Cloudera Streaming Analytics

Application Development

Date published: 2019-12-17
Date modified: 2019-12-17

https://docs.cloudera.com/

Legal Notice

Contents

Flink Quickstart Archetype

The Quickstart Archetype serves as a template for a Flink streaming application. You can use the Archetype to add source, sink and computation to the template. Like this you can practice the development of a simple Flink application, or use the Archetype as the starting point for a more complex application including state,watermark and checkpoint.

About this task

The Flink quickstart archetype can be used to quickly build a basic Flink streaming project.

Note: You need to install the archetype locally on your host as Cloudera does not release maven archetypes to the Maven Central Repository.

Procedure

1. Perform the following commands to create the archetype locally:

```
git clone https://github.com/cloudera/flink-tutorials
cd flink-tutorials
cd flink-quickstart-archetype
mvn clean install
cd ..
```

   The following entry should be seen in your local catalog:

```
cat ~/.m2/repository/archetype-catalog.xml
 ...
 <archetypes>
    <archetype>
      <groupId>com.cloudera.flink</groupId>
      <artifactId>flink-quickstart-archetype</artifactId>
      <version>1.10.0-csa1.2.0.0</version>
      <description>flink-quickstart-archetype</description>
    </archetype>
  </archetypes>
```

2. Generate the project skeleton with the following commands:

```
mvn archetype:generate \
      -DarchetypeGroupId=com.cloudera.flink \
      -DarchetypeArtifactId=flink-quickstart-archetype \
      -DarchetypeVersion=1.10.0-csa1.2.0.0
```

3. Provide basic information about the streaming application.

   You can choose to customize configurations yourself or use automatically generated information.

   - To use automatically generated information, press Enter.
   - If you choose to customize configuration, set the following properties:

```
Define value for property 'groupId': com.cloudera.flink
                                     Define value for property 'artifactId':
sample-project
                                     Define value for property 'version' 1.0-
SNAPSHOT: :
                                     Define value for property 'package' c
om.cloudera.flink: :
                                     Confirm properties configuration:
```

```
                                    groupId: com.cloudera.flink
                                    artifactId: sample-project
                                    version: 1.0-SNAPSHOT
                                    package: com.cloudera.flink
                                    Y: :
```

The generated project will look like this:

```
sample-project
### pom.xml
### src
    ### main
        ### java
        #   ### com
        #       ### cloudera
        #           ### flink
        #               ### StreamingJob.java
        ### resources
            ### log4j.properties
```

4. Open StreamingJob.java file.

   The archetype application will look like this:

```
public class StreamingJob {
 public static void main(String[] args) throws Exception {
  final StreamExecutionEnvironment env = StreamExecutionEnvironment.getEx
ecutionEnvironment();
  DataStream<Integer> ds = env.fromElements(1,2,3,4);
  ds.printToErr();
  env.execute("Flink Streaming Java API Skeleton");
 }
}
```

5. Customize the archetype application by adding source, stream transformation and sink to the Datastream class.
6. Execute the application with env.execute command.

Results

You have built your Flink streaming project.

What to do next

You can further develop your application, or you can run the Flink application archetype.

Related Information

Flink streaming application structure

Run a Flink job

Flink streaming application structure

You must understand the parts of application structure to build and develop a Flink streaming application. This design is implemented as the core logic with the execution environment, source, and sink.

A Flink application consists of the following structural parts:

- Application main class
- Data sources
- Processing operators
- Data sinks

The application main class defines the execution environment and creates the data pipeline. The data pipeline is the business logic of a Flink application where one or more operators are chained together. These processing operators apply transformations on the input data that comes from the data sources. After the transformation, the application forwards the transformed data to the data sinks.

For StreamExecutionEnvironment

StreamExecutionEnvironment class is needed to create DataStream and to configure important job parameters for maintaining the behavior of the application. The rest of the main class defines the application sources, processing flow and the sinks followed by the execute() call. The execute call triggers the actual execution of the pipeline either locally or on the cluster. The getExecutionEnvironment() static call guarantees that the pipeline always uses the correct environment based on the location it is executed on. When running from the IDE, a local execution environment, and when running from the client for cluster submission, it returns the YARN execution environment. The rest of the main class defines the application sources, processing flow and the sinks followed by the execute() call. The execute call triggers the actual execution of the pipeline either locally or on the cluster.

For Datastream

A DataStream represents the data records, and operators can be used to apply transformations that create new Data Streams. There are pre-implemented sources and sinks for Flink, and you can also use custom defined connectors to maintain the dataflow with other functions. Choosing the sources and sinks depends on the purpose of the application. As Flink can be implemented in any kind of an environment, various connectors are available. In most cases, Kafka is used as a connector as it has streaming capabilities and can be easily integrated with other services.

The following is an example of a Flink application logic:

```
public class KafkaToHDFSAvroJob {
 private static Logger LOG = LoggerFactory.getLogger(KafkaToHDFSAvroJob.cl
ass);
 public static void main(String[] args) throws Exception {
  ParameterTool params = Utils.parseArgs(args);

  StreamExecutionEnvironment env = StreamExecutionEnvironment.getExecution
Environment();

  KafkaDeserializationSchema<Message> schema = ClouderaRegistryKafkaDeserial
izationSchema
    .builder(Message.class)
    .setConfig(Utils.readSchemaRegistryProperties(params))
    .build();
  FlinkKafkaConsumer<Message> consumer = new FlinkKafkaConsumer<Message>(
params.getRequired(K_KAFKA_TOPIC), schema, Utils.readKafkaProperties(params)
);

  DataStream<String> source = env.addSource(consumer)
    .name("Kafka Source")
    .uid("Kafka Source")
    .map(record -> record.getId() + "," + record.getName() + "," + record.g
etDescription())
    .name("ToOutputString");
  StreamingFileSink<String> sink = StreamingFileSink
    .forRowFormat(new Path(params.getRequired(K_HDFS_OUTPUT)), new Simple
StringEncoder<String>("UTF-8"))
    .build();
  source.addSink(sink)
    .name("FS Sink")
    .uid("FS Sink");
  source.print();

  env.execute("Flink Streaming Secured Job Sample");
 }
```

```
}
```

Related Information

Simple Tutorial: Application logic

Stateful Tutorial: Build a Flink streaming application

Testing and validating Flink applications

After you have built your Flink streaming application, you can create a simple testing method to validate the correct behaviour of your application.

Pipelines can be extracted to static methods and can be easily tested with the JUnit framework.

A simple JUnit test can be written to verify the core application logic. The test is implemented in the test class and should be regarded as an integration test of the application flow.

The test mimics the application main class with only minor differences:

1. Create the StreamExecutionEnvironment the same way.
2. Use the env.fromElements(..) method to pre-populate a DataStream with some testing data.
3. Feed the testing data to the static data processing logic as before.
4. Verify the correctness once the test is finished.

```
    @Test
    public void testPipeline() throws Exception {
        final String alertMask = "42";
        StreamExecutionEnvironment env = StreamExecutionEnvironment.getExec
utionEnvironment();
        HeapMetrics alert1 = testStats(0.42);
        HeapMetrics regular1 = testStats(0.452);
        HeapMetrics regular2 = testStats(0.245);
        HeapMetrics alert2 = testStats(0.9423);

        DataStreamSource<HeapMetrics> testInput = env.fromElements(alert1,
 alert2, regular1, regular2);
        HeapMonitorPipeline.computeHeapAlerts(testInput, ParameterTool.fro
mArgs(new String[]{"--alertMask", alertMask}))
                .addSink(new SinkFunction<HeapAlert>() {
                    @Override
                    public void invoke(HeapAlert value) {
                        testOutput.add(value);
                    }
                })
                .setParallelism(1);
        env.execute();
        assertEquals(Sets.newHashSet(HeapAlert.maskRatioMatch(alertMask, al
ert1),
                HeapAlert.maskRatioMatch(alertMask, alert2)), testOutput);
    }
    private HeapMetrics testStats(double ratio) {
        return new HeapMetrics(HeapMetrics.OLD_GEN, 0, 0, ratio, 0, "test
host");
    }
}
```

Related Information

Simple Tutorial: Testing the data pipeline

Stateful Tutorial: Test and validate the streaming pipeline

Stream windowing in Flink applications

Within the application logic, you can add stream windowing after deciding the type of grouping on the unbounded data records. This way you can apply different computations on the windowed data which enables you to create more complex streaming applications.

The architecture of the windowing code depends on the type of stream, keyed or not, and the type of window assigner.

```
// If needed we create a window computation of the transaction summaries by
item and time window
  if (params.getBoolean(ENABLE_SUMMARIES_KEY, false)) {
   DataStream<TransactionSummary> transactionSummaryStream = processedTrans
actions
     .keyBy("transaction.itemId")
     .timeWindow(Time.minutes(10))
     .aggregate(new TransactionSummaryAggregator())
     .name("Create Transaction Summary")
     .uid("Create Transaction Summary")
     .filter(new SummaryAlertingCondition(params))
     .name("Filter High failure rate");

   writeTransactionSummaries(params, transactionSummaryStream);
  }

  return env;
```

DataStream Class	Definiton
.keyBy	Defining key for keyed streams.
.timeWindow	Type of windowing method with value.
.aggregate	Data transformation type that is applied on the windowed data.
.uid	Defining ID for the operator.
.filter	Data transformation type that is applied on the windowed data.

To develop the windowing and to configure parameters, see the example of Apache Flink documentation.

Related Information

Stateful Tutorial: Creating windowed summaries