problems

In addition to KeyBy, Flink supports other physical grouping methods when exchanging data between the operators. As shown in Figure 1, the physical grouping methods in Flink DataStream include:

- Global: An upstream operator sends all records to the first instance of the downstream operator.
- Broadcast: An upstream operator sends each record to all instances of the downstream operator.
- Forward: Each upstream operator instance sends records to the corresponding instance of the downstream operator. This method is only applicable when the number of instances of the upstream operator is the same as the downstream operator.
- Shuffle: An upstream operator randomly selects a downstream operator for each record.
- Rebalance: An upstream operator sends data on a round-robin basis.
- Rescale: When the number of instances of the upstream and downstream operators is 'n' and 'm' respectively, if 'n' < 'm,' each upstream instance sends data to ceil(m/n) or floor(m/n) downstream instances on a round-robin basis. If 'n' > 'm,' floor(n/m) or ceil(n/m) upstream instances send data to downstream instances on a round-robin basis.
- PartitionCustomer: When the built-in allocation method does not meet your needs, choose to customize a grouping method.



Physical grouping



© Apache Flink Community China. Commercial use is strictly prohibited.

Figure 6. Other physical grouping methods except for KeyBy.

In addition to the grouping method, another important concept in the Flink DataStream API is the system type.

As shown in Figure 7, Flink DataStream objects are strongly-typed. For each DataStream object, the type of element needs to be specified. The

underlying serialization mechanism of Flink relies on this information to optimize serialization. Specifically, at the bottom layer of Flink, it uses a `TypeInfo` object to describe a type. The `TypeInfo` object defines a string of type-related information for the serialization framework to use.

