

dataArtisans



Apache Flink® Training

DataStream API Basic

August 26, 2015

DataStream API



- Stream Processing
- Java and Scala
- All examples here in Java
- Documentation available at flink.apache.org
- Currently labeled as *beta* – some API changes are pending
 - Noted in the slides with a warning

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();

    DataSet<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())
        // group by the tuple field "0"
        .groupBy(0)
        // keep the last 5 minute of data
        .window(Time.of(5, TimeUnit.MINUTES))
        //sum up tuple field "1"
        .sum(1);

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

Stream Execution Environment



```
public static void main(String[] args) throws Exception {  
    // set up the execution environment  
    final StreamExecutionEnvironment env =  
        StreamExecutionEnvironment.getExecutionEnvironment();  
  
    DataSet<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())  
        // group by the tuple field "0"  
        .groupBy(0)  
        // keep the last 5 minute of data  
        .window(Time.of(5, TimeUnit.MINUTES))  
        //sum up tuple field "1"  
        .sum(1);  
  
    // print result in command line  
    counts.print();  
    // execute program  
    env.execute("Socket Incremental WordCount Example");  
}
```

Data Sources



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

    DataSet<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())
        // group by the tuple field "0"
        .groupBy(0)
        // keep the last 5 minute of data
        .window(Time.of(5, TimeUnit.MINUTES))
        //sum up tuple field "1"
        .sum(1);

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

Data types



```
public static void main(String[] args) throws Exception {  
    // set up the execution environment  
    final StreamExecutionEnvironment env =  
        StreamExecutionEnvironment.getExecutionEnvironment();  
  
    DataSet<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())  
        // group by the tuple field "0"  
        .groupBy(0)  
        // keep the last 5 minute of data  
        .window(Time.of(5, TimeUnit.MINUTES))  
        //sum up tuple field "1"  
        .sum(1);  
  
    // print result in command line  
    counts.print();  
    // execute program  
    env.execute("Socket Incremental WordCount Example");  
}
```

Transformations



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

    DataSet<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())
        // group by the tuple field "0"
        .groupBy(0)
        // keep the last 5 minute of data
        .window(Time.of(5, TimeUnit.MINUTES))
        //sum up tuple field "1"
        .sum(1);

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


User functions



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

    DataSet<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())
        // group by the tuple field "0"
        .groupBy(0)
        // keep the last 5 minute of data
        .window(Time.of(5, TimeUnit.MINUTES))
        //sum up tuple field "1"
        .sum(1);

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

DataSinks



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

    DataSet<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())
        // group by the tuple field "0"
        .groupBy(0)
        // keep the last 5 minute of data
        .window(Time.of(5, TimeUnit.MINUTES))
        //sum up tuple field "1"
        .sum(1);

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

Execute!



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

    DataSet<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())
        // group by the tuple field "0"
        .groupBy(0)
        // keep the last 5 minute of data
        .window(Time.of(5, TimeUnit.MINUTES))
        //sum up tuple field "1"
        .sum(1);

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

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

Transformations: Partitioning



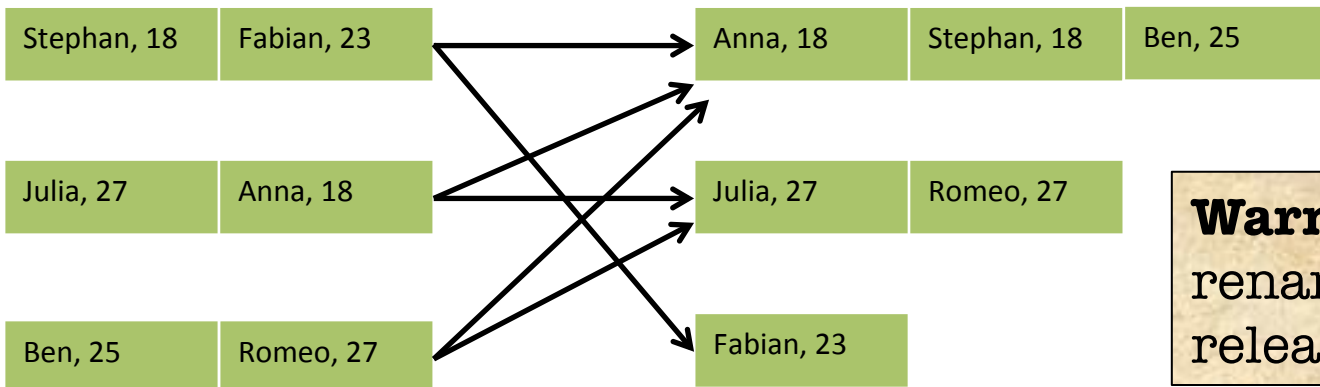
- DataStreams can be partitioned by a key

```
// (name, age) of employees
```

```
DataStream<Tuple2<String, Integer>> passengers = ...
```

```
// group by second field (age)
```

```
DataStream<Integer, Integer> grouped = passengers.groupBy(1)
```



Warning: Possible renaming in next releases

Data Shipping Strategies



- Optionally, you can specify how data is shipped between two transformations
- Forward: `stream.forward()`
 - Only local communication
- Rebalance: `stream.rebalance()`
 - Round-robin partitioning
- Partition by hash: `stream.partitionByHash(...)`
- Custom partitioning: `stream.partitionCustom(...)`
- Broadcast: `stream.broadcast()`
 - Broadcast to all nodes

Data Sources



Collection

- `fromCollection(collection)`
- `fromElements(1,2,3,4,5)`

Data Sources (2)



Text socket

- `socketTextStream("hostname", port)`

Text file

- `readFileStream("/path/to/file", 1000, WatchType.PROCESS_ONLY_APPENDED)`

Connectors

- E.g., Apache Kafka, RabbitMQ, ...

Data Sources: Collections



```
StreamExecutionEnvironment env =  
    StreamExecutionEnvironment.getExecutionEnvironment();  
  
// read from elements  
DataStream<String> names = env.fromElements("Some", "Example",  
    "Strings");  
  
// read from Java collection  
List<String> list = new ArrayList<String>();  
list.add("Some");  
list.add("Example");  
list.add("Strings");  
  
DataStream<String> names = env.fromCollection(list);
```

Data Sources: Files, Sockets, Connectors



```
StreamExecutionEnvironment env =  
    StreamExecutionEnvironment.getExecutionEnvironment();  
  
// read text socket from port  
DataStream<String> socketLines = env  
    .socketTextStream("localhost", 9999);  
  
// read a text file ingesting new elements every 100 milliseconds  
DataStream<String> localLines = env  
    .readFileStream("/path/to/file", 1000,  
        WatchType.PROCESS_ONLY_APPENDED);
```

Data Sinks



Text

- `writeAsText("/path/to/file")`

CSV

- `writeAsCsv("/path/to/file")`

Return data to the Client

- `print()`

Note: Identical to
DataSet API

Data Sinks (2)



Socket

- `writeToSocket(hostname, port, SerializationSchema)`

Connectors

- E.g., Apache Kafka, Elasticsearch, Rolling HDFS Files

Data Sinks



- Lazily executed when `env.execute()` is called

```
DataStream<...> result;
```

```
// nothing happens
```

```
result.writeToSocket(...);
```

```
// nothing happens
```

```
result.writeAsText("/path/to/file", "\n", "|");
```

```
// Execution really starts here
```

```
env.execute();
```

Fault Tolerance

Fault Tolerance in Flink



- Flink provides recovery by taking a consistent checkpoint every N milliseconds and rolling back to the checkpointed state
 - https://ci.apache.org/projects/flink/flink-docs-master/internals/stream_checkpointing.html
- Exactly once (default)
 - *// Take checkpoint every 5000 milliseconds*
`env.enableCheckpointing (5000)`
- At least once (for lower latency)
 - *// Take checkpoint every 5000 milliseconds*
`env.enableCheckpointing (5000, CheckpointingMode.AT_LEAST_ONCE)`
- Setting the interval to few seconds should be good for most applications
- If checkpointing is not enabled, no recovery guarantees are provided

Best Practices

Some advice



- Use `env.fromElements(...)` or `env.fromCollection(...)` to quickly get a `DataStream` to experiment with
- Use `print()` to quickly print a `DataStream`

Update Guide

From 0.9 to 0.10



- `groupBy(...)` -> `keyBy(...)`
- `DataStream` renames:
 - `KeyedDataStream` -> `KeyedStream`
 - `WindowedDataStream` -> `WindowedStream`
 - `ConnectedDataStream` -> `ConnectedStream`
 - `JoinOperator` -> `JoinedStreams`

