**EBook Gratis**

# APRENDIZAJE apache-flink

Free unaffiliated eBook created from
**Stack Overflow contributors.**

#apache-flink

# Tabla de contenido

# Acerca de

You can share this PDF with anyone you feel could benefit from it, downloaded the latest version from: apache-flink

It is an unofficial and free apache-flink ebook created for educational purposes. All the content is extracted from Stack Overflow Documentation, which is written by many hardworking individuals at Stack Overflow. It is neither affiliated with Stack Overflow nor official apache-flink.



Use the content presented in this book at your own risk; it is not guaranteed to be correct nor accurate, please send your feedback and corrections to info@zzzprojects.com

# Capítulo 1: Empezando con el apache-flink

## Observaciones

Esta sección proporciona una descripción general de qué es apache-flink y por qué un desarrollador puede querer usarlo.

También debe mencionar cualquier tema grande dentro de apache-flink, y vincular a los temas relacionados. Dado que la Documentación para apache-flink es nueva, es posible que deba crear versiones iniciales de esos temas relacionados.

## Examples

### Resumen y requisitos

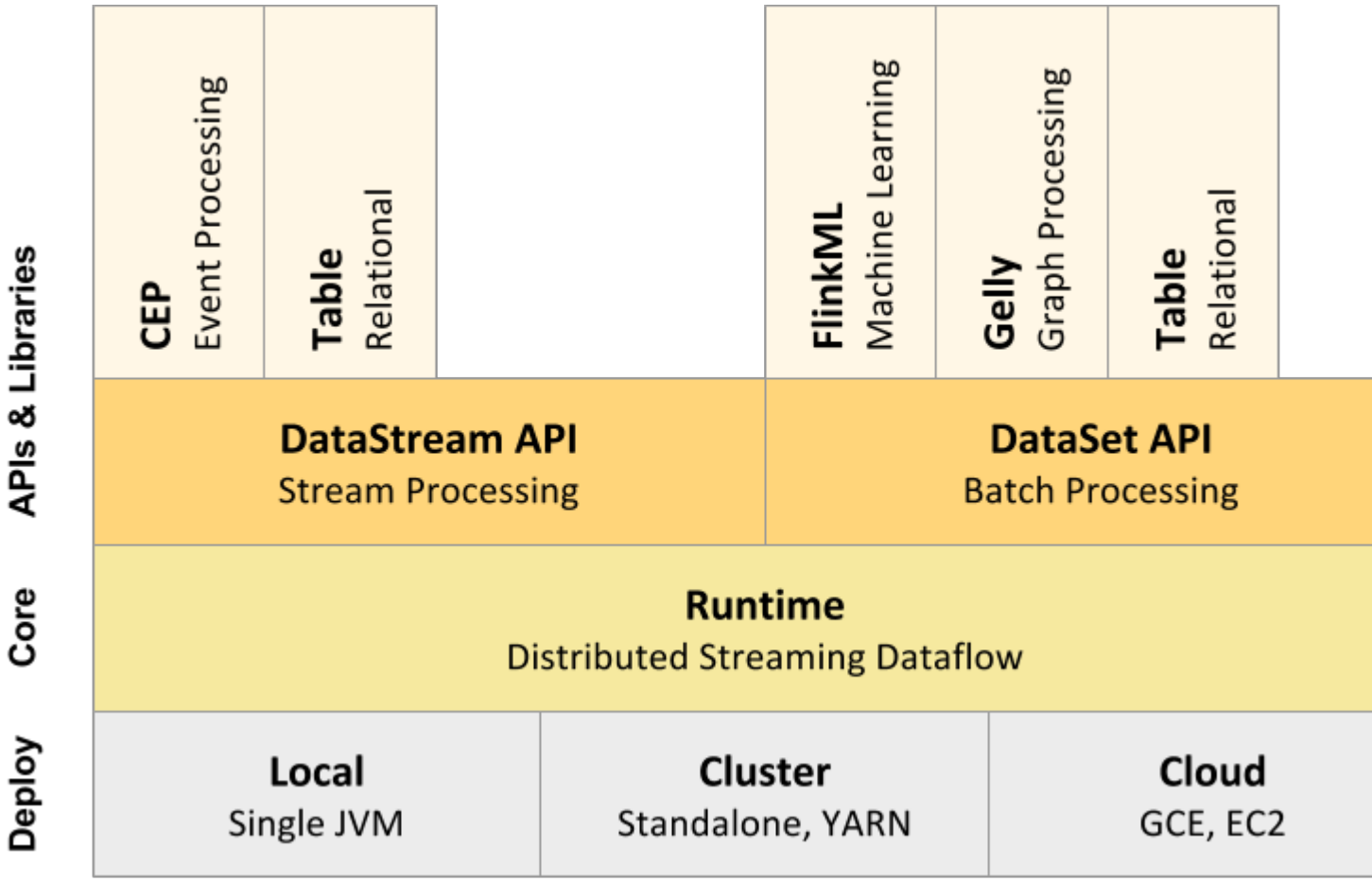
# Que es flink

Al igual que Apache Hadoop y Apache Spark , Apache Flink es un marco de código abierto impulsado por la comunidad para el análisis de Big Data distribuido. Escrito en Java, Flink tiene API para Scala, Java y Python, lo que permite el análisis de transmisión por lotes y en tiempo real.

# Requerimientos

- un entorno similar a UNIX, como Linux, Mac OS X o Cygwin;
- Java 6.X o posterior;
- [opcional] Maven 3.0.4 o posterior.

# Apilar




# Entornos de ejecución

> Apache Flink es un sistema de procesamiento de datos y **una alternativa al componente MapReduce de Hadoop** . Viene con su *propio tiempo de ejecución en* lugar de construir sobre MapReduce. Como tal, puede funcionar completamente independientemente del ecosistema de Hadoop.

El entorno de `ExecutionEnvironment` es el contexto en el que se ejecuta un programa. Hay diferentes entornos que puede utilizar, dependiendo de sus necesidades.

1. *Entorno JVM* : Flink puede ejecutarse en una única máquina virtual Java, lo que permite a los usuarios probar y depurar programas de Flink directamente desde su IDE. Al utilizar este entorno, todo lo que necesita son las dependencias de Maven correctas.

2. *Entorno local* : para poder ejecutar un programa en una instancia de Flink en ejecución (no desde su IDE), debe instalar Flink en su máquina. Ver configuración local .

3. *Entorno de clúster* : ejecutar Flink de forma totalmente distribuida requiere un clúster independiente o un hilado. Consulte la página de configuración del clúster o este slideshare para obtener más información. mportant__: el `2.11` en el nombre del artefacto es la *versión de scala* , asegúrese de coincidir con el que tiene en su sistema.

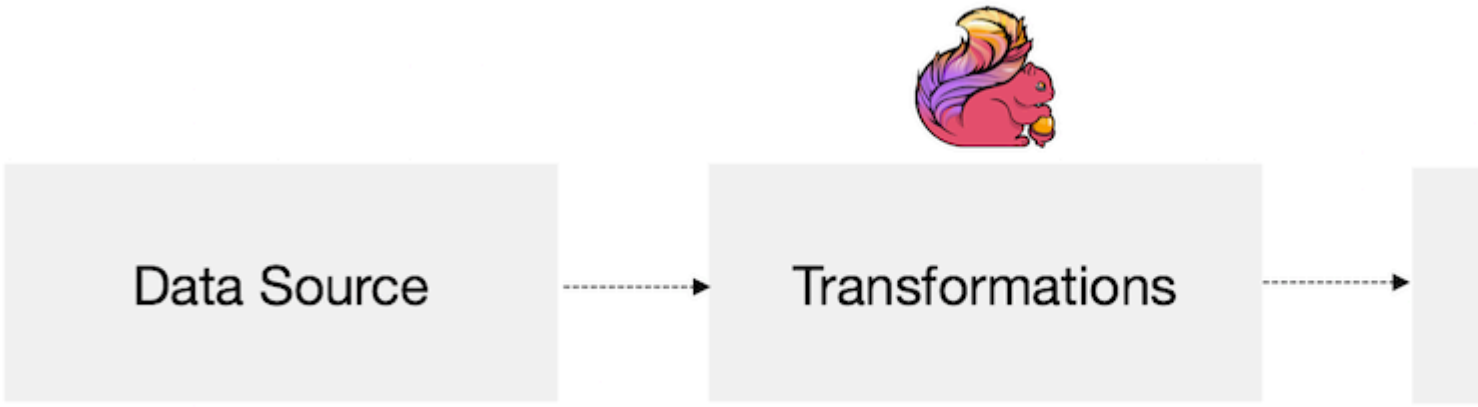
# APIs

Flink se puede utilizar para el procesamiento de flujo o por lotes. Ofrecen tres APIs:

- **API de DataStream** : procesamiento de flujos, es decir, transformaciones (filtros, ventanas de tiempo, agregaciones) en flujos de datos ilimitados.
- **DataSet API** : procesamiento por lotes, es decir, transformaciones en conjuntos de datos.
- **Tabla API** : un lenguaje de expresión similar a SQL (como los marcos de datos en Spark) que se puede incrustar tanto en aplicaciones de lotes como de transmisión.

# Bloques de construcción

En el nivel más básico, Flink está hecho de fuente (s), transformaciones (es) y sumideros (s).




En el nivel más básico, un programa Flink se compone de:

- **Fuente de datos** : Datos entrantes que Flink procesa.
- **Transformaciones** : el paso de procesamiento, cuando Flink modifica los datos entrantes.
- **Fregadero de datos** : donde Flink envía datos después del procesamiento

Las fuentes y los sumideros pueden ser archivos locales / HDFS, bases de datos, colas de mensajes, etc. Ya hay muchos conectores de terceros disponibles, o puede crear fácilmente los suyos.

## Configuración de tiempo de ejecución local

0. asegúrese de tener java 6 o superior y de que la variable de entorno `JAVA_HOME` esté establecida.

1. descarga el último binario de flink aquí :

   ```
   wget flink-XXXX.tar.gz
   ```

   Si no piensa trabajar con Hadoop, elija la versión hadoop 1. Además, tenga en cuenta la

versión de Scala que descarga, para que pueda agregar las dependencias de Maven correctas en sus programas.

2. empezar a flink

```
tar xzvf flink-XXXX.tar.gz
./flink/bin/start-local.sh
```

Flink ya está configurado para ejecutarse localmente. Para asegurarse de que flink se está ejecutando, puede inspeccionar los registros en `flink/log/` o abrir la interfaz del gestor de tareas flink que se ejecuta en `http://localhost:8081` .

3. dejar de flink

```
./flink/bin/stop-local.sh
```

## Configuración de Flink Environment

Para ejecutar un programa de flink desde su IDE (podemos usar Eclipse o Intellij IDEA (preferido)), necesita dos dependencias: `flink-java` / `flink-scala` y `flink-clients` (a partir de febrero de 2016). Estos JARS se pueden agregar usando Maven y SBT (si está usando scala).

- **Maven**

```
<dependency>
    <groupId>org.apache.flink</groupId>
    <artifactId>flink-java</artifactId>
    <version>1.1.4</version>
</dependency>

<dependency>
    <groupId>org.apache.flink</groupId>
    <artifactId>flink-clients_2.11</artifactId>
    <version>1.1.4</version>
</dependency>
```

- Nombre de **SBT** : = ""

```
version := "1.0"

scalaVersion := "2.11.8"

libraryDependencies ++= Seq(
  "org.apache.flink" %% "flink-scala" % "1.2.0",
  "org.apache.flink" %% "flink-clients" % "1.2.0"
)
```

**importante** : el `2.11` en el nombre del artefacto es la *versión de scala* , asegúrese de coincidir con el que tiene en su sistema.

## WordCount - API de tabla

Este ejemplo es el mismo que *WordCount* , pero utiliza la API de tabla. Ver *WordCount* para detalles sobre ejecución y resultados.

## Maven

Para usar la API de la tabla, agregue `flink-table` como una dependencia de Maven:

```
<dependency>
    <groupId>org.apache.flink</groupId>
    <artifactId>flink-table_2.11</artifactId>
    <version>1.1.4</version>
</dependency>
```

## El código

```
public class WordCountTable{

    public static void main( String[] args ) throws Exception{

        // set up the execution environment
        final ExecutionEnvironment env = ExecutionEnvironment.getExecutionEnvironment();
        final BatchTableEnvironment tableEnv = TableEnvironment.getTableEnvironment( env );

        // get input data
        DataSource<String> source = env.fromElements(
                "To be, or not to be,--that is the question:--",
                "Whether 'tis nobler in the mind to suffer",
                "The slings and arrows of outrageous fortune",
                "Or to take arms against a sea of troubles"
        );

        // split the sentences into words
        FlatMapOperator<String, String> dataset = source
                .flatMap( ( String value, Collector<String> out ) -> {
                    for( String token : value.toLowerCase().split( "\\W+" ) ){
                        if( token.length() > 0 ){
                            out.collect( token );
                        }
                    }
                } )
                // with lambdas, we need to tell flink what type to expect
                .returns( String.class );

        // create a table named "words" from the dataset
        tableEnv.registerDataSet( "words", dataset, "word" );

        // word count using an sql query
        Table results = tableEnv.sql( "select word, count(*) from words group by word" );
        tableEnv.toDataSet( results, Row.class ).print();
    }
}
```

*Nota* : para una versión que use Java <8, reemplace el lambda por una clase anónima:

```
FlatMapOperator<String, String> dataset = source.flatMap( new FlatMapFunction<String,
```

```
String>(){
        @Override
        public void flatMap( String value, Collector<String> out ) throws Exception{
            for( String token : value.toLowerCase().split( "\\W+" ) ){
                if( token.length() > 0 ){
                    out.collect( token );
                }
            }
        }
    } );
```

### El recuento de palabras

## Maven

Agregue las dependencias `flink-java` y `flink-client` (como se explica en el ejemplo de *configuración del entorno JVM* ).

## El código

```
public class WordCount{

    public static void main( String[] args ) throws Exception{

        // set up the execution environment
        final ExecutionEnvironment env = ExecutionEnvironment.getExecutionEnvironment();

        // input data
        // you can also use env.readTextFile(...) to get words
        DataSet<String> text = env.fromElements(
                "To be, or not to be,--that is the question:--",
                "Whether 'tis nobler in the mind to suffer",
                "The slings and arrows of outrageous fortune",
                "Or to take arms against a sea of troubles,"
        );

        DataSet<Tuple2<String, Integer>> counts =
                // split up the lines in pairs (2-tuples) containing: (word,1)
                text.flatMap( new LineSplitter() )
                        // group by the tuple field "0" and sum up tuple field "1"
                        .groupBy( 0 )
                        .aggregate( Aggregations.SUM, 1 );

        // emit result
        counts.print();
    }
}
```

`LineSplitter.java` :

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

    public void flatMap( String value, Collector<Tuple2<String, Integer>> out ){
        // normalize and split the line into words
        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 ) );
            }
        }
    }
}
```

Si usa Java 8, puede reemplazar `.flatmap(new LineSplitter())` por una expresión lambda:

```
DataSet<Tuple2<String, Integer>> counts = text
    // split up the lines in pairs (2-tuples) containing: (word,1)
    .flatMap( ( String value, Collector<Tuple2<String, Integer>> out ) -> {
        // normalize and split the line into words
        String[] tokens = value.toLowerCase().split( "\\W+" );

        // emit the pairs
        for( String token : tokens ){
            if( token.length() > 0 ){
                out.collect( new Tuple2<>( token, 1 ) );
            }
        }
    } )
    // group by the tuple field "0" and sum up tuple field "1"
    .groupBy( 0 )
    .aggregate( Aggregations.SUM, 1 );
```

# Ejecución

**Desde el IDE** : simplemente pulse *Ejecutar* en su IDE. Flink creará un entorno dentro de la JVM.

**Desde la línea de comandos de flink** : para ejecutar el programa utilizando un entorno local independiente, haga lo siguiente:

1. asegúrese de que flink esté en ejecución ( `flink/bin/start-local.sh` );
2. crear un archivo jar ( `maven package` );
3. use la herramienta de línea de comandos `flink` (en la carpeta `bin` de su instalación de flink) para iniciar el programa:

   ```
   flink run -c your.package.WordCount target/your-jar.jar
   ```

   La opción `-c` permite especificar la clase a ejecutar. No es necesario si el jar es ejecutable / define una clase principal.

# Resultado

```
(a,1)
(against,1)
```

```
(and,1)
(arms,1)
(arrows,1)
(be,2)
(fortune,1)
(in,1)
(is,1)
(mind,1)
(nobler,1)
(not,1)
(of,2)
(or,2)
(outrageous,1)
(question,1)
(sea,1)
(slings,1)
(suffer,1)
(take,1)
(that,1)
(the,3)
(tis,1)
(to,4)
(troubles,1)
(whether,1)
```

### WordCount - API de streaming

Este ejemplo es el mismo que *WordCount* , pero utiliza la API de tabla. Ver *WordCount* para detalles sobre ejecución y resultados.

## Maven

Para usar la API de transmisión, agregue `flink-streaming` como una dependencia de Maven:

```
<dependency>
    <groupId>org.apache.flink</groupId>
    <artifactId>flink-streaming-java_2.11</artifactId>
    <version>1.1.4</version>
</dependency>
```

## El código

```
public class WordCountStreaming{

    public static void main( String[] args ) throws Exception{

        // set up the execution environment
        StreamExecutionEnvironment env = StreamExecutionEnvironment.getExecutionEnvironment();

        // get input data
        DataStreamSource<String> source = env.fromElements(
                "To be, or not to be,--that is the question:--",
                "Whether 'tis nobler in the mind to suffer",
                "The slings and arrows of outrageous fortune",
```

```
                "Or to take arms against a sea of troubles"
        );

        source
                // split up the lines in pairs (2-tuples) containing: (word,1)
                .flatMap( ( String value, Collector<Tuple2<String, Integer>> out ) -> {
                    // emit the pairs
                    for( String token : value.toLowerCase().split( "\\W+" ) ){
                        if( token.length() > 0 ){
                            out.collect( new Tuple2<>( token, 1 ) );
                        }
                    }
                } )
                // due to type erasure, we need to specify the return type
                .returns( TupleTypeInfo.getBasicTupleTypeInfo( String.class, Integer.class ) )
                // group by the tuple field "0"
                .keyBy( 0 )
                // sum up tuple on field "1"
                .sum( 1 )
                // print the result
                .print();

        // start the job
        env.execute();
    }
}
```

Lea Empezando con el apache-flink en línea: https://riptutorial.com/es/apache-flink/topic/5798/empezando-con-el-apache-flink

# Capítulo 2: Cómo definir un esquema de serialización (des) personalizado

## Introducción

Algunos conectores (Kafka, RabbitMQ) utilizan los esquemas para convertir los mensajes en objetos Java y viceversa.

## Examples

### Ejemplo de esquema personalizado

Para usar un esquema personalizado, todo lo que necesita hacer es implementar una de las interfaces `SerializationSchema` o `DeserializationSchema` .

```
public class MyMessageSchema implements DeserializationSchema<MyMessage>,
SerializationSchema<MyMessage> {

    @Override
    public MyMessage deserialize(byte[] bytes) throws IOException {
        return MyMessage.fromString(new String(bytes));
    }

    @Override
    public byte[] serialize(MyMessage myMessage) {
        return myMessage.toString().getBytes();
    }

    @Override
    public TypeInformation<MyMessage> getProducedType() {
        return TypeExtractor.getForClass(MyMessage.class);
    }

    // Method to decide whether the element signals the end of the stream.
    // If true is returned the element won't be emitted.
    @Override
    public boolean isEndOfStream(MyMessage myMessage) {
        return false;
    }
}
```

La clase `MyMessage` se define de la siguiente manera:

```
public class MyMessage{

    public int id;
    public String payload;
    public Date timestamp;

    public MyMessage(){}
```

```
    public static MyMessage fromString( String s ){
        String[] tokens = s.split( "," );
        if(tokens.length != 3) throw new RuntimeException( "Invalid record: " + s );

        try{
            MyMessage message = new MyMessage();
            message.id = Integer.parseInt(tokens[0]);
            message.payload = tokens[1];
            message.timestamp = new Date( Long.parseLong(tokens[0]));
            return message;
        }catch(NumberFormatException e){
            throw new RuntimeException("Invalid record: " + s);
        }
    }

    public String toString(){
        return String.format("%d,%s,%dl", id, payload, timestamp.getTime());
    }
}
```

Lea Cómo definir un esquema de serialización (des) personalizado en línea: https://riptutorial.com/es/apache-flink/topic/9004/como-definir-un-esquema-de-serializacion--des--personalizado

# Capítulo 3: Consumir datos de Kafka.

## Examples

### Ejemplo de KafkaConsumer

`FlinkKafkaConsumer` permite consumir datos de uno o más temas de kafka.

## versiones

El consumidor a utilizar depende de su distribución de kafka.

- `FlinkKafkaConsumer08` : utiliza la antigua API `SimpleConsumer` de Kafka. Las compensaciones son manejadas por Flink y comprometidas con el guardián del zoológico.
- `FlinkKafkaConsumer09` : utiliza la nueva API de consumo de Kafka, que maneja las compensaciones y el rebalanceo automáticamente.
- `FlinkKafkaProducer010` : este conector admite mensajes Kafka con marcas de tiempo tanto para producir como para consumir (útil para operaciones de ventana).

## uso

Los binarios no son parte de flink core, por lo que necesita importarlos:

```
<dependency>
  <groupId>org.apache.flink</groupId>
  <artifactId>flink-connector-kafka-0.${kafka.version}_2.10</artifactId>
  <version>RELEASE</version>
</dependency>
```

El constructor toma tres argumentos:

- uno o más temas para leer
- un esquema de deserialización que le dice a Flink cómo interpretar / decodificar los mensajes
- Propiedades de configuración del consumidor kafka. Esos son lo mismo que un consumidor kafka "regular". Los mínimos requeridos son:
    - `bootstrap.servers` : una lista separada por comas de los corredores Kafka en el formulario ip: puerto. Para la versión 8, use `zookeeper.connect` (lista de servidores de zookeeper) en su lugar
    - `group.id` : el id del grupo de consumidores (consulte la documentación de kafka para obtener más detalles)

En Java:

```
Properties properties = new Properties();
properties.put("group.id", "flink-kafka-example");
```

```
properties.put("bootstrap.servers", "localhost:9092");

DataStream<String> inputStream = env.addSource(
        new FlinkKafkaConsumer09<>(
            kafkaInputTopic, new SimpleStringSchema(), properties));
```

En Scala:

```
val properties = new Properties();
properties.setProperty("bootstrap.servers", "localhost:9092");
properties.setProperty("group.id", "test");

inputStream = env.addSource(
        new FlinkKafkaConsumer08[String](
            "topic", new SimpleStringSchema(), properties))
```

Durante el desarrollo, puede usar las propiedades kafka `enable.auto.commit=false` y `auto.offset.reset=earliest` para volver a resumir los mismos datos cada vez que inicie su programa.

## Tolerancia a fallos

Como se explica en los documentos ,

> Con el punto de control de Flink habilitado, el consumidor de Flink Kafka consumirá registros de un tema y periódicamente revisará todos sus compensaciones de Kafka, junto con el estado de otras operaciones, de manera consistente. En caso de una falla en el trabajo, Flink restaurará el programa de transmisión al estado del último punto de control y volverá a consumir los registros de Kafka, a partir de las compensaciones que se almacenaron en el punto de control.
>
> El intervalo de los puntos de control de dibujo, por lo tanto, define cuánto puede tener que retroceder el programa a lo sumo, en caso de una falla.

Para usar los consumidores Kafka con tolerancia a fallas, debe habilitar el control en el entorno de ejecución utilizando el método `enableCheckpointing` :

```
final StreamExecutionEnvironment env = StreamExecutionEnvironment.getExecutionEnvironment();
env.enableCheckpointing(5000); // checkpoint every 5 seconds
```

## Esquemas de deserialización incorporados

**SimpleStringSchema** : `SimpleStringSchema` deserializa el mensaje como una cadena. En caso de que sus mensajes tengan claves, este último será ignorado.

```
new FlinkKafkaConsumer09<>(kafkaInputTopic, new SimpleStringSchema(), prop);
```

### JSONDeserializationSchema

`JSONDeserializationSchema` deserializa los mensajes con formato json usando *jackson* y devuelve

una secuencia de objetos `com.fasterxml.jackson.databind.node.ObjectNode` . Luego puede usar el `.get("property")` para acceder a los campos. Una vez más, las claves son ignoradas.

```
new FlinkKafkaConsumer09<>(kafkaInputTopic, new JSONDeserializationSchema(), prop);
```

### JSONKeyValueDeserializationSchema

`JSONKeyValueDeserializationSchema` es muy similar al anterior, pero trata los mensajes con claves Y valores codificados con json.

```
boolean fetchMetadata = true;
new FlinkKafkaConsumer09<>(kafkaInputTopic, new
JSONKeyValueDeserializationSchema(fetchMetadata), properties);
```

El `ObjectNode` devuelto contiene los siguientes campos:

- `key` : todos los campos presentes en la clave.
- `value` : todos los campos de mensaje
- (opcional) `metadata` : expone el `offset` , la `partition` y el `topic` del mensaje (pase `true` al constructor para obtener también los metadatos).

Por ejemplo:

```
kafka-console-producer --broker-list localhost:9092 --topic json-topic \
    --property parse.key=true \
    --property key.separator=|
{"keyField1": 1, "keyField2": 2} | {"valueField1": 1, "valueField2" : {"foo": "bar"}}
^C
```

Será decodificado como:

```
{
    "key":{"keyField1":1,"keyField2":2},
    "value":{"valueField1":1,"valueField2":{"foo":"bar"}},
    "metadata":{
        "offset":43,
        "topic":"json-topic",
        "partition":0
    }
}
```

## Particiones Kafka y paralelismo Flink.

En kafka, a cada consumidor del mismo grupo de consumidores se le asigna una o más particiones. Tenga en cuenta que no es posible que dos consumidores consuman desde la misma partición. El número de consumidores de flink depende del paralelismo de flink (el valor predeterminado es 1).

Hay tres casos posibles:

1. **kafka particiones == flink paralelismo** : este caso es ideal, ya que cada consumidor se

ocupa de una partición. Si sus mensajes están equilibrados entre particiones, el trabajo se distribuirá de manera uniforme entre los operadores de flink;

2. **kafka particiones <paralelismo flink** : algunas instancias de flink no recibirán ningún mensaje. Para evitar eso, debe llamar al `rebalance` en su flujo de entrada *antes de cualquier operación* , lo que hace que los datos se vuelvan a particionar:

```
inputStream = env.addSource(new FlinkKafkaConsumer10("topic", new SimpleStringSchema(),
properties));

inputStream
    .rebalance()
    .map(s -> "message" + s)
    .print();
```

3. **Particiones kafka> paralelismo flink** : en este caso, algunas instancias manejarán múltiples particiones. Una vez más, puede utilizar el `rebalance` para distribuir los mensajes de manera uniforme entre los trabajadores.

Lea Consumir datos de Kafka. en línea: https://riptutorial.com/es/apache-flink/topic/9003/consumir-datos-de-kafka-

# Capítulo 4: explotación florestal

## Introducción

Este tema muestra cómo usar y configurar el registro (log4j) en las aplicaciones de Flink.

## Examples

### Usando un registrador en su código

Agregue la dependencia `slf4j` a su `pom.xml` :

```
<properties>
    <slf4j.version>1.7.21</slf4j.version>
</properties>

<!-- ... -->

<dependency>
    <groupId>org.slf4j</groupId>
    <artifactId>slf4j-api</artifactId>
    <version>${slf4j.version}</version>
</dependency>
<dependency>
    <groupId>org.slf4j</groupId>
    <artifactId>slf4j-log4j12</artifactId>
    <version>${slf4j.version}</version>
</dependency>
```

Crea un objeto logger para usar en tu clase:

```
private Logger LOGGER = LoggerFactory.getLogger(FlinkApp.class);
```

En las clases que necesitan ser serializadas, como las subclases de `RichMapFunction` , no olvide declarar `LOGGER` como `transient` :

```
private transient Logger LOG = LoggerFactory.getLogger(MyRichMapper.class);
```

En su código, use `LOGGER` como de costumbre. Use marcadores de posición ( `{}` ) para formatear objetos y tales:

```
LOGGER.info("my app is starting");
LOGGER.warn("an exception occurred processing {}", record, exception);
```

### Configuración de registro

## Modo local

En el modo local, por ejemplo, al ejecutar su aplicación desde un IDE, puede configurar `log4j` como de costumbre, es decir, haciendo que `log4j.properties` esté disponible en la ruta de `log4j.properties` . Una forma fácil en maven es crear `log4j.properties` en la carpeta `src/main/resources` . Aquí hay un ejemplo:

```
log4j.rootLogger=INFO, console

# patterns:
#  d = date
#  c = class
#  F = file
#  p = priority (INFO, WARN, etc)
#  x = NDC (nested diagnostic context) associated with the thread that generated the logging
event
#  m = message


# Log all infos in the console
log4j.appender.console=org.apache.log4j.ConsoleAppender
log4j.appender.console.layout=org.apache.log4j.PatternLayout
log4j.appender.console.layout.ConversionPattern=%d{dd/MM/yyyy HH:mm:ss.SSS} %5p [%-10c] %m%n

# Log all infos in flink-app.log
log4j.appender.file=org.apache.log4j.FileAppender
log4j.appender.file.file=flink-app.log
log4j.appender.file.append=false
log4j.appender.file.layout=org.apache.log4j.PatternLayout
log4j.appender.file.layout.ConversionPattern=%d{dd/MM/yyyy HH:mm:ss.SSS} %5p [%-10c] %m%n

# suppress info messages from flink
log4j.logger.org.apache.flink=WARN
```

# Modo independiente

> En modo independiente, la configuración real utilizada no es la de su archivo `jar` . Esto se debe a que Flink tiene sus propios archivos de configuración, que tienen prioridad sobre los suyos.

**Archivos predeterminados** : Flink se envía con los siguientes archivos de propiedades predeterminadas:

- `log4j-cli.properties` : utilizado por el cliente de la línea de comandos de Flink (por ejemplo, `flink run` ) (no código ejecutado en el clúster)
- `log4j-yarn-session.properties` : utilizado por el cliente de la línea de comandos de Flink al iniciar una sesión `yarn-session.sh` ( `yarn-session.sh` )
- `log4j.properties` : JobManager / Taskmanager logs (tanto independientes como YARN)

Tenga en cuenta que `${log.file}` defecto `flink/log` . Se puede anular en `flink-conf.yaml` , configurando `env.log.dir` ,

> `env.log.dir` define el directorio donde se guardan los registros de Flink. Tiene que ser un camino absoluto.

**Ubicación** del registro: los registros son *locales* , es decir, se producen en las máquinas que ejecutan los JobManager (s) / Taskmanager (s).

**Hilo** : cuando ejecute Flink en Hilo, debe confiar en las capacidades de registro de Hadoop YARN. La característica más útil para eso es la agregación de registros YARN . Para habilitarlo, establezca la `yarn.log-aggregation-enable` en `true` en el `yarn-site.xml file` . Una vez que está habilitado, puede recuperar todos los archivos de registro de una sesión YARN (fallida) usando:

```
yarn logs -applicationId <application ID>
```

Desafortunadamente, los registros están disponibles *solo después de que una sesión dejó de ejecutarse* , por ejemplo, después de una falla.

## Usando diferentes configuraciones para cada aplicación

En caso de que necesite diferentes configuraciones para sus diversas aplicaciones, no existe una manera fácil de hacerlo (a partir de Flink 1.2).

Si usa el modo flink de *one-Yarn-cluster-per-job* (es decir, inicia sus scripts con: `flink run -m yarn-cluster` ...), aquí tiene una solución:

1. Crea un directorio `conf` algún lugar cerca de tu proyecto.
2. crear enlaces simbólicos para todos los archivos en `flink/conf` :

   ```
   mkdir conf
   cd conf
   ln -s flink/conf/* .
   ```

3. reemplace el enlace simbólico `log4j.properties` (o cualquier otro archivo que desee cambiar) por su propia configuración
4. antes de lanzar tu trabajo, corre

   ```
   export FLINK_CONF_DIR=/path/to/my/conf
   ```

Dependiendo de su versión de flink, es posible que deba editar el archivo `flink/bin/config.sh` . Si se ejecuta a través de esta línea:

```
FLINK_CONF_DIR=$FLINK_ROOT_DIR_MANGLED/conf
```

cambiarlo con:

```
if [ -z "$FLINK_CONF_DIR" ]; then
    FLINK_CONF_DIR=$FLINK_ROOT_DIR_MANGLED/conf;
fi
```

## Solución de Flink-on-Yarn: obtenga registros en tiempo real con rsyslog

Yarn no agrega de forma predeterminada los registros antes de que finalice una aplicación, lo que puede ser problemático con trabajos de transmisión que ni siquiera terminan.

Una solución es usar `rsyslog` , que está disponible en la mayoría de las máquinas Linux.

Primero, permita las solicitudes de udp entrantes sin comentar las siguientes líneas en `/etc/rsyslog.conf` :

```
$ModLoad imudp
$UDPServerRun 514
```

Edite su `log4j.properties` (vea los otros ejemplos en esta página) para usar `SyslogAppender` :

```
log4j.rootLogger=INFO, file

# TODO: change package logtest to your package
log4j.logger.logtest=INFO, SYSLOG

# Log all infos in the given file
log4j.appender.file=org.apache.log4j.FileAppender
log4j.appender.file.file=${log.file}
log4j.appender.file.append=false
log4j.appender.file.layout=org.apache.log4j.PatternLayout
log4j.appender.file.layout.ConversionPattern=bbdata: %d{yyyy-MM-dd HH:mm:ss,SSS} %-5p %-60c %x
- %m%n

# suppress the irrelevant (wrong) warnings from the netty channel handler
log4j.logger.org.jboss.netty.channel.DefaultChannelPipeline=ERROR, file

# rsyslog
# configure Syslog facility SYSLOG appender
# TODO: replace host and myTag by your own
log4j.appender.SYSLOG=org.apache.log4j.net.SyslogAppender
log4j.appender.SYSLOG.syslogHost=10.10.10.102
log4j.appender.SYSLOG.port=514
#log4j.appender.SYSLOG.appName=bbdata
log4j.appender.SYSLOG.layout=org.apache.log4j.EnhancedPatternLayout
log4j.appender.SYSLOG.layout.conversionPattern=myTag: [%p] %c:%L - %m %throwable %n
```

El diseño es importante, porque rsyslog trata una nueva línea como una nueva entrada de registro. Arriba, las líneas nuevas (en stacktraces por ejemplo) se omitirán. Si realmente desea que los registros de `rsyslog.conf` / pestañas funcionen "normalmente", edite `rsyslog.conf` y agregue:

```
$EscapeControlCharactersOnReceive off
```

El uso de `myTag:` al principio de `conversionPattern` es útil si desea redirigir todos sus registros a un archivo específico. Para hacerlo, edite `rsyslog.conf` y agregue la siguiente regla:

```
if $programname == 'myTag' then /var/log/my-app.log
& stop
```

Lea explotación florestal en línea: https://riptutorial.com/es/apache-flink/topic/9713/explotacion-

florestal

# Capítulo 5: Punto de control

## Introducción

(probado en Flink 1.2 y por debajo)

Cada función, fuente u operador en Flink puede ser con estado. Los puntos de control permiten a Flink recuperar el estado y las posiciones en los flujos para dar a la aplicación la misma semántica que una ejecución sin fallas. Es el mecanismo detrás de las garantías de *tolerancia a fallas* y el procesamiento *exacto de una vez* .

Lea este artículo para entender los aspectos internos.

## Observaciones

Los puntos de control solo son útiles cuando ocurre una falla en el clúster, por ejemplo cuando falla un administrador de tareas. No persisten después de que el trabajo en sí falló o fue cancelado.

Para poder reanudar un trabajo de estado después del fracaso / cancelación, echar un vistazo a **los puestos de control** o **puntos de retorno externalizados (Flink 1.2+).**

## Examples

### Configuración y configuración

La configuración del punto de control se realiza en dos pasos. Primero, necesitas elegir un *backend* . Luego, puede especificar el intervalo y el modo de los puntos de control por aplicación.

---

# Backends

#### Backends disponibles

El lugar donde se almacenan los puntos de control depende del backend configurado:

- `MemoryStateBackend` : estado en memoria, copia de seguridad en la memoria de JobManager / ZooKeeper. Se debe usar solo para el estado mínimo (predeterminado a un máximo de 5 MB, para almacenar compensaciones de Kafka, por ejemplo) o para pruebas y depuración local.
- `FsStateBackend` : el estado se mantiene en memoria en los TaskManagers, y las instantáneas de estado (es decir, los puntos de control) se almacenan en un sistema de archivos (HDFS, DS3, sistema de archivos local, ...). Se recomienda esta configuración para estados grandes o ventanas largas y para configuraciones de alta disponibilidad.

- `RocksDBStateBackend` : almacena datos en vuelo en una base de datos RocksDB que está (por defecto) almacenada en los directorios de datos de TaskManager. En el punto de control, toda la base de datos de RocksDB se escribe en un archivo (como arriba). En comparación con FsStateBackend, permite estados más grandes (limitados solo por el espacio en disco frente al tamaño de la memoria del administrador de tareas), pero el rendimiento será menor (los datos no siempre en la memoria, deben cargarse desde el disco).

Tenga en cuenta que, independientemente del backend, los metadatos (número de puntos de control, localización, etc.) siempre se almacenan en la memoria del administrador de tareas y los puntos de control **no persistirán después de la finalización / cancelación de la aplicación** .

**Especificando el backend**

Usted especifica el backend en el método `main` su programa usando:

```
StreamExecutionEnvironment env = StreamExecutionEnvironment.getExecutionEnvironment();
env.setStateBackend(new FsStateBackend("hdfs://namenode:40010/flink/checkpoints"));
```

O establezca el backend predeterminado en `flink/conf/flink-conf.yaml` :

```
# Supported backends:
#  - jobmanager (MemoryStateBackend),
#  - filesystem (FsStateBackend),
#  - rocksdb (RocksDBStateBackend),
#  - <class-name-of-factory>
state.backend: filesystem

# Directory for storing checkpoints in a Flink-supported filesystem
# Note: State backend must be accessible from the JobManager and all TaskManagers.
# Use "hdfs://" for HDFS setups, "file://" for UNIX/POSIX-compliant file systems,
# "S3://" for S3 file system.
state.backend.fs.checkpointdir: file:///tmp/flink-backend/checkpoints
```

---

# Habilitando puntos de control

Cada aplicación debe habilitar explícitamente los puntos de control:

```
long checkpointInterval = 5000; // every 5 seconds

StreamExecutionEnvironment env = StreamExecutionEnvironment.getExecutionEnvironment();
env.enableCheckpointing(checkpointInterval);
```

Opcionalmente puede especificar un *modo de punto de control* . Si no es así, por defecto es *exactamente una vez* :

```
env.enableCheckpointing(checkpointInterval, CheckpointingMode.AT_LEAST_ONCE);
```

El modo de punto de control define la consistencia que garantiza el sistema en presencia de fallas. Cuando se activa el punto de control, las secuencias de datos se reproducen de manera tal

que se repiten las partes perdidas del procesamiento. Con `EXACTLY_ONCE` , el sistema dibuja puntos de control de manera que la recuperación se comporte como si los operadores / funciones vieran cada registro "exactamente una vez". Con `AT_LEAST_ONCE` , los puntos de control se dibujan de una manera más simple que normalmente encuentra algunos duplicados en la recuperación.

### Puestos de control de prueba

## El código

Aquí hay una aplicación flink simple que usa un mapeador con estado con un estado administrado de `Integer` . Puedes jugar con las variables `checkpointEnable` , `checkpointInterval` y `checkpointMode` para ver su efecto:

```
public class CheckpointExample {

    private static Logger LOG = LoggerFactory.getLogger(CheckpointExample.class);
    private static final String KAFKA_BROKER = "localhost:9092";
    private static final String KAFKA_INPUT_TOPIC = "input-topic";
    private static final String KAFKA_GROUP_ID = "flink-stackoverflow-checkpointer";
    private static final String CLASS_NAME = CheckpointExample.class.getSimpleName();


    public static void main(String[] args) throws Exception {

        // play with them
        boolean checkpointEnable = false;
        long checkpointInterval = 1000;
        CheckpointingMode checkpointMode = CheckpointingMode.EXACTLY_ONCE;

        // -------------------------------------------------------

        LOG.info(CLASS_NAME + ": starting...");
        final StreamExecutionEnvironment env =
StreamExecutionEnvironment.getExecutionEnvironment();

        // kafka source
        // https://ci.apache.org/projects/flink/flink-docs-release-
1.2/dev/connectors/kafka.html#kafka-consumer
        Properties prop = new Properties();
        prop.put("bootstrap.servers", KAFKA_BROKER);
        prop.put("group.id", KAFKA_GROUP_ID);
        prop.put("auto.offset.reset", "latest");
        prop.put("enable.auto.commit", "false");

        FlinkKafkaConsumer09<String> source = new FlinkKafkaConsumer09<>(
                KAFKA_INPUT_TOPIC, new SimpleStringSchema(), prop);

        // checkpoints
        // internals: https://ci.apache.org/projects/flink/flink-docs-
master/internals/stream_checkpointing.html#checkpointing
        // config: https://ci.apache.org/projects/flink/flink-docs-release-
1.3/dev/stream/checkpointing.html
        if (checkpointEnable) env.enableCheckpointing(checkpointInterval, checkpointMode);

        env
                .addSource(source)
```

```
                .keyBy((any) -> 1)
                .flatMap(new StatefulMapper())
                .print();

        env.execute(CLASS_NAME);
    }

        /* ****************************************************************
         * Stateful mapper
         * (cf. https://ci.apache.org/projects/flink/flink-docs-release-
1.3/dev/stream/state.html)
         * ****************************************************************/

    public static class StatefulMapper extends RichFlatMapFunction<String, String> {
        private transient ValueState<Integer> state;

        @Override
        public void flatMap(String record, Collector<String> collector) throws Exception {
            // access the state value
            Integer currentState = state.value();

            // update the counts
            currentState += 1;
            collector.collect(String.format("%s: (%s,%d)",
                    LocalDateTime.now().format(ISO_LOCAL_DATE_TIME), record, currentState));
            // update the state
            state.update(currentState);
        }

        @Override
        public void open(Configuration parameters) throws Exception {
            ValueStateDescriptor<Integer> descriptor =
                    new ValueStateDescriptor<>("CheckpointExample",
TypeInformation.of(Integer.class), 0);
            state = getRuntimeContext().getState(descriptor);
        }
    }
}
```

# Ejecutando el ejemplo y simulando el fracaso.

Para poder comprobar los puntos de control, debe iniciar un `cluster` . La forma más fácil es usar el script `start-cluster.sh` en el directorio `flink/bin` :

```
start-cluster.sh
Starting cluster.
[INFO] 1 instance(s) of jobmanager are already running on virusnest.
Starting jobmanager daemon on host virusnest.
Password:
Starting taskmanager daemon on host virusnest.
```

Ahora, empaqueta tu aplicación y envíala a flink:

```
mvn clean package
flink run target/flink-checkpoints-test.jar -c CheckpointExample
```

Crea algunos datos:

```
kafka-console-producer --broker-list localhost:9092 --topic input-topic
a
b
c
^D
```

La salida debe estar disponible en `flink/logs/flink-<user>-jobmanager-0-<host>.out` . Por ejemplo:

```
tail -f flink/logs/flink-Derlin-jobmanager-0-virusnest.out
2017-03-17T08:21:51.249: (a,1)
2017-03-17T08:21:51.545: (b,2)
2017-03-17T08:21:52.363: (c,3)
```

Para probar los puntos de control, simplemente elimine el administrador de tareas (esto simulará una falla), genere algunos datos e inicie uno nuevo:

```
# killing the taskmanager
ps -ef | grep -i taskmanager
kill <taskmanager PID>

# starting a new taskmanager
flink/bin/taskmanager.sh start
```

**Nota:** al iniciar un nuevo administrador de tareas, utilizará otro archivo de registro, a saber, `flink/logs/flink-<user>-jobmanager-1-<host>.out` (observe el incremento de enteros).

## Que esperar

- *Puntos de control deshabilitados* : si produce datos durante la falla, definitivamente se perderán. Pero sorprendentemente, los contadores estarán en lo cierto!
- *Puntos de control habilitados* : ya no hay pérdida de datos (y contadores correctos).
- *puntos de control con el modo al menos una vez* : puede ver duplicados, especialmente si establece un intervalo de punto de control en un número alto y mata al administrador de tareas varias veces

Lea Punto de control en línea: https://riptutorial.com/es/apache-flink/topic/9465/punto-de-control

# Capítulo 6: Puntos de ahorro y puntos de control externalizados

## Introducción

Los puntos de ahorro son *puntos de control "guardados", almacenados externamente,* que nos permiten reanudar un programa de estado con un enlace permanente después de una falla permanente, una cancelación o una actualización del código. Antes de Flink 1.2 y la introducción de *puntos* de *control externalizados* , *los puntos de* salvaguarda debían activarse explícitamente.

## Examples

### Puntos de salvaguardia: requisitos y notas preliminares.

Un punto guardado almacena dos cosas: (a) las posiciones de todas las fuentes de datos, (b) los estados de los operadores. Los puntos de salvaguarda son útiles en muchas circunstancias:

- ligeras actualizaciones de código de aplicación
- Actualización de flink
- cambios en el paralelismo
- ...

A partir de la **versión 1.3** (también válida para versiones anteriores):

- El punto de control **debe estar habilitado** para que los puntos de salvaguarda sean posibles. Si olvida habilitar explícitamente el punto de control mediante:

  ```
  StreamExecutionEnvironment env = StreamExecutionEnvironment.getExecutionEnvironment();
  env.enableCheckpointing(checkpointInterval);
  ```

  conseguirás:

  ```
  java.lang.IllegalStateException: Checkpointing disabled. You can enable it via the
  execution environment of your job
  ```

- cuando se utilizan las operaciones de la ventana, es crucial usar el tiempo de evento (vs ingestión o tiempo de procesamiento) para obtener resultados adecuados;

- para poder actualizar un programa y reutilizar los puntos de guardado, se **debe configurar el uid manual** . Esto se debe a que, de manera predeterminada, Flink cambia el UID del operador después de cualquier cambio en su código;

- Los operadores encadenados se identifican por el ID de la primera tarea. No es posible asignar manualmente una ID a una tarea encadenada intermedia, por ejemplo, en la cadena [a -> b -> c] solo a puede tener su ID asignada manualmente, pero no b o c. Para evitar

esto, puede definir manualmente las cadenas de tareas. Si confía en la asignación automática de ID, un cambio en el comportamiento de encadenamiento también cambiará los ID (consulte el punto anterior).

Más información está disponible en las preguntas frecuentes .

### Puntos de salvaguarda

## Configuración

La configuración se encuentra en el archivo `flink/conf/flink-conf.yaml` (en Mac OSX a través de homebrew, es `/usr/local/Cellar/apache-flink/1.1.3/libexec/conf/flink-conf.yaml` ).

**Flink <1.2** : La configuración es muy similar a la configuración de los puntos de control (tema disponible). La única diferencia es que no tiene sentido definir un back-end de punto de salvaguarda en memoria, ya que necesitamos los puntos de salvaguarda para persistir después del cierre de Flink.

```
# Supported backends: filesystem, <class-name-of-factory>
savepoints.state.backend: filesystem
```

```
# Use "hdfs://" for HDFS setups, "file://" for UNIX/POSIX-compliant file systems,
# (or any local file system under Windows), or "S3://" for S3 file system.
# Note: must be accessible from the JobManager and all TaskManagers !
savepoints.state.backend.fs.checkpointdir: file:///tmp/flink-backend/savepoints
```

*Nota* : Si no especifica un backend, el backend predeterminado es *jobmanager* , lo que significa que sus puntos de *salvamento* desaparecerán una vez que el clúster se cierre. Esto es útil para la depuración solamente.

**Flink 1.2+** : como se explica en este ticket de jira , permitir que se guarde un punto de salvaguarda en la memoria del administrador de tareas no tiene mucho sentido. Desde Flink 1.2, los puntos de guardado se almacenan necesariamente en archivos. La configuración anterior ha sido reemplazada por:

```
# Default savepoint target directory
state.savepoints.dir: hdfs:///flink/savepoints
```

## Uso

### Obtención de la identificación del trabajo

Para activar un punto de salvaguarda, todo lo que necesita es el ID de trabajo de la aplicación. La ID del trabajo se imprime en la línea de comando cuando inicia el trabajo o se puede recuperar más tarde utilizando la `flink list` :

```
flink list
Retrieving JobManager.
```

```
Using address localhost/127.0.0.1:6123 to connect to JobManager.
------------------ Running/Restarting Jobs -------------------
17.03.2017 11:44:03 : 196b8ce6788d0554f524ba747c4ea54f : CheckpointExample (RUNNING)
--------------------------------------------------------------
No scheduled jobs.
```

**Disparando un punto de salvaguarda**

Para activar un punto de salvaguarda, use `flink savepoint <jobID>` :

```
flink savepoint 196b8ce6788d0554f524ba747c4ea54f
Retrieving JobManager.
Using address /127.0.0.1:6123 to connect to JobManager.
Triggering savepoint for job 196b8ce6788d0554f524ba747c4ea54f.
Waiting for response...
Savepoint completed. Path: file:/tmp/flink-backend/savepoints/savepoint-a40111f915fc
You can resume your program from this savepoint with the run command.
```

Tenga en cuenta que también puede proporcionar un directorio de destino como segundo argumento, anulará el predeterminado definido en `flink/bin/flink-conf.yaml` .

En Flink 1.2+, también es posible cancelar un trabajo Y hacer un punto de salvaguarda al mismo tiempo, usando la opción `-s` :

```
flink cancel -s 196b8ce6788d0554f524ba747c4ea54f # use default savepoints dir
flink cancel -s hdfs:///savepoints 196b8ce6788d0554f524ba747c4ea54f # specify target dir
```

*Nota* : es posible mover un punto de salvaguarda, ¡pero no renombrarlo!

**Reanudando desde un punto de salvaguarda**

Para reanudar desde un punto de guardado específico, use la opción `-s [savepoint-dir]` del comando `flink run` :

```
flink run -s /tmp/flink-backend/savepoints/savepoint-a40111f915fc app.jar
```

# Especificando el UID del operador

Para poder reanudar desde un punto de guardado después de un cambio de código, debe asegurarse de que el nuevo código use el mismo UID para el operador. Para asignar manualmente un UID, llame a la función `.uid(<name>)` justo después del operador:

```
env
    .addSource(source)
    .uid(className + "-KafkaSource01")
    .rebalance()
    .keyBy((node) -> node.get("key").asInt())
    .flatMap(new StatefulMapper())
    .uid(className + "-StatefulMapper01")
    .print();
```

### Puntos de control externos (Flink 1.2+)

Antes de la versión 1.2, la única manera de mantener el estado / retener un punto de control después de una terminación / cancelación / falla persistente del trabajo era a través de un punto de salvaguarda, que se activa manualmente. La versión 1.2 introdujo puntos de control persistentes.

Los puntos de control persistentes se comportan de manera muy similar a los puntos de control periódicos regulares, excepto por las siguientes diferencias:

1. Persisten sus metadatos en un almacenamiento persistente (como puntos de guardado).
2. No se descartan cuando el trabajo de propietario falla de forma permanente. Además, se pueden configurar para que no se desechen cuando se cancela el trabajo.

Por lo tanto, es muy similar a los puntos de salvaguarda; de hecho, los puntos de guardado son simplemente puntos de control externalizados con un poco más de información.

*Nota importante* : en este momento, el coordinador de puntos de control de Flink solo retiene el último punto de control completado con éxito. Esto significa que cada vez que se complete un nuevo punto de control, se descartará el último punto de control completado. Esto también se aplica a los puntos de control externalizados.

## Configuración

Donde se almacenan los metadatos sobre los puntos de control [externalizados] se configura en `flink-conf.yaml` (y no se puede anular mediante el código):

```
# path to the externalized checkpoints
state.checkpoints.dir: file:///tmp/flink-backend/ext-checkpoints
```

Tenga en cuenta que este directorio *solo contiene los metadatos del punto de control* necesarios para restaurar el punto de control. Los archivos de punto de control reales todavía se almacenan en su directorio configurado (es decir, la propiedad `state.bachend.fs.checkpointdir` ).

## Uso

`getCheckpointConfig()` habilitar explícitamente los puntos de control externos en el código usando el método `getCheckpointConfig()` del entorno de transmisión:

```
StreamExecutionEnvironment env = StreamExecutionEnvironment.getExecutionEnvironment();
// enable regular checkpoints
env.enableCheckpointing(5000); // every 5 sec.
// enable externalized checkpoints
env.getCheckpointConfig()

.enableExternalizedCheckpoints(CheckpointConfig.ExternalizedCheckpointCleanup.RETAIN_ON_CANCELLATION);
```

Los modos de `ExternalizedCheckpointCleanup` disponibles son:

- `RETAIN_ON_CANCELLATION` : el último punto de control y sus metadatos se mantienen en la cancelación de trabajos; Es tu responsabilidad limpiar después.
- `DELETE_ON_CANCELLATION` : el último punto de control se elimina al cancelarse, lo que significa que solo está disponible si la aplicación falla.

Para reanudar desde un punto de control externo, use la sintaxis del punto de salvaguarda. Por ejemplo:

```
flink run -s /tmp/flink-backend/ext-checkpoints/savepoint-02d0cf7e02ea app.jar
```

Lea Puntos de ahorro y puntos de control externalizados en línea: https://riptutorial.com/es/apache-flink/topic/9466/puntos-de-ahorro-y-puntos-de-control-externalizados

# Capítulo 7: Tabla API

## Examples

### Dependencias maven

Para usar la API de la tabla, agregue `flink-table` como una dependencia de Maven (además de `flink-clients` y `flink-core` ):

```
<dependency>
    <groupId>org.apache.flink</groupId>
    <artifactId>flink-table_2.11</artifactId>
    <version>1.1.4</version>
</dependency>
```

Asegúrese de que la versión de scala (aquí 2.11) sea compatible con su sistema.

### Agregación simple desde un CSV

Dado el archivo CSV `peoples.csv` :

```
1,Reed,United States,Female
2,Bradley,United States,Female
3,Adams,United States,Male
4,Lane,United States,Male
5,Marshall,United States,Female
6,Garza,United States,Male
7,Gutierrez,United States,Male
8,Fox,Germany,Female
9,Medina,United States,Male
10,Nichols,United States,Male
11,Woods,United States,Male
12,Welch,United States,Female
13,Burke,United States,Female
14,Russell,United States,Female
15,Burton,United States,Male
16,Johnson,United States,Female
17,Flores,United States,Male
18,Boyd,United States,Male
19,Evans,Germany,Male
20,Stephens,United States,Male
```

Queremos contar personas por país y por país + género:

```
public class TableExample{
    public static void main( String[] args ) throws Exception{
        // create the environments
        final ExecutionEnvironment env = ExecutionEnvironment.getExecutionEnvironment();
        final BatchTableEnvironment tableEnv = TableEnvironment.getTableEnvironment( env );

        // get the path to the file in resources folder
        String peoplesPath = TableExample.class.getClassLoader().getResource( "peoples.csv"
```

```
).getPath();
        // load the csv into a table
        CsvTableSource tableSource = new CsvTableSource(
                peoplesPath,
                "id,last_name,country,gender".split( "," ),
                new TypeInformation[]{ Types.INT(), Types.STRING(), Types.STRING(),
Types.STRING() } );
        // register the table and scan it
        tableEnv.registerTableSource( "peoples", tableSource );
        Table peoples = tableEnv.scan( "peoples" );

        // aggregation using chain of methods
        Table countriesCount = peoples.groupBy( "country" ).select( "country, id.count" );
        DataSet<Row> result1 = tableEnv.toDataSet( countriesCount, Row.class );
        result1.print();

        // aggregation using SQL syntax
        Table countriesAndGenderCount = tableEnv.sql(
                "select country, gender, count(id) from peoples group by country, gender" );

        DataSet<Row> result2 = tableEnv.toDataSet( countriesAndGenderCount, Row.class );
        result2.print();
    }
}
```

Los resultados son:

```
Germany,2
United States,18

Germany,Male,1
United States,Male,11
Germany,Female,1
United States,Female,7
```

## Ejemplo de unir tablas

Además de `peoples.csv` (ver *agregación simple de un CSV* ) tenemos dos CSV más que representan productos y ventas.

`sales.csv` (people_id, product_id):

```
19,5
6,4
10,4
2,4
8,1
19,2
8,4
5,5
13,5
4,4
6,1
3,3
8,3
17,2
6,2
```

```
1,2
3,5
15,5
3,3
6,3
13,2
20,4
20,2
```

`products.csv` (id, nombre, precio):

```
1,Loperamide,47.29
2,pain relief pm,61.01
3,Citalopram,48.13
4,CTx4 Gel 5000,12.65
5,Namenda,27.67
```

Queremos obtener el nombre y el producto por cada venta de más de 40 $:

```
public class SimpleJoinExample{
    public static void main( String[] args ) throws Exception{

        final ExecutionEnvironment env = ExecutionEnvironment.getExecutionEnvironment();
        final BatchTableEnvironment tableEnv = TableEnvironment.getTableEnvironment( env );

        String peoplesPath = TableExample.class.getClassLoader().getResource( "peoples.csv"
).getPath();
        String productsPath = TableExample.class.getClassLoader().getResource( "products.csv"
).getPath();
        String salesPath = TableExample.class.getClassLoader().getResource( "sales.csv"
).getPath();

        Table peoples = csvTable(
                tableEnv,
                "peoples",
                peoplesPath,
                "pe_id,last_name,country,gender",
                new TypeInformation[]{ Types.INT(), Types.STRING(), Types.STRING(),
Types.STRING() } );

        Table products = csvTable(
                tableEnv,
                "products",
                productsPath,
                "prod_id,product_name,price",
                new TypeInformation[]{ Types.INT(), Types.STRING(), Types.FLOAT() } );

        Table sales = csvTable(
                tableEnv,
                "sales",
                salesPath,
                "people_id,product_id",
                new TypeInformation[]{ Types.INT(), Types.INT() } );

        // here is the interesting part:
        Table join = peoples
                .join( sales ).where( "pe_id = people_id" )
                .join( products ).where( "product_id = prod_id" )
                .select( "last_name, product_name, price" )
```

```
                .where( "price < 40" );

        DataSet<Row> result = tableEnv.toDataSet( join, Row.class );
        result.print();

    }//end main


    public static Table csvTable( BatchTableEnvironment tableEnv, String name, String path,
String header,
                                  TypeInformation[]
                                          typeInfo ){
        CsvTableSource tableSource = new CsvTableSource( path, header.split( "," ), typeInfo);
        tableEnv.registerTableSource( name, tableSource );
        return tableEnv.scan( name );
    }

}//end class
```

Tenga en cuenta que es importante usar nombres diferentes para cada columna, de lo contrario, flink se quejará de "nombres ambiguos en la combinación".

Resultado:

```
Burton,Namenda,27.67
Marshall,Namenda,27.67
Burke,Namenda,27.67
Adams,Namenda,27.67
Evans,Namenda,27.67
Garza,CTx4 Gel 5000,12.65
Fox,CTx4 Gel 5000,12.65
Nichols,CTx4 Gel 5000,12.65
Stephens,CTx4 Gel 5000,12.65
Bradley,CTx4 Gel 5000,12.65
Lane,CTx4 Gel 5000,12.65
```

### Usando sumideros externos

Una tabla se puede escribir en un TableSink, que es una interfaz genérica para admitir diferentes formatos y sistemas de archivos. Una tabla de lotes solo se puede escribir en un `BatchTableSink` , mientras que una tabla de transmisión requiere un `StreamTableSink` .

Actualmente, flink ofrece solo la interfaz `CsvTableSink` .

## Uso

En los ejemplos anteriores, reemplace:

```
DataSet<Row> result = tableEnv.toDataSet( table, Row.class );
result.print();
```

con:

```
TableSink sink = new CsvTableSink("/tmp/results", ",");
// write the result Table to the TableSink
table.writeToSink(sink);
// start the job
env.execute();
```

`/tmp/results` es una carpeta, porque flink realiza operaciones paralelas. Por lo tanto, si tiene 4 procesadores, es probable que tenga 4 archivos en la carpeta de resultados.

Además, tenga en cuenta que llamamos a `env.execute()` explícitamente: esto es necesario para iniciar un trabajo de flink, pero en los ejemplos anteriores, `print()` hizo por nosotros.

Lea Tabla API en línea: https://riptutorial.com/es/apache-flink/topic/8966/tabla-api

# Creditos

| S. No | Capítulos | Contributors |
|---|---|---|
| 1 | Empezando con el apache-flink | Community, Derlin, vdep |
| 2 | Cómo definir un esquema de serialización (des) personalizado | Derlin |
| 3 | Consumir datos de Kafka. | alpinegizmo, Derlin |
| 4 | explotación florestal | Derlin |
| 5 | Punto de control | Derlin |
| 6 | Puntos de ahorro y puntos de control externalizados | Derlin |
| 7 | Tabla API | Derlin |