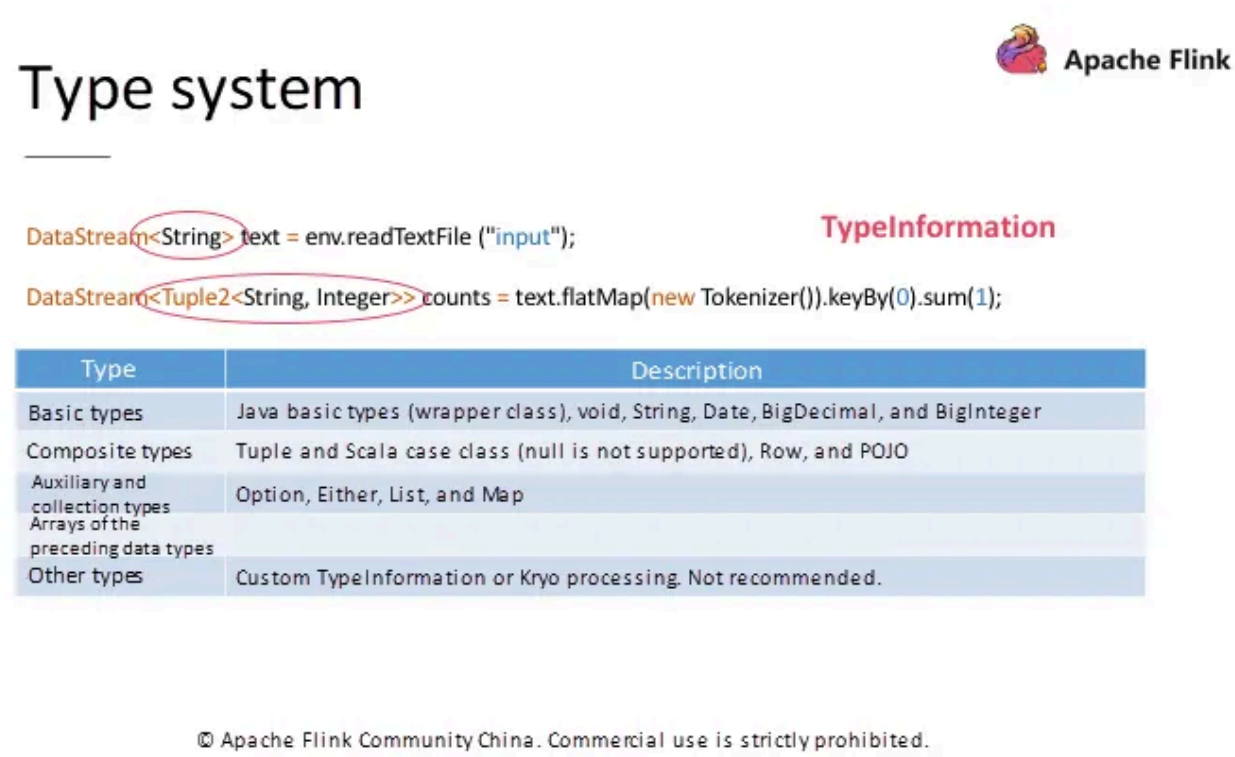


Figure 7. The type system in Flink DataStream API.

Flink has some commonly used built-in basic types. For these, Flink also provides their type information, which can be used directly without additional declarations. Flink can identify the corresponding types through the type inference mechanism. However, there are exceptions.

For example, the Flink DataStream API supports both Java and Scala. Many Scala APIs pass type information through implicit parameters, so if you need to call a Scala API through Java, you must pass the type information through implicit parameters. Another example is the Java erasure of the generic type. If the stream type is a generic one, you may not need to infer the type of information after the erasure. In this case, the type of information also needs to be explicitly specified.

In Flink, Java APIs generally use the `Tuple` type to combine multiple fields, while Scala APIs use the `Row` or `Case Class` types more often. Compared to the `Row` type, the `Tuple` type has two restrictions: the number of fields cannot exceed 25, and null values are not allowed in all fields.

Finally, Flink allows you to customize new types, `TypeInfo`, and use `Kryo` for serialization. However, this may cause some migration problems. Therefore, we recommend avoiding custom types.

Example

Let’s look at a more complicated example. Assuming that there is a data source that monitors orders in the system. When placing a new order, it uses

Tuple2 to output the type and transaction volume of items in the order.
Then, count the transaction volume of all types of items in real-time.

Table 4 An example of real-time order statistics:

```

public class GroupedProcessingTimeWindowSample {
    private static class DataSource extends
RichParallelSourceFunction<Tuple2<String, Integer>> {
        private volatile boolean isRunning = true;

        @Override
        public void run(SourceContext<Tuple2<String, Integer>> ctx)
throws Exception {
            Random random = new Random();
            while (isRunning) {

Thread.sleep((getRuntimeContext().getIndexOfThisSubtask() + 1) * 1000
* 5);
                String key = "    " + (char) ('A' +
random.nextInt(3));
                int value = random.nextInt(10) + 1;

                System.out.println(String.format("Emits\t(%s, %d)",
key, value));
                ctx.collect(new Tuple2<>(key, value));
            }
        }

        @Override
        public void cancel() {
            isRunning = false;
        }
    }

    public static void main(String[] args) throws Exception {
        StreamExecutionEnvironment env =
StreamExecutionEnvironment.getExecutionEnvironment();
        env.setParallelism(2);

        DataStream<Tuple2<String, Integer>> ds = env.addSource(new
DataSource());
        KeyedStream<Tuple2<String, Integer>, Tuple> keyedStream =
ds.keyBy(0);

        keyedStream.sum(1).keyBy(new KeySelector<Tuple2<String,
Integer>, Object>() {
            @Override
            public Object getKey(Tuple2<String, Integer>
stringIntegerTuple2) throws Exception {
                return "";
            }
        }).fold(new HashMap<String, Integer>(), new
FoldFunction<Tuple2<String, Integer>, HashMap<String, Integer>>() {
            @Override
            public HashMap<String, Integer> fold(HashMap<String,
Integer> accumulator, Tuple2<String, Integer> value) throws Exception
{
                accumulator.put(value.f0, value.f1);
                return accumulator;
            }
        }).addSink(new SinkFunction<HashMap<String, Integer>>() {
            @Override
            public void invoke(HashMap<String, Integer> value,
Context context) throws Exception {
                //
                System.out.println(value);
                //
            }
        });

        System.out.println(value.values().stream().mapToInt(v -> v).sum());
    }

    env.execute();
}

```

Table 4 shows the implementation of the example. Here, implement a simulated data source, which inherits from `RichParallelSourceFunction`. `RichParallelSourceFunction` is the API for `SourceFunction` with multiple instances.

Implement two methods: the `Run` method and the `Cancel` method. Flink calls the `Run` method directly to the source at runtime. It needs to continuously output data to form an initial stream. When implementing the `Run` method, records of item types and transaction volumes are randomly generated and then sent by using the `ctx#collect` method. Use the `Cancel` method when Flink needs to cancel a source task used with a `Volatile` variable to mark and control the execution state.

Next, start building the graph in the `Main` method. First, create a `StreamExecutionEnvironment` object. The `getExecutionEnvironment` method called to create the object automatically determines the environment, thus creating an appropriate object. For example, if we right-click and run the method in the IDE, we create a `LocalStreamExecutionEnvironment` object.

When it is run in an actual environment, it creates a `RemoteStreamExecutionEnvironment` object. Based on the `Environment` object, create a source to obtain the initial stream. Then, to count the transaction volume of each item type, use `KeyBy` to group the input streams through the first field (item type) of `Tuple` and then sum the values in the second field (transaction volume) of the record corresponding to each `Key`.

At the bottom layer, the `Sum` operator uses the `State` method to maintain the transaction volume sum corresponding to each `Key` (item type). When a new record arrives, the `Sum` operator updates the maintained volume sum and outputs a record of .

If you only count the type volume, the program ends here. Add a `Sink` operator directly after the `Sum` operator to output the continuously updated transaction volume of each item type. However, to count the total transaction volume of all types, output all records of the same compute node.

Use `KeyBy` to return the same `Key` for all records and group them together to send all records to the same instance.

Then, use the `Fold` method to maintain the volume of each item type in the operator. Note that although the `Fold` method has been marked as `Deprecated`, no other operations in the `DataStream API` can be used to replace it today. Therefore, this method receives an initial value.

Next, when each record in the subsequent stream arrives, the operator calls the FoldFunction that is passed to update the initial value and sends the updated value.

Use a HashMap to maintain the current transaction volume of each item type. When a new arrives, update the HashMap. This way, through Sink, you receive the HashMap of the latest item type and transaction volume, rely on this value to output the transaction volume of each item and the total transaction volume.

This example demonstrates the usage of the DataStream API. You can write it more efficiently. The higher-level Table and SQL also support the retraction mechanism, which deals with this situation even better.

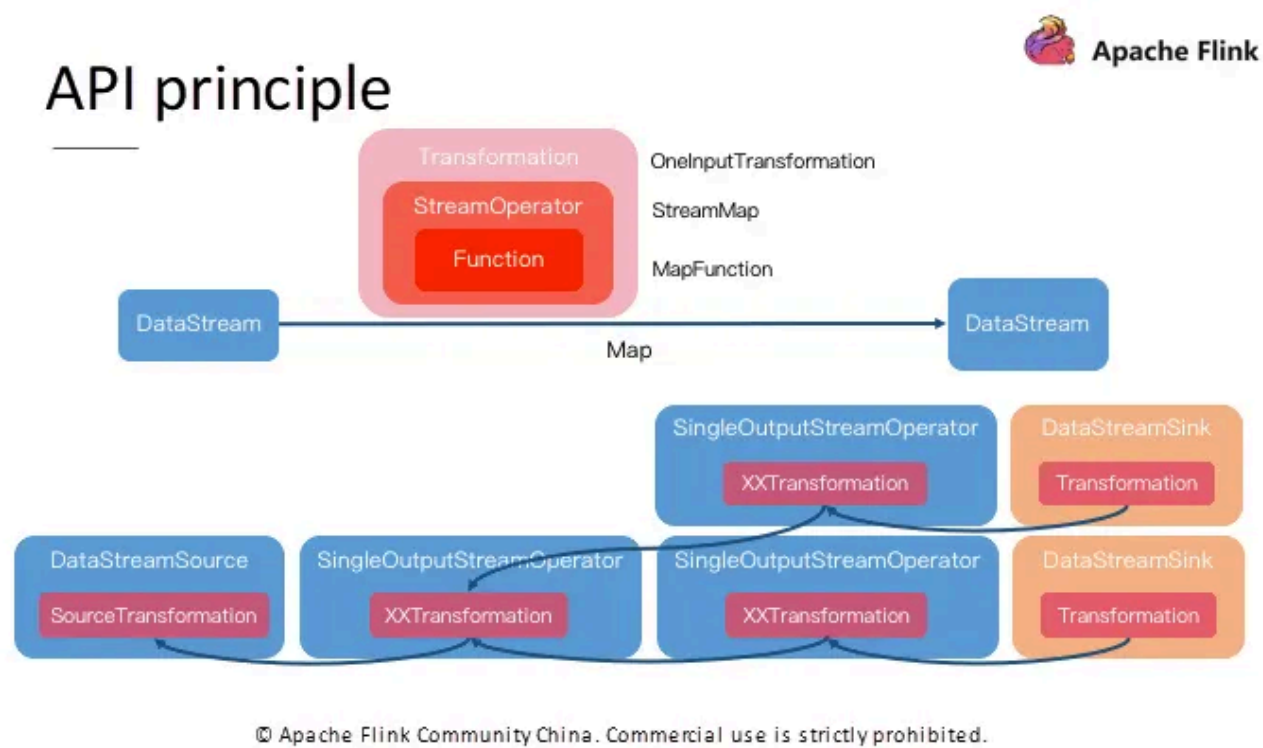


Figure 8. API schematic.

Finally, let's take a look at the DataStream API principle. Upon calling the `DataStream#map` algorithm, Flink creates a Transformation object at the bottom layer. This object represents a node in the computational logic graph. It records the MapFunction, which is the User-defined Function (UDF).

By using more methods, create more DataStream objects. Each object has a Transformation object; these objects form a graph structure based on the computing dependencies.

This is the computational graph. After that, Flink further transforms the graph structure to finally generate the JobGraph required for submitting the job.

Summary

This article introduces the Flink DataStream API, which is a lower-level Flink API. In actual development, you need to use some concepts yourself based on the API, such as State and Time, which requires a lot of work. Subsequent

courses will also introduce more upper-level Table and SQL APIs. In the future, Table and SQL may become mainstream Flink APIs.

However, a lower-level API creates a more powerful expression ability. In some cases where fine-grained operations are required, the DataStream API is required.


Original Source:

Basic Apache Flink Tutorial: DataStream API Programming

Apache Flink Community China December 25, 2019 137 By Cui Xingcan, an external committer and collated by Gao Yun This...

www.alibabacloud.com

- Big Data
- Alibaba
- Alibabacloud
- Java
- Apache



Written by Alibaba Cloud


5.9K followers · 25 following

Follow me to keep abreast with the latest technology news, industry insights, and developer trends. Alibaba Cloud website:<https://www.alibabacloud.com>

Follow

No responses yet





Iury Miguel

What are your thoughts?

More from Alibaba Cloud




 Alibaba Cloud

Analysis of UDP packet loss problem in Linux system

Recent work encountered a server application UDP packet loss, in the process ...



Jan 21, 2022  7  




 Alibaba Cloud

Memory Model and Synchronization Primitive—Part ...


Part 1 of this 2-part series explains Memory Barrier and its associated functions in depth.

May 18, 2021  10  


 Alibaba Cloud

PL/SQL Developer usage skills and shortcut keys

Oracle case: PL/SQL Developer usage skills, shortcut keys

Jan 21, 2022  7  



 Alibaba Cloud

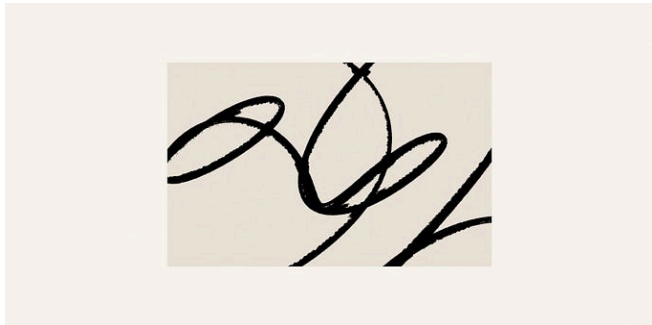
Introduction to HTML Space Notation `nbsp`; `ensp`; `emsp`;

This article introduces the HTML space symbol `nbsp`; `ensp`; `emsp`; and how to achie...

Jan 25, 2022  11  

See all from Alibaba Cloud


Recommended from Medium

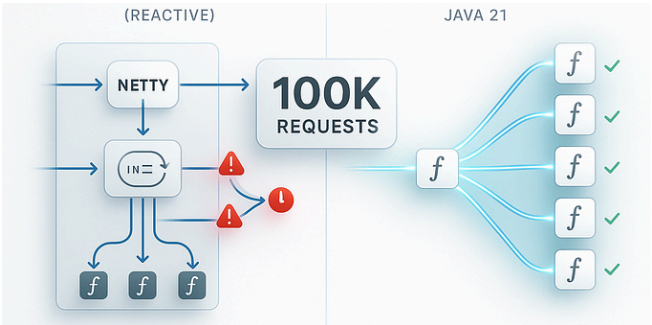



 Dinesh Arney

Designing High-Throughput Spring Boot Microservices (5000+ QPS)

High-throughput microservices (think 5000+ QPS) demand careful design choices. In this...

May 21  2  

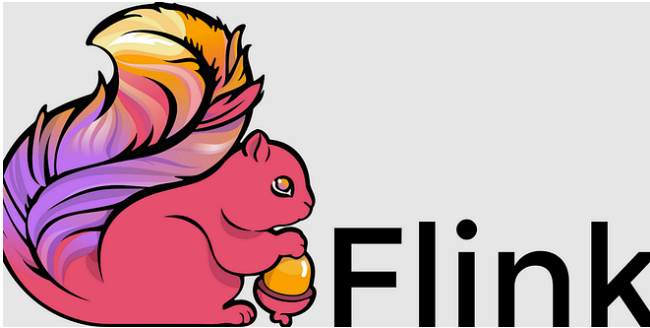


 Concurrent Mind

WebFlux vs Virtual Threads: I Hit Both With 100K Requests—One...

I did what any backend developer with scars would do.

 Jul 25  276  9  

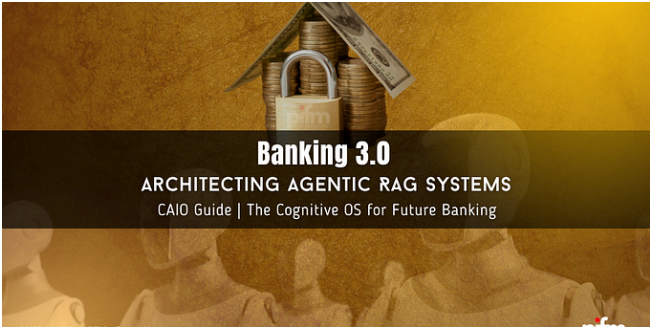



 In Yugen.ai Technology Blog by Yugen.ai

Real Time Fraud Detection Using Apache Flink—Part 2

Detect suspicious transactions using complex event patterns

Feb 23  19  




 In Plan It For Me (PIFM) by Ajit Mishra

Banking 3.0 | Architecting Agentic RAG Systems—The Cognitive OS...

From Generation to Autonomy to Resoning and Cognition—How Agentic AI Redefines...

 4d ago  293  1  

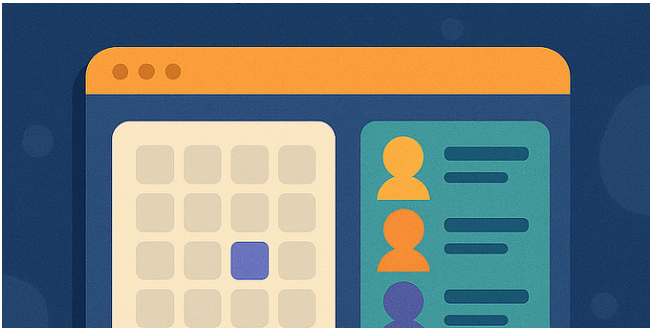



 In Coding Odyssey by Shivam Srivastava

RSocket with Spring Boot: Deep Dive with Interview Questions

A Comprehensive Guide

 Feb 22  13  



 Ankit Kumar Srivastava

Design a Collaborative Meeting Scheduler

System requirements :

 Jul 23  10  

See more recommendations