

dataArtisans



Apache Flink® Training

DataStream API Basic

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;
        }
    });

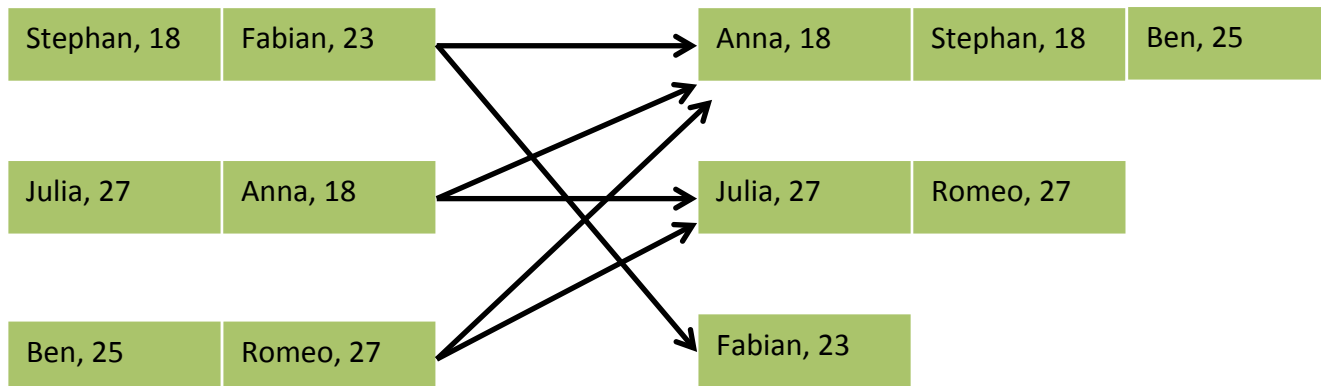
filtered.print();
> 1, 2, 4
```

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)
```



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

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 ()`

Data Sources

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



```
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", 100,  
        WatchType.PROCESS_ONLY_APPENDED);
```

Custom SourceFunctions & Connectors



```
StreamExecutionEnvironment env =  
    StreamExecutionEnvironment.getExecutionEnvironment();  
  
// read data stream from custom source function  
DataStream<Tuple2<Long, String> stream = env  
    .addSource(new MySourceFunction());
```

Flink has connectors for many stream serving systems

- Apache Kafka
- RabbitMQ
- ...

Data Sinks

Data Sinks



Write as text file using toString()

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

Write as CSV file

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

Print to the standard output

- `stream.print()`

Emit to socket

- `stream`
 `.writeToSocket(host,port,SerializationSchema)`

Custom Sinks & Connectors



Emit data with a custom sink function

- `stream.addSink(new MySinkFunction());`

Several connectors

- Apache Kafka
- Elasticsearch
- Rolling Files (HDFS, S3, ...)

Execution



- Programs are lazily executed when `execute()` is called

```
DataStream<T> result;
```

```
// nothing happens
```

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

```
// nothing happens
```

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

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

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

Fault-Tolerance and Operator State

Fault Tolerance in Flink



- Flink takes a checkpoint of an application every N milliseconds and rolls back to the checkpointed state in case of a failure
- Enable exactly-once consistency (checkpoint every 5 seconds)
`env.enableCheckpointing(5000)`
- Enable at-least-once consistency (for lower latency)
`env.enableCheckpointing(5000, CheckpointingMode.AT_LEAST_ONCE)`
- Setting the interval to few seconds should be good for most application
- If checkpointing is not enabled, no recovery guarantees are provided
- See documentation for details:
https://ci.apache.org/projects/flink/flink-docs-master/internals/stream_checkpointing.html

Stateful Functions



- All DataStream functions can be stateful
 - State is checkpointed and recovered in case of a failure (if checkpointing is enabled).
- You can define two types of state
 - *Local State*: Functions can register local variables to be checkpointed.
 - *Key-Partitioned State*: Functions on a keyed stream can access and update state scoped to the current key.

Defining Local State



```
DataStream<String> aStream;  
DataStream<Long> lengths = aStream.map(new MapWithCounter());
```

```
public static class MapWithCounter  
    implements MapFunction<String, Long>, Checkpointed<Long> {  
  
    private long totalLength = 0;  
  
    @Override  
    public Long map (String value) {  
        totalLength += value.length();  
        return totalLength;  
    }  
  
    @Override  
    public Long snapshotState(long cpId, long cpTimestamp) throws Exception {  
        return totalLength;  
    }  
  
    @Override  
    public void restoreState(Long state) throws Exception {  
        totalLength = state;  
    }  
}
```

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();
    }
}
```

Best Practices

Some advice



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

