EBook Gratuito

APPRENDIMENTO apache-flink

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

#apache-flink

Sommario

Di 1

Capitolo 1: Iniziare con apache-flink 2

Osservazioni 2

Examples 2

Panoramica e requisiti 2

Cos'è Flink 2

Requisiti 2

Pila 2

Ambienti di esecuzione 3

API 3

Costruzioni 4

Configurazione locale del runtime 4

Configurazione dell'ambiente Flink 5

WordCount - API Table 5

Maven 6

Il codice 6

WordCount 7

Maven 7

Il codice 7

Esecuzione 8

Risultato 8

WordCount - Streaming API 9

Maven 9

Il codice 9

Capitolo 2: checkpointing 11

introduzione 11

Osservazioni 11

Examples 11

Configurazione e configurazione 11

backend 11
Abilitazione dei checkpoint 12
Testare i checkpoint 13
Il codice 13
Esecuzione dell'esempio e simulazione del fallimento 14
Cosa aspettarsi 15
Capitolo 3: Come definire uno schema di serializzazione personalizzato (de) 16
introduzione 16
Examples 16
Esempio di schema personalizzato 16
Capitolo 4: Consuma i dati da Kafka 18
Examples 18
Esempio di KafkaConsumer 18
versioni 18
uso 18
Tolleranza ai guasti 19
Schemi di deserializzazione incorporati 19
Partizioni di Kafka e parallelismo di Flink 20
Capitolo 5: Punti di salvataggio e punti di controllo esterni 22
introduzione 22
Examples 22
Punti di salvataggio: requisiti e note preliminari 22
punti di salvataggio 23
Configurazione 23
uso 23
Specifica UID dell'operatore 24
Punti di controllo esterni (Flink 1.2+) 24
Configurazione 25
uso 25
Capitolo 6: registrazione 27
introduzione 27

Examples 27

Usando un logger nel tuo codice 27

Registrazione della configurazione 27

Modalità locale 27

Modalità autonoma 28

Utilizzo di diverse configurazioni per ciascuna applicazione 29

Soluzione alternativa Flink-on-Yarn: ottieni i log in tempo reale con rsyslog 29

Capitolo 7: Tabella API 32

Examples 32

Dipendenze Maven 32

Semplice aggregazione da un CSV 32

Unisciti alle tabelle di esempio 33

Utilizzando lavandini esterni 35

uso 35

Titoli di coda 37

Di

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.

The content is released under Creative Commons BY-SA, and the list of contributors to each chapter are provided in the credits section at the end of this book. Images may be copyright of their respective owners unless otherwise specified. All trademarks and registered trademarks are the property of their respective company owners.

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

Capitolo 1: Iniziare con apache-flink

Osservazioni

Questa sezione fornisce una panoramica su cosa sia Apache-flink e sul motivo per cui uno sviluppatore potrebbe volerlo utilizzare.

Dovrebbe anche menzionare qualsiasi argomento di grandi dimensioni all'interno di apache-flink e collegarsi agli argomenti correlati. Poiché la documentazione di apache-flink è nuova, potrebbe essere necessario creare versioni iniziali di tali argomenti correlati.

Examples

Panoramica e requisiti

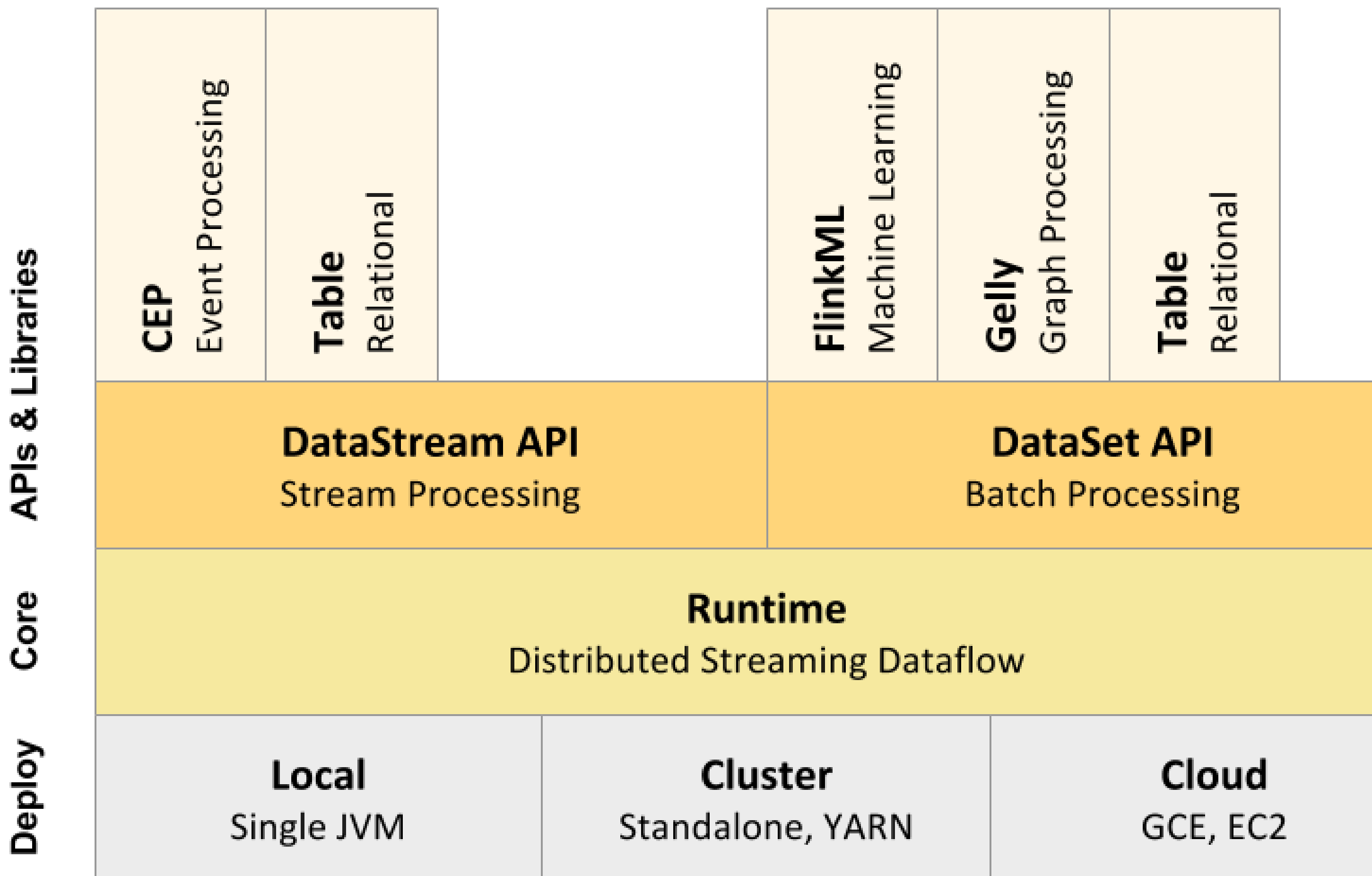
Cos'è Flink

Come Apache Hadoop e Apache Spark , Apache Flink è un framework open source guidato dalla comunità per la distribuzione di Big Data Analytics. Scritto in Java, Flink ha API per Scala, Java e Python, consentendo l'analisi di streaming Batch e Real-Time.

Requisiti

- un ambiente simile a UNIX, come Linux, Mac OS X o Cygwin;
- Java 6.X o successivo;
- [facoltativo] Maven 3.0.4 o successivo.

Pila



Ambienti di esecuzione

> Apache Flink è un sistema di elaborazione dati e **un'alternativa al componente MapReduce di Hadoop** . Viene fornito con un *proprio runtime* piuttosto che sulla base di MapReduce. Come tale, può funzionare completamente indipendentemente dall'ecosistema Hadoop.

`ExecutionEnvironment` è il contesto in cui viene eseguito un programma. Ci sono diversi ambienti che puoi usare, a seconda delle tue esigenze.

1. *Ambiente JVM* : Flink può essere eseguito su una singola Java Virtual Machine, consentendo agli utenti di testare e debugare i programmi Flink direttamente dal proprio IDE. Quando si utilizza questo ambiente, tutto ciò che serve sono le corrette dipendenze Maven.

2. *Ambiente locale* : per poter eseguire un programma su un'istanza di Flink in esecuzione (non dall'interno dell'IDE), è necessario installare Flink sul computer. Vedi setup locale .

3. *Cluster environment* : l'esecuzione di Flink in modalità completamente distribuita richiede un cluster autonomo o un filato. Vedere la pagina di configurazione del cluster o questa slide per maggiori informazioni. mportante__: il `2.11` nel nome del manufatto è la *versione scala* , assicurati di corrispondere a quello che hai sul tuo sistema.

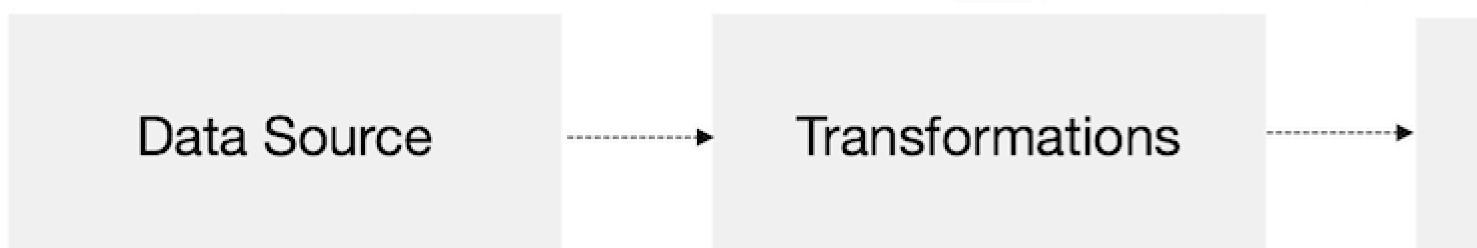
API

Flink può essere utilizzato per l'elaborazione di stream o batch. Offrono tre API:

- **API DataStream** : elaborazione stream, ovvero trasformazioni (filtri, time-windows, aggregazioni) su flussi di dati illimitati.
- **DataSet API** : elaborazione in batch, ovvero trasformazioni su set di dati.
- **Table API** : un linguaggio di espressione simile a SQL (come i dataframes in Spark) che può essere incorporato in applicazioni batch e di streaming.

Costruzioni

Al livello più elementare, Flink è composto da sorgenti, trasformazioni e sink (s).



Al livello più elementare, un programma Flink è composto da:

- **Fonte dei dati** : dati ricevuti che Flink elabora
- **Trasformazioni** : la fase di elaborazione, quando Flink modifica i dati in entrata
- **Data sink** : Dove Flink invia i dati dopo l'elaborazione

Le fonti e i sink possono essere file locali / HDFS, database, code di messaggi, ecc. Sono già disponibili molti connettori di terze parti, oppure è possibile crearne facilmente.

Configurazione locale del runtime

0. assicurati di avere java 6 o successivo e che sia `JAVA_HOME` la variabile di ambiente `JAVA_HOME` .

1. scarica l'ultimo binario di flink qui :

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

   Se non hai intenzione di lavorare con Hadoop, scegli la versione di hadoop 1. Inoltre, si noti la versione scala scaricata, in modo da poter aggiungere le corrette dipendenze Maven nei programmi.

2. inizio flink:

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

Flink è già configurato per essere eseguito localmente. Per garantire che il flink sia in esecuzione, è possibile ispezionare i registri in `flink/log/` o aprire il lavoro di flinkinterfacciaManager in esecuzione su `http://localhost:8081` .

3. fermare il flink:

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

Configurazione dell'ambiente Flink

Per eseguire un programma di flink dal tuo IDE (possiamo usare Eclipse o Intellij IDEA (preffered)), hai bisogno di due dipendenze: `flink-java` / `flink-scala` e `flink-clients` (a febbraio 2016). Questi JARS possono essere aggiunti usando Maven e SBT (se stai 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>
```

- Nome **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 : il `2.11` nel nome del manufatto è la *versione scala* , assicurati di corrispondere a quello che hai sul tuo sistema.

WordCount - API Table

Questo esempio è uguale a *WordCount* , ma utilizza l'API Table. Vedi *WordCount* per i dettagli sull'esecuzione e sui risultati.

Maven

Per utilizzare l'API Table, aggiungi `flink-table` come dipendenza da maven:

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

Il codice

```
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 : per una versione che utilizza Java <8, sostituire lambda con una classe anonima:

```
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

Aggiungi le dipendenze `flink-java` e `flink-client` (come spiegato nell'esempio di *configurazione dell'ambiente JVM*).

Il codice

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

Se si utilizza Java 8, è possibile sostituire `.flatmap(new LineSplitter())` con un'espressione 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 );
```

Esecuzione

Dall'IDE : basta premere " *Esegui"* nel tuo IDE. Flink creerà un ambiente all'interno della JVM.

Dalla riga di comando del flink : per eseguire il programma utilizzando un ambiente locale autonomo, effettuare le seguenti operazioni:

1. assicurarsi che il flink sia in esecuzione (`flink/bin/start-local.sh`);
2. creare un file jar (`maven package`);
3. utilizzare lo strumento da riga di comando di `flink` (nella cartella `bin` della propria installazione di flink) per avviare il programma:

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

   L'opzione `-c` ti consente di specificare la classe da eseguire. Non è necessario se il jar è eseguibile / definisce una classe principale.

Risultato

```
(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 - Streaming API

Questo esempio è uguale a *WordCount* , ma utilizza l'API Table. Vedi *WordCount* per i dettagli sull'esecuzione e sui risultati.

Maven

Per utilizzare l'API Streaming, aggiungi `flink-streaming` come dipendenza Maven:

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

Il codice

```
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();
    }
}
```

Leggi Iniziare con apache-flink online: https://riptutorial.com/it/apache-flink/topic/5798/iniziare-con-apache-flink

Capitolo 2: checkpointing

introduzione

(testato su Flink 1.2 e sotto)

Ogni funzione, sorgente o operatore in Flink può essere di stato. I checkpoint consentono a Flink di recuperare stato e posizioni negli stream per dare all'applicazione la stessa semantica di un'esecuzione priva di errori. È il mecanismo dietro le garanzie di *tolleranza agli errori* e l'elaborazione *esattamente una volta* .

Leggi questo articolo per capire gli interni.

Osservazioni

I checkpoint sono utili solo quando si verifica un errore nel cluster, ad esempio quando un task manager non riesce. Non persistono dopo che il lavoro stesso ha fallito o è stato annullato.

Per essere in grado di riprendere un lavoro stateful dopo il fallimento / cancellazione, dare un'occhiata a **punti di salvataggio** o **posti di blocco esternalizzate (Flink 1.2+).**

Examples

Configurazione e configurazione

La configurazione del checkpoint avviene in due passaggi. Per prima cosa, devi scegliere un *back-end* . