eBook Gratuit

APPRENEZ apache-flink

eBook gratuit non affilié créé à partir des **contributeurs de Stack Overflow.**

#apache-flink

Table des matières

À propos

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

Chapitre 1: Démarrer avec apache-flink

Remarques

Cette section fournit une vue d'ensemble de ce qu'est apache-flink et pourquoi un développeur peut vouloir l'utiliser.

Il devrait également mentionner tous les grands sujets dans apache-flink, et établir un lien avec les sujets connexes. Comme la documentation pour apache-flink est nouvelle, vous devrez peut-être créer des versions initiales de ces rubriques connexes.

Examples

Vue d'ensemble et exigences

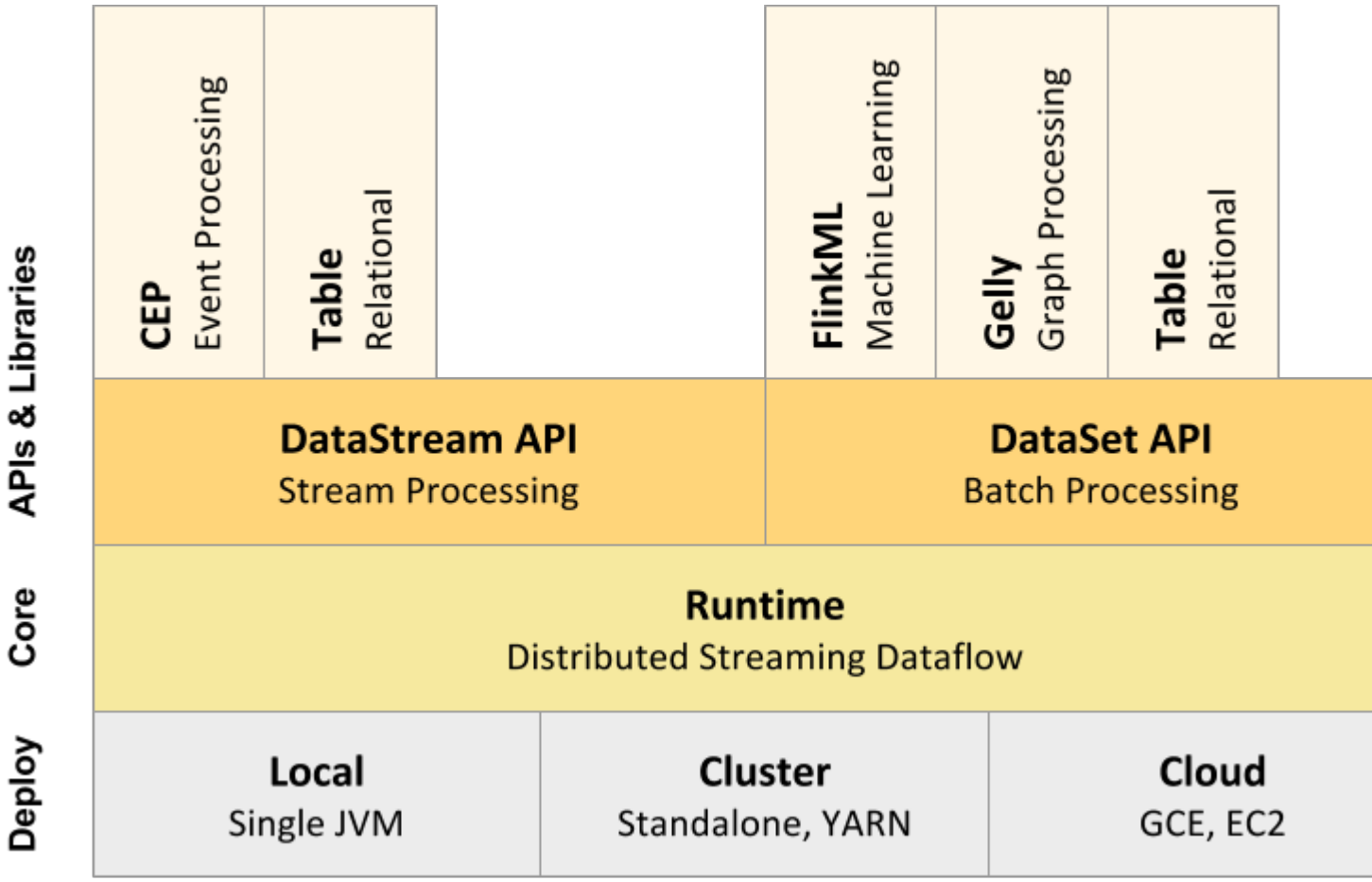
Qu'est-ce que Flink

Comme Apache Hadoop et Apache Spark , Apache Flink est un framework open source piloté par la communauté pour les analyses Big Data distribuées. Écrit en Java, Flink a des API pour Scala, Java et Python, permettant des analyses en continu en temps réel et par lots.

