dataArtisans

Apache Flink® Training

DataStream API Basics

DataStream API

- Stream Processing
- Java and Scala
- All examples here in Java for Flink 1.0
- Documentation available at

flink.apache.org

DataStream API by Example

Window WordCount: main Method

```
public static void main(String[] args) throws Exception {
    // set up the execution environment
    final StreamExecutionEnvironment env =
            StreamExecutionEnvironment.getExecutionEnvironment();
    // configure event time
    env.setStreamTimeCharacteristic(TimeCharacteristic.EventTime);

    DataStream<Tuple2<String, Integer>> counts = env
            // read stream of words from socket
            .socketTextStream("localhost", 9999)
            // split up the lines in tuples containing: (word,1)
            .flatMap(new Splitter())
            // key stream by the tuple field "0"
            .keyBy(0)
            // compute counts every 5 minutes
                .timeWindow(Time.minutes(5))
            //sum up tuple field "1"
            .sum(1);

    // print result in command line
    counts.print();
    // execute program
    env.execute("Socket WordCount Example");
}
```

Stream Execution Environment

```
public static void main(String[] args) throws Exception {
    // set up the execution environment
    final StreamExecutionEnvironment env =
            StreamExecutionEnvironment.getExecutionEnvironment();
    // configure event time
    env.setStreamTimeCharacteristic(TimeCharacteristic.EventTime);

    DataStream<Tuple2<String, Integer>> counts = env
            // read stream of words from socket
            .socketTextStream("localhost", 9999)
            // split up the lines in tuples containing: (word,1)
            .flatMap(new Splitter())
            // key stream by the tuple field "0"
            .keyBy(0)
            // compute counts every 5 minutes
                .timeWindow(Time.minutes(5))
            //sum up tuple field "1"
            .sum(1);

    // print result in command line
    counts.print();
    // execute program
    env.execute("Socket WordCount Example");
}
```

Data Sources

```
public static void main(String[] args) throws Exception {
    // set up the execution environment
    final StreamExecutionEnvironment env =
            StreamExecutionEnvironment.getExecutionEnvironment();
    // configure event time
    env.setStreamTimeCharacteristic(TimeCharacteristic.EventTime);

    DataStream<Tuple2<String, Integer>> counts = env
            // read stream of words from socket
            .socketTextStream("localhost", 9999)
            // split up the lines in tuples containing: (word,1)
            .flatMap(new Splitter())
            // key stream by the tuple field "0"
            .keyBy(0)
            // compute counts every 5 minutes
                .timeWindow(Time.minutes(5))
            //sum up tuple field "1"
            .sum(1);

    // print result in command line
    counts.print();
    // execute program
    env.execute("Socket WordCount Example");
}
```

Data Types

```
public static void main(String[] args) throws Exception {
    // set up the execution environment
    final StreamExecutionEnvironment env =
            StreamExecutionEnvironment.getExecutionEnvironment();
    // configure event time
    env.setStreamTimeCharacteristic(TimeCharacteristic.EventTime);

    DataStream<Tuple2<String, Integer>> counts = env
            // read stream of words from socket
            .socketTextStream("localhost", 9999)
            // split up the lines in tuples containing: (word,1)
            .flatMap(new Splitter())
            // key stream by the tuple field "0"
            .keyBy(0)
            // compute counts every 5 minutes
                .timeWindow(Time.minutes(5))
            //sum up tuple field "1"
            .sum(1);

    // print result in command line
    counts.print();
    // execute program
    env.execute("Socket WordCount Example");
}
```

Transformations

```
public static void main(String[] args) throws Exception {
    // set up the execution environment
    final StreamExecutionEnvironment env =
            StreamExecutionEnvironment.getExecutionEnvironment();
    // configure event time
    env.setStreamTimeCharacteristic(TimeCharacteristic.EventTime);

    DataStream<Tuple2<String, Integer>> counts = env
            // read stream of words from socket
            .socketTextStream("localhost", 9999)
            // split up the lines in tuples containing: (word,1)
            .flatMap(new Splitter())
            // key stream by the tuple field "0"
            .keyBy(0)
            // compute counts every 5 minutes
                .timeWindow(Time.minutes(5))
            //sum up tuple field "1"
            .sum(1);

    // print result in command line
    counts.print();
    // execute program
    env.execute("Socket WordCount Example");
}
```

User functions

```
public static void main(String[] args) throws Exception {
    // set up the execution environment
    final StreamExecutionEnvironment env =
            StreamExecutionEnvironment.getExecutionEnvironment();
    // configure event time
    env.setStreamTimeCharacteristic(TimeCharacteristic.EventTime);

    DataStream<Tuple2<String, Integer>> counts = env
            // read stream of words from socket
            .socketTextStream("localhost", 9999)
            // split up the lines in tuples containing: (word,1)
            .flatMap(new Splitter())
            // key stream by the tuple field "0"
            .keyBy(0)
            // compute counts every 5 minutes
                .timeWindow(Time.minutes(5))
            //sum up tuple field "1"
            .sum(1);

    // print result in command line
    counts.print();
    // execute program
    env.execute("Socket WordCount Example");
}
```

DataSinks

```
public static void main(String[] args) throws Exception {
    // set up the execution environment
    final StreamExecutionEnvironment env =
            StreamExecutionEnvironment.getExecutionEnvironment();
    // configure event time
    env.setStreamTimeCharacteristic(TimeCharacteristic.EventTime);

    DataStream<Tuple2<String, Integer>> counts = env
            // read stream of words from socket
            .socketTextStream("localhost", 9999)
            // split up the lines in tuples containing: (word,1)
            .flatMap(new Splitter())
            // key stream by the tuple field "0"
            .keyBy(0)
            // compute counts every 5 minutes
                .timeWindow(Time.minutes(5))
            //sum up tuple field "1"
            .sum(1);

    // print result in command line
    counts.print();
    // execute program
    env.execute("Socket WordCount Example");
}
```

Execute!

```
public static void main(String[] args) throws Exception {
    // set up the execution environment
    final StreamExecutionEnvironment env =
            StreamExecutionEnvironment.getExecutionEnvironment();
    // configure event time
    env.setStreamTimeCharacteristic(TimeCharacteristic.EventTime);

    DataStream<Tuple2<String, Integer>> counts = env
            // read stream of words from socket
            .socketTextStream("localhost", 9999)
            // split up the lines in tuples containing: (word,1)
            .flatMap(new Splitter())
            // key stream by the tuple field "0"
            .keyBy(0)
            // compute counts every 5 minutes
                .timeWindow(Time.minutes(5))
            //sum up tuple field "1"
            .sum(1);

    // print result in command line
    counts.print();
    // execute program
    env.execute("Socket WordCount Example");
}
```

Window WordCount: FlatMap

```
public static class Splitter
  implements FlatMapFunction<String, Tuple2<String, Integer>> {

  @Override
  public void flatMap(String value,
                      Collector<Tuple2<String, Integer>> out)
    throws Exception {
      // normalize and split the line
      String[] tokens = value.toLowerCase().split("\\W+");

      // emit the pairs
      for (String token : tokens) {
        if (token.length() > 0) {
          out.collect(
              new Tuple2<String, Integer>(token, 1));
        }
      }
    }
}
```

WordCount: Map: Interface

```
public static class Splitter
  implements FlatMapFunction<String, Tuple2<String, Integer>> {

  @Override
  public void flatMap(String value,
                      Collector<Tuple2<String, Integer>> out)
    throws Exception {
      // normalize and split the line
      String[] tokens = value.toLowerCase().split("\\W+");

      // emit the pairs
      for (String token : tokens) {
        if (token.length() > 0) {
          out.collect(
              new Tuple2<String, Integer>(token, 1));
        }
      }
    }
}
```

WordCount: Map: Types

```
public static class Splitter
  implements FlatMapFunction<String, Tuple2<String, Integer>> {

  @Override
  public void flatMap(String value,
                      Collector<Tuple2<String, Integer>> out)
    throws Exception {
      // normalize and split the line
      String[] tokens = value.toLowerCase().split("\\W+");

      // emit the pairs
      for (String token : tokens) {
        if (token.length() > 0) {
          out.collect(
              new Tuple2<String, Integer>(token, 1));
        }
      }
    }
}
```

WordCount: Map: Collector

```
public static class Splitter
  implements FlatMapFunction<String, Tuple2<String, Integer>> {

  @Override
  public void flatMap(String value,
                      Collector<Tuple2<String, Integer>> out)
    throws Exception {
      // normalize and split the line
      String[] tokens = value.toLowerCase().split("\\W+");

      // emit the pairs
      for (String token : tokens) {
        if (token.length() > 0) {
          out.collect(
              new Tuple2<String, Integer>(token, 1));
        }
      }
    }
}
```

DataStream API Concepts

(Selected) Data Types

- Basic Java Types
  - String, Long, Integer, Boolean,…
  - Arrays
- Composite Types
  - Tuples
  - Many more (covered in the advanced slides)

Tuples

- The easiest and most lightweight way of encapsulating data in Flink
- Tuple1 up to Tuple25

```
Tuple2<String, String> person = new Tuple2<>("Max", "Mustermann”);

Tuple3<String, String, Integer> person = new Tuple3<>("Max", "Mustermann", 42);

Tuple4<String, String, Integer, Boolean> person =
  new Tuple4<>("Max", "Mustermann", 42, true);

// zero based index!
String firstName = person.f0;
String secondName = person.f1;
Integer age = person.f2;
Boolean fired = person.f3;
```

