

Apache Flink单元测试编写指南

ApacheHudi 2020-02-18

以下文章来源于Flink，作者Kartik Khare



Flink

Apache Flink & Apache Hudi. 由Committer & PMC 维护。

Writing unit tests is one of the essential tasks of designing a production-grade application. Without tests, a single change in code can result in cascades of failure in production. Thus unit tests should be written for all types of applications, be it a simple job cleaning data and training a model or a complex multi-tenant, real-time data processing system. In the following sections, we provide a guide for unit testing of Apache Flink applications. Apache Flink provides a robust unit testing framework to make sure your applications behave in production as expected during development. You need to include the following dependencies to utilize the provided framework.

```
<dependency>
  <groupId>org.apache.flink</groupId>
  <artifactId>flink-test-utils_${scala.binary.version}</artifactId>
  <version>${flink.version}</version>
  <scope>test</scope>
</dependency>
<dependency>
  <groupId>org.apache.flink</groupId>
  <artifactId>flink-runtime_2.11</artifactId>
  <version>1.9.0</version>
  <scope>test</scope>
  <classifier>tests</classifier>
</dependency>
<dependency>
  <groupId>org.apache.flink</groupId>
  <artifactId>flink-streaming-java_2.11</artifactId>
  <version>1.9.0</version>
  <scope>test</scope>
  <classifier>tests</classifier>
</dependency>
```



The strategy of writing unit tests differs for various operators. You can break down the strategy into the following three buckets:

- Stateless Operators
- Stateful Operators
- Timed Process Operators

Stateless Operators

Writing unit tests for a stateless operator is a breeze. You need to follow the basic norm of writing a test case, i.e., create an instance of the function class and test the appropriate methods. Let's take an example of a simple Map operator.

```
public class MyStatelessMap implements MapFunction<String, String> {  
    @Override  
    public String map(String in) throws Exception {  
        String out = "hello " + in;  
        return out;  
    }  
}
```



The test case for the above operator should look like

```
@Test  
public void testMap() throws Exception {  
    MyStatelessMap statelessMap = new MyStatelessMap();  
    String out = statelessMap.map("world");  
    Assert.assertEquals("hello world", out);  
}
```



Pretty simple, right? Let's take a look at one for the FlatMap operator.

```
public class MyStatelessFlatMap implements FlatMapFunction<String, String> {  
    @Override  
    public void flatMap(String in, Collector<String> collector) throws Exception {  
        String out = "hello " + in;  
        collector.collect(out);  
    }  
}
```



FlatMap operators require a Collector object along with the input. For the test case, we have two options:

1. Mock the Collector object using Mockito
2. Use the ListCollector provided by Flink

I prefer the second method as it requires fewer lines of code and is suitable for most of the cases.

```
@Test  
public void testFlatMap() throws Exception {  
    MyStatelessFlatMap statelessFlatMap = new MyStatelessFlatMap();  
    List<String> out = new ArrayList<>();  
    ListCollector<String> listCollector = new ListCollector<>(out);  
    statelessFlatMap.flatMap("world", listCollector);  
    Assert.assertEquals(Lists.newArrayList("hello world"), out);  
}
```



Stateful Operators

Writing test cases for stateful operators requires more effort. You need to check whether the operator state is updated correctly and if it is cleaned up properly along with the output of the operator.

Let's take an example of stateful `FlatMap` function

```
public class StatefulFlatMap extends RichFlatMapFunction<String, String> {
    ValueState<String> previousInput;

    @Override
    public void open(Configuration parameters) throws Exception {
        previousInput = getRuntimeContext().getState(
            new ValueStateDescriptor<String>("previousInput", Types.STRING));
    }

    @Override
    public void flatMap(String in, Collector<String> collector) throws Exception {
        String out = "hello " + in;
        if(previousInput.value() != null){
            out = out + " " + previousInput.value();
        }
        previousInput.update(in);
        collector.collect(out);
    }
}
```



The intricate part of writing tests for the above class is to mock the configuration as well as the runtime context of the application. Flink provides `TestHarness` classes so that users don't have to create the mock objects themselves. Using the `KeyedOperatorHarness`, the test looks like:

```

import org.apache.flink.streaming.api.operators.StreamFlatMap;
import org.apache.flink.streaming.runtime.streamrecord.StreamRecord;
import org.apache.flink.streaming.util.KeyedOneInputStreamOperatorTestHarness;
import org.apache.flink.streaming.util.OneInputStreamOperatorTestHarness;

@Test
public void testFlatMap() throws Exception{
    StatefulFlatMap statefulFlatMap = new StatefulFlatMap();

    // OneInputStreamOperatorTestHarness takes the input and output types as type parameters
    OneInputStreamOperatorTestHarness<String, String> testHarness =
        // KeyedOneInputStreamOperatorTestHarness takes three arguments:
        // Flink operator object, key selector and key type
        new KeyedOneInputStreamOperatorTestHarness<>(
            new StreamFlatMap<>(statefulFlatMap), x -> "1", Types.STRING);
    testHarness.open();

    // test first record
    testHarness.processElement("world", 10);
    ValueState<String> previousInput =
        statefulFlatMap.getRuntimeContext().getState(
            new ValueStateDescriptor<>("previousInput", Types.STRING));
    String stateValue = previousInput.value();
    Assert.assertEquals(
        Lists.newArrayList(new StreamRecord<>("hello world", 10)),
        testHarness.extractOutputStreamRecords());
    Assert.assertEquals("world", stateValue);

    // test second record
    testHarness.processElement("parallel", 20);
    Assert.assertEquals(
        Lists.newArrayList(
            new StreamRecord<>("hello world", 10),
            new StreamRecord<>("hello parallel world", 20)),
        testHarness.extractOutputStreamRecords());
    Assert.assertEquals("parallel", previousInput.value());
}

```



The test harness provides many helper methods, three of which are being used here:

1. `open`: calls the `open` of the `FlatMap` function with relevant parameters. It also initializes the context.
2. `processElement`: allows users to pass an input element as well as the timestamp associated with the element.
3. `extractOutputStreamRecords`: gets the output records along with their timestamps from the `Collector`.


The test harness simplifies the unit testing for the stateful functions to a large extent.

You might also need to check whether the state value is being set correctly. You can get the state value directly from the operator using a mechanism similar to the one used while creating the state. This is also demonstrated in the previous example.

Timed Process Operators

Writing tests for process functions, that work with time, is quite similar to writing tests for stateful functions because you can also use test harness. However, you need to take care of another aspect, which is providing timestamps for events and controlling the current time of the application. By setting the current (processing or event) time, you can trigger registered timers, which will call the `onTimer` method of the function

```
public class MyProcessFunction extends KeyedProcessFunction<String, String, String> {  
    @Override  
    public void processElement(String in, Context context, Collector<String> collector) throws Exception {  
        context.timerService().registerProcessingTimeTimer(50);  
        String out = "hello " + in;  
        collector.collect(out);  
    }  
  
    @Override  
    public void onTimer(long timestamp, OnTimerContext ctx, Collector<String> out) throws Exception {  
        out.collect(String.format("Timer triggered at timestamp %d", timestamp));  
    }  
}
```



We need to test both the methods in the `KeyedProcessFunction`, i.e., `processElement` as well as `onTimer`. Using a test harness, we can control the current time of the function. Thus, we can trigger the timer at will rather than waiting for a specific time.

Let's take a look at the test case

```

@Test
public void testProcessElement() throws Exception{
    MyProcessFunction myProcessFunction = new MyProcessFunction();
    OneInputStreamOperatorTestHarness<String, String> testHarness =
        new KeyedOneInputStreamOperatorTestHarness<>(
            new KeyedProcessOperator<>(myProcessFunction), x -> "1", Types.STRING);

    // Function time is initialized to 0
    testHarness.open();
    testHarness.processElement("world", 10);

    Assert.assertEquals(
        Lists.newArrayList(new StreamRecord<>("hello world", 10)),
        testHarness.extractOutputStreamRecords());
}

@Test
public void testOnTimer() throws Exception {
    MyProcessFunction myProcessFunction = new MyProcessFunction();
    OneInputStreamOperatorTestHarness<String, String> testHarness =
        new KeyedOneInputStreamOperatorTestHarness<>(
            new KeyedProcessOperator<>(myProcessFunction), x -> "1", Types.STRING);

    testHarness.open();
    testHarness.processElement("world", 10);
    Assert.assertEquals(1, testHarness.numProcessingTimeTimers());

    // Function time is set to 50
    testHarness.setProcessingTime(50);
    Assert.assertEquals(
        Lists.newArrayList(
            new StreamRecord<>("hello world", 10),
            new StreamRecord<>("Timer triggered at timestamp 50")),
        testHarness.extractOutputStreamRecords());
}

```



The mechanism to test the multi-input stream operators such as CoProcess functions is similar to the ones described in this article. You should use the TwoInput variant of the harness for these operators, such as TwoInputStreamOperatorTestHarness.

Summary

In the previous sections we showcased how unit testing in Apache Flink works for stateless, stateful and times-aware-operators. We hope you found the steps easy to follow and execute while developing your Flink applications. If you have any questions or feedback you can reach out to me here or contact the community on the Apache Flink user mailing list.