Exigences

- un environnement de type UNIX, tel que Linux, Mac OS X ou Cygwin;
- Java 6.X ou ultérieur;
- [facultatif] Maven 3.0.4 ou ultérieur.

Empiler



Environnements d'exécution

Apache Flink est un système de traitement de données et **une alternative au composant MapReduce de Hadoop** . Il est livré avec son *propre runtime* plutôt que de construire sur MapReduce. En tant que tel, il peut fonctionner complètement indépendamment de l'écosystème Hadoop.

`ExecutionEnvironment` est le contexte dans lequel un programme est exécuté. Vous pouvez utiliser différents environnements en fonction de vos besoins.

1. *Environnement JVM* : Flink peut s'exécuter sur une seule machine virtuelle Java, ce qui permet aux utilisateurs de tester et de déboguer les programmes Flink directement à partir de leur IDE. Lorsque vous utilisez cet environnement, vous n'avez besoin que des dépendances Maven correctes.

2. *Environnement local* : pour pouvoir exécuter un programme sur une instance Flink en cours d'exécution (pas depuis votre IDE), vous devez installer Flink sur votre ordinateur. Voir la configuration locale .

3. *Environnement de cluster* : exécuter Flink de manière totalement distribuée nécessite un cluster autonome ou un cluster de fils. Reportez-vous à la page de configuration du cluster ou à ce partage de diapositives pour plus d'informations. Important: le `2.11` dans le nom de l'artefact est la *version scala* , veillez à faire correspondre celle de votre système.

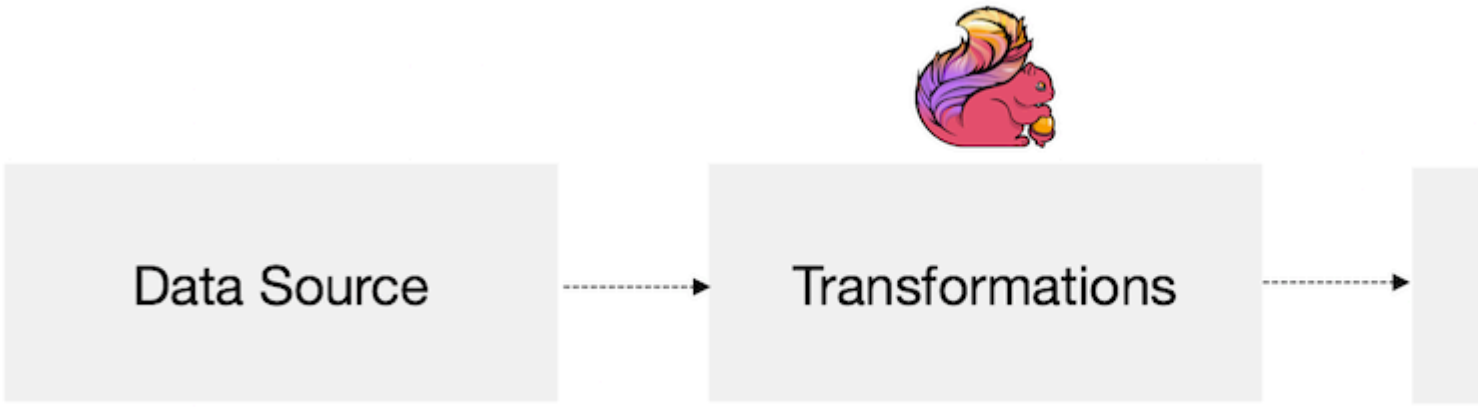
Apis

Flink peut être utilisé pour le traitement par flux ou par lots. Ils offrent trois API:

- **DataStream API** : traitement des flux, c'est-à-dire transformations (filtres, fenêtres temporelles, agrégations) sur des flux de données illimités.
- **DataSet API** : traitement par lots, c'est-à-dire transformations sur des ensembles de données.
- **Table API** : un langage d'expression de type SQL (comme les dataframes dans Spark) pouvant être intégré à la fois aux applications batch et streaming.

Blocs de construction

Au niveau le plus élémentaire, Flink est constitué de sources, de transformations et de puits.



Au niveau le plus élémentaire, un programme Flink est composé de:

- **Source de données** : données entrantes traitées par Flink
- **Transformations** : étape de traitement lorsque Flink modifie les données entrantes
- **Dissipateur de données** : où Flink envoie des données après traitement

Les sources et les puits peuvent être des fichiers locaux / HDFS, des bases de données, des files d'attente de messages, etc. De nombreux connecteurs tiers sont déjà disponibles ou vous pouvez facilement créer les vôtres.

Configuration d'exécution locale

0. Assurez-vous d'avoir Java 6 ou supérieur et que la variable d'environnement `JAVA_HOME` est définie.

1. Téléchargez le dernier fichier binaire flink ici :

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

Si vous ne prévoyez pas de travailler avec Hadoop, choisissez la version hadoop 1. Notez également la version de scala que vous téléchargez, afin que vous puissiez ajouter les dépendances maven correctes dans vos programmes.

2. commencer flink:

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

Flink est déjà configuré pour s'exécuter localement. Pour vous assurer que flink fonctionne, vous pouvez inspecter les journaux dans `flink/log/` ou ouvrir l'interface de flink jobManager qui s'exécute sur `http://localhost:8081` .

3. stop flink:

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

Configuration de l'environnement Flink

Pour exécuter un programme flink à partir de votre IDE (nous pouvons utiliser Eclipse ou Intellij IDEA (preffered)), vous avez besoin de deux dépendances: `flink-java` / `flink-scala` et `flink-clients` (à partir de février 2016). Ces JARS peuvent être ajoutés en utilisant Maven et SBT (si vous utilisez 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>
```

- Nom **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"
)
```

important : le `2.11` dans le nom de l'artefact est la *version scala* , assurez-vous de correspondre à celle que vous avez sur votre système.

WordCount - API de table

Cet exemple est identique à *WordCount* , mais utilise l'API Table. Voir *WordCount* pour plus de détails sur l'exécution et les résultats.

Maven

Pour utiliser l'API de table, ajoutez `flink-table` tant que dépendance maven:

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

Le code

```
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();
    }
}
```

Remarque : Pour une version utilisant Java <8, remplacez le lambda par une classe anonyme:

```
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 );
                }
            }
        }
    } );
```

WordCount

Maven

Ajoutez les dépendances `flink-java` et `flink-client` (comme expliqué dans l'exemple de *configuration de l'environnement JVM*).

Le code

```
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 vous utilisez Java 8, vous pouvez remplacer `.flatmap(new LineSplitter())` par une expression 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 );
```

Exécution

De l'IDE : appuyez simplement sur *exécuter* dans votre IDE. Flink créera un environnement à l'intérieur de la JVM.

À partir de la ligne de commande flink : pour exécuter le programme en utilisant un environnement local autonome, procédez comme suit:

1. s'assurer que flink est en cours d'exécution (`flink/bin/start-local.sh`);
2. créer un fichier jar (`maven package`);
3. Utilisez l'outil de ligne de commande `flink` (dans le dossier `bin` de votre installation flink) pour lancer le programme:

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

   L'option `-c` vous permet de spécifier la classe à exécuter. Ce n'est pas nécessaire si le fichier jar est exécutable / définit une classe principale.

Résultat

```
(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 diffusion en continu

Cet exemple est identique à *WordCount* , mais utilise l'API Table. Voir *WordCount* pour plus de détails sur l'exécution et les résultats.

Maven

Pour utiliser l'API de diffusion, ajoutez `flink-streaming` tant que dépendance maven:

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

Le code

```
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();
    }
}
```

Lire Démarrer avec apache-flink en ligne: https://riptutorial.com/fr/apache-flink/topic/5798/demarrer-avec-apache-flink

Chapitre 2: API de table

Examples

Dépendances Maven

Pour utiliser l'API Table, ajoutez `flink-table` tant que dépendance `flink-clients` (en plus `flink-clients` `flink-core` et `flink-core`):

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

Assurez-vous que la version scala (ici 2.11) est compatible avec votre système.

Agrégation simple à partir d'un CSV

Compte tenu du fichier 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
```

Nous voulons compter les personnes par pays et par pays + genre:

```
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();
    }
}
```

Les résultats sont les suivants:

```
Germany,2
United States,18

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

Exemple de jointure de tables

En plus de `peoples.csv` (voir l' *agrégation simple à partir d'un fichier CSV*), nous avons deux autres *CSV* représentant des produits et des ventes.

`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, nom, prix):

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

Nous voulons obtenir le nom et le produit pour chaque vente de plus 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
```

Notez qu'il est important d'utiliser des noms différents pour chaque colonne, sinon flink se plaindra des "noms ambigus dans la jointure".

Résultat:

```
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
```

Utiliser des éviers externes

Une table peut être écrite sur un TableSink, qui est une interface générique pour prendre en charge différents formats et systèmes de fichiers. Une table de commandes ne peut être écrite que dans un `BatchTableSink` , tandis qu'une table de transmission en continu nécessite un `StreamTableSink` .

Actuellement, flink propose uniquement l'interface `CsvTableSink` .

Usage

Dans les exemples ci-dessus, remplacez:

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

avec:

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

`/tmp/results` est un dossier, car flink effectue des opérations parallèles. Par conséquent, si vous avez 4 processeurs, vous aurez probablement 4 fichiers dans le dossier de résultats.

De plus, notez que nous appelons explicitement `env.execute()` : cela est nécessaire pour démarrer une tâche flink, mais dans les exemples précédents, `print()` fait pour nous.

Lire API de table en ligne: https://riptutorial.com/fr/apache-flink/topic/8966/api-de-table

Chapitre 3: Comment définir un schéma de sérialisation personnalisé

Introduction

Certains schémas (Kafka, RabbitMQ) utilisent des schémas pour transformer des messages en objets Java et inversement.

Examples

Exemple de schéma personnalisé

Pour utiliser un schéma personnalisé, il vous suffit d'implémenter l'une des interfaces `SerializationSchema` ou `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 classe `MyMessage` est définie comme suit:

```
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());
    }
}
```

Lire Comment définir un schéma de sérialisation personnalisé en ligne: https://riptutorial.com/fr/apache-flink/topic/9004/comment-definir-un-schema-de-serialisation-personnalise

Chapitre 4: Consommez des données de Kafka

Examples

Exemple KafkaConsumer

`FlinkKafkaConsumer` vous permet de consommer des données d'un ou plusieurs sujets kafka.

versions

Le consommateur à utiliser dépend de votre distribution kafka.

- `FlinkKafkaConsumer08` : utilise l'ancienne API `SimpleConsumer` de Kafka. Les compensations sont gérées par Flink et engagées dans zookeeper.
- `FlinkKafkaConsumer09` : utilise la nouvelle API Consumer de Kafka, qui gère automatiquement les décalages et le rééquilibrage.
- `FlinkKafkaProducer010` : ce connecteur prend en charge les messages Kafka avec horodatage à la fois pour la production et la consommation (utile pour les opérations de fenêtre).

usage

Les binaires ne font pas partie du noyau de flink, vous devez donc les importer:

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

Le constructeur prend trois arguments:

- un ou plusieurs sujets à lire
- un schéma de désérialisation indiquant à Flink comment interpréter / décoder les messages
- Propriétés de configuration du consommateur kafka. Ce sont les mêmes que pour un consommateur de kafka "régulier". Les minimum requis sont:
    - `bootstrap.servers` : une liste de courtiers Kafka séparés par des virgules sous la forme ip: port. Pour la version 8, utilisez plutôt `zookeeper.connect` (liste des serveurs zookeeper)
    - `group.id` : l'id du groupe de consommateurs (voir la documentation de kafka pour plus de détails)

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))
```

Au cours du développement, vous pouvez utiliser les propriétés kafka `enable.auto.commit=false` et `auto.offset.reset=earliest` pour récupérer les mêmes données à chaque lancement de votre programme.

Tolérance de panne

Comme expliqué dans les docs ,

> Lorsque les points de contrôle de Flink sont activés, le consom- mateur Flink Kafka consomme les enregistrements d'un sujet et vérifie régulièrement tous ses décalages de Kafka, ainsi que l'état des autres opérations, de manière cohérente. En cas d'échec du travail, Flink restaurera le programme de diffusion en continu à l'état du dernier point de contrôle et consommera à nouveau les enregistrements de Kafka, en commençant par les décalages stockés dans le point de contrôle.
>
> L'intervalle des points de contrôle du dessin définit donc combien le programme peut devoir revenir au maximum, en cas de panne.

Pour utiliser les consommateurs Kafka à tolérance de pannes, vous devez activer les points de contrôle sur l'environnement d'exécution à l'aide de la méthode `enableCheckpointing` :

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

Schémas de désérialisation intégrés

SimpleStringSchema : `SimpleStringSchema` désérialise le message sous forme de chaîne. Si vos messages ont des clés, celles-ci seront ignorées.

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

JSONDeserializationSchema

`JSONDeserializationSchema` désérialise les messages au format json à l'aide de *jackson* et renvoie un flux d'objets `com.fasterxml.jackson.databind.node.ObjectNode` . Vous pouvez ensuite utiliser la `.get("property")` pour accéder aux champs. Encore une fois, les clés sont ignorées.

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

JSONKeyValueDeserializationSchema

`JSONKeyValueDeserializationSchema` est très similaire à la précédente, mais traite des messages avec des clés ET des valeurs codées json.

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

Le `ObjectNode` renvoyé contient les champs suivants:

- `key` : tous les champs présents dans la clé
- `value` : tous les champs de message
- (facultatif) `metadata` : expose le `offset` , la `partition` et le `topic` du message (transmettez `true` au constructeur afin de récupérer également les métadonnées).

Par exemple:

```
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
```

Sera décodé comme:

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

Kafka partitions et parallélisme Flink

Dans kafka, chaque consommateur du même groupe de consommateurs se voit attribuer une ou plusieurs partitions. Notez qu'il n'est pas possible pour deux consommateurs de consommer de la même partition. Le nombre de consommateurs de flink dépend du parallélisme de flink (1 par défaut).

Il y a trois cas possibles:

1. **partitions kafka == parallélisme flink** : ce cas est idéal, car chaque consommateur prend en charge une partition. Si vos messages sont équilibrés entre les partitions, le travail sera uniformément réparti entre les opérateurs de flink;

2. **partitions kafka <parallélisme flink** : certaines instances de flink ne recevront aucun message. Pour éviter cela, vous devez appeler le `rebalance` sur votre flux d'entrée *avant toute opération* , ce qui entraîne le re-partitionnement des données:

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

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

3. **partitions kafka> parallélisme flink** : dans ce cas, certaines instances géreront plusieurs partitions. Encore une fois, vous pouvez utiliser le `rebalance` pour répartir les messages de manière égale entre les travailleurs.

Lire Consommez des données de Kafka en ligne: https://riptutorial.com/fr/apache-flink/topic/9003/consommez-des-donnees-de-kafka

Chapitre 5: enregistrement

Introduction

Cette rubrique montre comment utiliser et configurer la journalisation (log4j) dans les applications Flink.

Examples

Utiliser un enregistreur dans votre code

Ajoutez la dépendance `slf4j` à votre `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>
```

Créez un objet de journalisation à utiliser dans votre classe:

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

Dans les classes devant être sérialisées, telles que les sous-classes de `RichMapFunction` , n'oubliez pas de déclarer `LOGGER` comme `transient` :

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

Dans votre code, utilisez `LOGGER` comme d'habitude. Utilisez des espaces réservés (`{}`) pour formater des objets, par exemple:

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

Configuration de la journalisation

Mode local

En mode local, par exemple lorsque vous exécutez votre application à partir d'un IDE, vous pouvez configurer `log4j` comme d'habitude, c'est-à-dire en mettant un `log4j.properties` disponible dans le `log4j.properties` . Une méthode simple dans maven consiste à créer `log4j.properties` dans le dossier `src/main/resources` . Voici un exemple:

```
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
```

Mode autonome

> En mode autonome, la configuration réelle utilisée n'est pas celle de votre fichier `jar` . Cela est dû au fait que Flink possède ses propres fichiers de configuration, qui prévalent sur les vôtres.

Fichiers par défaut : Flink est livré avec les fichiers de propriétés par défaut suivants:

- `log4j-cli.properties` : Utilisé par le client de ligne de commande Flink (par `flink run`) (pas le code exécuté sur le cluster)
- `log4j-yarn-session.properties` : Utilisé par le client de ligne de commande Flink lors du démarrage d'une session `yarn-session.sh` (`yarn-session.sh`)
- `log4j.properties` : `log4j.properties` JobManager / Taskmanager (autonomes et YARN)

Notez que `${log.file}` par défaut pour `flink/log` . Il peut être remplacé dans `flink-conf.yaml` , en définissant `env.log.dir` ,

> `env.log.dir` définit le répertoire dans lequel les journaux Flink sont enregistrés. Ce doit

être un chemin absolu.

Emplacement du journal : les journaux sont *locaux* , c.-à-d. Qu'ils sont produits dans la ou les machines exécutant le ou les JobManager (s) / Taskmanager (s).

Fil : lorsque vous utilisez Flink on Yarn, vous devez vous fier aux capacités de journalisation de Hadoop YARN. La fonctionnalité la plus utile pour cela est l' agrégation de journaux YARN . Pour l'activer, définissez la `yarn.log-aggregation-enable` sur `true` dans le `yarn-site.xml file` . Une fois cette option activée, vous pouvez récupérer tous les fichiers journaux d'une session YARN (ayant échoué) à l'aide de:

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

Malheureusement, les journaux ne sont disponibles *qu'une fois la session arrêtée* , par exemple après un échec.

Utiliser différentes configurations pour chaque application

Au cas où vous auriez besoin de paramètres différents pour vos différentes applications, il n'ya pas de solution simple (à partir de Flink 1.2).

Si vous utilisez le mode *one-yarn-cluster-per-job* de flink (c'est-à-dire que vous lancez vos scripts avec: `flink run -m yarn-cluster` ...), voici une solution:

1. créer un répertoire `conf` quelque part près de votre projet
2. créer des liens symboliques pour tous les fichiers en `flink/conf` :

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

3. remplacez le lien symbolique `log4j.properties` (ou tout autre fichier que vous souhaitez modifier) par votre propre configuration
4. avant de lancer votre travail, exécutez

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

Selon votre version de flink, vous devrez peut-être modifier le fichier `flink/bin/config.sh` . Si votre course à travers cette ligne:

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

le changer avec:

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

Solution Flink-on-Yarn: obtenez les journaux en temps réel avec rsyslog

Par défaut, Yarn ne regroupe pas les journaux avant la fin d'une application, ce qui peut être problématique avec les jobs de streaming qui ne se terminent même pas.

Une solution consiste à utiliser `rsyslog` , disponible sur la plupart des machines Linux.

Tout d'abord, autorisez les requêtes udp entrantes en décommentant les lignes suivantes dans `/etc/rsyslog.conf` :

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

Modifiez votre `log4j.properties` (voir les autres exemples sur cette page) pour utiliser `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
```

La mise en page est importante car rsyslog traite une nouvelle ligne comme une nouvelle entrée de journal. Au-dessus, les nouvelles lignes (dans les superpositions par exemple) seront ignorées. Si vous voulez vraiment que les journaux multilignes / à onglets fonctionnent "normalement", éditez `rsyslog.conf` et ajoutez:

```
$EscapeControlCharactersOnReceive off
```

L'utilisation de `myTag:` au début de la `conversionPattern` est utile si vous souhaitez rediriger tous vos journaux dans un fichier spécifique. Pour ce faire, éditez le `rsyslog.conf` et ajoutez la règle suivante:

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

Lire enregistrement en ligne: https://riptutorial.com/fr/apache-flink/topic/9713/enregistrement

Chapitre 6: Point de contrôle

Introduction

(testé sur Flink 1.2 et ci-dessous)

Chaque fonction, source ou opérateur dans Flink peut être dynamique. Les points de contrôle permettent à Flink de récupérer l'état et les positions dans les flux pour donner à l'application la même sémantique qu'une exécution sans défaillance. C'est le mécanisme derrière les garanties de *tolérance aux fautes* et de traitement *exact* .

Lisez cet article pour comprendre les internes.

Remarques

Les points de contrôle ne sont utiles que lorsqu'une défaillance se produit dans le cluster, par exemple lorsqu'un gestionnaire de tâches échoue. Ils ne persistent pas après que le travail a échoué ou a été annulé.

Pour pouvoir reprendre un travail avec état après une défaillance / annulation, consultez les **points** de **sauvegarde** ou **les points de contrôle externalisés (flink 1.2+)** .

Examples

Configuration et installation

La configuration des points de contrôle se fait en deux étapes. D'abord, vous devez choisir un *backend* . Ensuite, vous pouvez spécifier l'intervalle et le mode des points de contrôle par application.

Backends

Backends disponibles

L'emplacement des points de contrôle dépend du backend configuré:

- `MemoryStateBackend` : état en mémoire, sauvegarde dans la mémoire de JobManager / ZooKeeper. Doit être utilisé uniquement pour un état minimal (par défaut, max. 5 Mo, pour stocker des décalages de Kafka par exemple) ou des tests et un débogage local.
- `FsStateBackend` : l'état est conservé en mémoire sur les TaskManagers et les instantanés d'état (c.-à-d. Les points de contrôle) sont stockés dans un système de fichiers (HDFS, DS3, système de fichiers local, ...). Cette configuration est encouragée pour les grands états ou les longues fenêtres et pour les configurations à haute disponibilité.

- `RocksDBStateBackend` : contient des données en vol dans une base de données RocksDB (par défaut) stockée dans les répertoires de données TaskManager. Lors de la vérification, toute la base de données RocksDB est écrite dans un fichier (comme ci-dessus). Comparé à FsStateBackend, il permet des états plus importants (limités uniquement par l'espace disque et la taille de la mémoire du gestionnaire de tâches), mais le débit sera inférieur (les données ne sont pas toujours en mémoire, elles doivent être chargées à partir du disque).

Notez que quel que soit le backend, les métadonnées (nombre de points de contrôle, localisation, etc.) sont toujours stockées dans la mémoire du gestionnaire de travaux et les points de contrôle **ne persisteront pas après la fin / l'annulation de l'application** .

Spécifier le backend

Vous spécifiez le backend dans la méthode `main` votre programme en utilisant:

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

Ou définissez le backend par défaut dans `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
```

Activation des points de contrôle

Chaque application doit activer explicitement les points de contrôle:

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

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

Vous pouvez éventuellement spécifier un *mode de point de contrôle* . Si ce n'est pas le cas, la valeur par défaut est *exactement une fois* :

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

Le mode checkpointing définit la cohérence que le système garantit en présence de pannes. Lorsque le point de contrôle est activé, les flux de données sont relus de sorte que les parties

perdues du traitement soient répétées. Avec `EXACTLY_ONCE` , le système dessine des points de contrôle de telle sorte qu'une reprise se comporte comme si les opérateurs / fonctions voyaient chaque enregistrement "exactement une fois". Avec `AT_LEAST_ONCE` , les points de contrôle sont dessinés de manière plus simple et rencontrent généralement des doublons lors de la récupération.

Test des points de contrôle

Le code

Voici une application flink simple utilisant un mappeur avec état avec un état géré `Integer` . Vous pouvez jouer avec les variables `checkpointEnable` , `checkpointInterval` et `checkpointMode` pour voir leur effet:

```
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);
        }
    }
}
```

Exécuter l'exemple et simuler une panne

Pour pouvoir vérifier les points de contrôle, vous devez démarrer un `cluster` . Le moyen le plus simple consiste à utiliser le script `start-cluster.sh` dans le `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.
```

Maintenant, empaquetez votre application et soumettez-la à flink:

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

Créer des données:

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

La sortie doit être disponible dans `flink/logs/flink-<user>-jobmanager-0-<host>.out` . Par exemple:

```
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)
```

Pour tester les points de contrôle, il suffit de tuer le gestionnaire de tâches (cela émule un échec), de générer des données et d'en démarrer une nouvelle:

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

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

Remarque: lors du démarrage d'un nouveau gestionnaire de tâches, il utilisera un autre fichier journal, à savoir `flink/logs/flink-<user>-jobmanager-1-<host>.out` (notez l'incrément entier).

Quoi attendre

- *points de contrôle désactivés* : si vous produisez des données pendant l'échec, ils seront définitivement perdus. Mais étonnamment, les pions auront raison!
- *points de contrôle activés* : aucune perte de données (et correcteurs).
- *points de contrôle avec le mode au moins une fois* : vous pouvez voir des doublons, en particulier si vous définissez un intervalle de point de contrôle sur un nombre élevé et si vous tuez le gestionnaire de tâches plusieurs fois

Lire Point de contrôle en ligne: https://riptutorial.com/fr/apache-flink/topic/9465/point-de-controle

Chapitre 7: Points de sauvegarde et points de contrôle externalisés

Introduction

Les points de sauvegarde sont *des points de contrôle stockés de manière externe,* qui permettent de reprendre un programme de filtrage avec état après une défaillance permanente, une annulation ou une mise à jour du code. Avant Flink 1.2 et l'introduction de *points de contrôle externalisés* , les points de sauvegarde devaient être déclenchés explicitement.

Examples

Points de sauvegarde: exigences et notes préliminaires

Un point de sauvegarde stocke deux choses: (a) les positions de toutes les sources de données, (b) les états des opérateurs. Les points de sauvegarde sont utiles dans de nombreuses circonstances:

- légères mises à jour du code d'application
- Mise à jour Flink
- changements dans le parallélisme
- ...

Depuis la **version 1.3** (également valable pour les versions antérieures):

- le point de contrôle **doit être activé** pour que les points de sauvegarde soient possibles. Si vous oubliez d'activer explicitement le point de contrôle en utilisant:

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

  tu auras:

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

- Lors de l'utilisation des opérations sur les fenêtres, il est essentiel d'utiliser l'heure des événements (vs l'ingestion ou le temps de traitement) pour obtenir des résultats corrects.

- pour pouvoir mettre à jour un programme et réutiliser les points de sauvegarde, **le code manuel doit être défini** . C'est parce que, par défaut, Flink modifie l'UID de l'opérateur après toute modification de son code.

- Les opérateurs chaînés sont identifiés par l'ID de la première tâche. Il n'est pas possible d'affecter manuellement un ID à une tâche chaînée intermédiaire, par exemple, dans la

chaîne [a -> b -> c], seul un identifiant peut être attribué manuellement, mais pas b ou c. Pour contourner ce problème, vous pouvez définir manuellement les chaînes de tâches. Si vous comptez sur l'affectation automatique d'ID, une modification du comportement de chaînage modifiera également les ID (voir le point ci-dessus).

Plus d'informations sont disponibles dans la FAQ .

Points de sauvegarde

Configuration

La configuration se trouve dans le fichier `flink/conf/flink-conf.yaml` (sous Mac OSX via homebrew, il s'agit de `/usr/local/Cellar/apache-flink/1.1.3/libexec/conf/flink-conf.yaml`).

Flink <1.2 : La configuration est très similaire à la configuration des points de contrôle (rubrique disponible). La seule différence est qu'il n'est pas logique de définir un backend de sauvegarde en mémoire, puisque les points de sauvegarde doivent persister après l'arrêt 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
```

Remarque : Si vous ne spécifiez pas de backend, le backend par défaut est *jobmanager* , ce qui signifie que vos points de sauvegarde disparaîtront une fois le cluster arrêté. Ceci est utile pour le débogage uniquement.

Flink 1.2+ : comme expliqué dans ce billet jira , il est peu judicieux d'enregistrer un point de sauvegarde dans la mémoire du gestionnaire de tâches. Depuis Flink 1.2, les points de sauvegarde sont nécessairement stockés dans des fichiers. La configuration ci-dessus a été remplacée par:

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

Usage

Obtenir l'ID du travail

Pour déclencher un point de sauvegarde, tout ce dont vous avez besoin est l'ID du travail de l'application. L'ID de travail est imprimé dans la ligne de commande lorsque vous lancez le travail ou peut être récupéré ultérieurement à l'aide de 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.
```

Déclencher un point de sauvegarde

Pour déclencher un point de sauvegarde, utilisez le `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.
```

Notez que vous pouvez également fournir un répertoire cible en second argument, il remplacera celui par défaut défini dans `flink/bin/flink-conf.yaml` .

Dans Flink 1.2+, il est également possible d'annuler un travail ET de faire un point de sauvegarde en même temps, en utilisant l'option `-s` :

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

Note : il est possible de déplacer un point de sauvegarde, mais ne le renommez pas!

Reprise d'un point de sauvegarde

Pour reprendre à partir d'un point de sauvegarde spécifique, utilisez l'option `-s [savepoint-dir]` de la commande `flink run` :

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

Spécifiant l'opérateur UID

Pour pouvoir reprendre à partir d'un point de sauvegarde après un changement de code, vous devez vous assurer que le nouveau code utilise le même UID pour l'opérateur. Pour assigner manuellement un UID, appelez la fonction `.uid(<name>)` juste après l'opérateur:

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

Points de contrôle externalisés (Flink 1.2+)

Avant 1.2, la seule façon de conserver l'état / conserver un point de contrôle après une interruption de travail / une annulation / une défaillance persistante était via un point de sauvegarde, qui est déclenché manuellement. La version 1.2 a introduit des points de contrôle persistants.

Les points de contrôle persistants se comportent très bien comme les points de contrôle périodiques réguliers, à l'exception des différences suivantes:

1. Ils conservent leurs métadonnées dans un stockage persistant (comme les points de sauvegarde).
2. Ils ne sont pas supprimés lorsque le travail propriétaire échoue de manière permanente. En outre, ils peuvent être configurés pour ne pas être supprimés lorsque le travail est annulé.

Il est donc très similaire aux points de sauvegarde; en fait, les points de sauvegarde ne sont que des points de contrôle externalisés avec un peu plus d'informations.

Remarque importante : pour le moment, le coordinateur de point de contrôle de Flink ne conserve que le dernier point de contrôle complété avec succès. Cela signifie que chaque fois qu'un nouveau point de contrôle se termine, le dernier point de contrôle terminé sera supprimé. Cela s'applique également aux points de contrôle externalisés.

Configuration

Où les métadonnées sur les points de contrôle [externalisés] sont stockées est configuré dans `flink-conf.yaml` (et ne peut pas être remplacé par du code):

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

Notez que ce répertoire *contient uniquement les métadonnées de point de contrôle* requises pour restaurer le point de contrôle. Les fichiers de points de contrôle réels sont toujours stockés dans leur répertoire configuré (par exemple, propriété `state.bachend.fs.checkpointdir`).

Usage

Vous devez activer explicitement les points de contrôle externes dans le code à l'aide de la méthode `getCheckpointConfig()` de l'environnement de diffusion en continu:

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

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

Les modes `ExternalizedCheckpointCleanup` disponibles sont les suivants:

- `RETAIN_ON_CANCELLATION` : le dernier point de contrôle et ses métadonnées sont conservés lors de l'annulation du travail; c'est votre responsabilité de nettoyer après.
- `DELETE_ON_CANCELLATION` : le dernier point de contrôle est supprimé lors de l'annulation, ce qui signifie qu'il n'est disponible que si l'application échoue.

Pour reprendre un point de contrôle externalisé, utilisez la syntaxe du point de sauvegarde. Par exemple:

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

Lire Points de sauvegarde et points de contrôle externalisés en ligne: https://riptutorial.com/fr/apache-flink/topic/9466/points-de-sauvegarde-et-points-de-controle-externalises

Crédits

S. No	Chapitres	Contributeurs
1	Démarrer avec apache-flink	Community, Derlin, vdep
2	API de table	Derlin
3	Comment définir un schéma de sérialisation personnalisé	Derlin
4	Consommez des données de Kafka	alpinegizmo, Derlin
5	enregistrement	Derlin
6	Point de contrôle	Derlin
7	Points de sauvegarde et points de contrôle externalisés	Derlin