Transformations: Map

```
DataStream<Integer> integers = env.fromElements(1, 2, 3, 4);

// Regular Map - Takes one element and produces one element
DataStream<Integer> doubleIntegers =
   integers.map(new MapFunction<Integer, Integer>() {
      @Override
      public Integer map(Integer value) {
        return value * 2;
      }
   });

doubleIntegers.print();
> 2, 4, 6, 8


// Flat Map - Takes one element and produces zero, one, or more elements.
DataStream<Integer> doubleIntegers2 =

   integers.flatMap(new FlatMapFunction<Integer, Integer>() {
        @Override
        public void flatMap(Integer value, Collector<Integer> out) {
          out.collect(value * 2);
        }
   });

doubleIntegers2.print();
> 2, 4, 6, 8
```

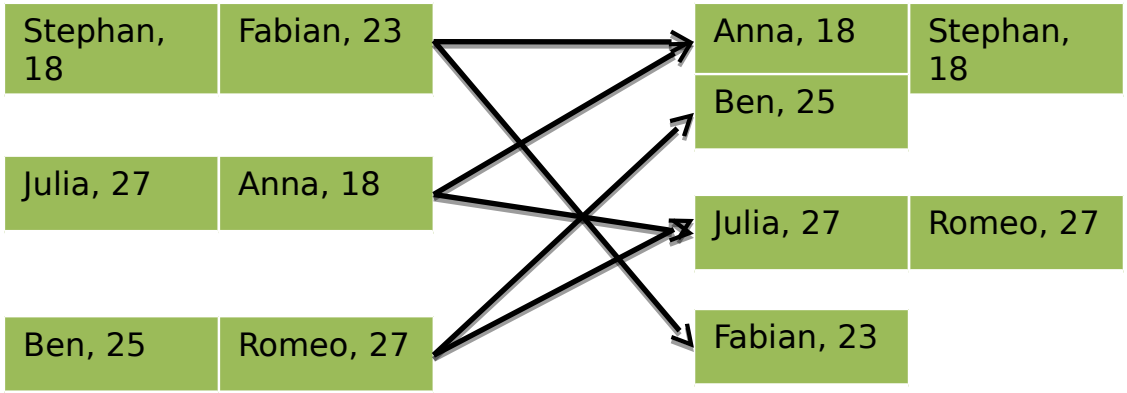
Transformations: Filter

```
// The DataStream
DataStream<Integer> integers = env.fromElements(1, 2, 3, 4);

DataStream<Integer> filtered =

      integers.filter(new FilterFunction<Integer>() {
            @Override
          public boolean filter(Integer value) {
           return value != 3;
      }
  });

filtered.print();
> 1, 2, 4
```

Transformations: KeyBy

- A DataStream can be organized by a key
  - Partitions the data (all elements with the same key are processed by the same operator)
  - Certain operators are key-aware
  - Operator state can be partitioned by key

```
// (name, age) of employees
DataStream<Tuple2<String, Integer>> passengers = ...

// group by second field (age)
DataStream<Integer, Integer> grouped = passengers.keyBy(1)
```



Rich Functions

RichFunctions

- Function interfaces have only one method
  - Single abstract method (SAM)
  - Support for Java8 Lambda functions

- There is a “Rich” variant for each function.
  - RichFlatMapFunction, …
  - Additional methods
    - open(Configuration c)
    - close()
    - getRuntimeContext()

RichFunctions & RuntimeContext

- RuntimeContext has useful methods:
  - getIndexOfThisSubtask ()
  - getNumberOfParallelSubtasks()
  - getExecutionConfig()
- Hands out partitioned state (later discussed)
  - getKeyValueState()

Fault-Tolerance and Operator State

Stateful Functions

- DataStream functions can be stateful
  - Function state is checkpointed and recovered in case of a failure
- State is organized by key
  - Functions on a keyed stream can access and update state scoped to the current key
- See documentation for details: https://ci.apache.org/projects/flink/flink-docs-master/internals/stream_checkpointing.html

Defining Key-Partitioned State

```
DataStream<Tuple2<String, String>> aStream;
KeyedStream<Tuple2<String, String>, Tuple> keyedStream = aStream.keyBy(0);
DataStream<Long> lengths = keyedStream.map(new MapWithCounter());

public static class MapWithCounter
        extends RichMapFunction<Tuple2<String, String>, Long> {

        private ValueState<Long> totalLengthByKey;

       @Override
              public void open (Configuration conf) {
               totalLengthByKey = getRuntimeContext()
                      .getState("totalLengthByKey", Long.class, 0L);
        }

        @Override
        public Long map (Tuple2<String,String> value) throws Exception {
               long newTotalLength = totalLengthByKey.value() + value.f1.length();
               totalLengthByKey.update(newTotalLength);
               return totalLengthByKey.value();
        }
}
```

Exercises!

Exercises

- Start working on the exercises

  http://dataartisans.github.io/flink-training

- Starter exercise: Taxi Ride Cleansing
- Advanced exercise: Average Taxi Ride Speed

- Don’t hesitate to ask us!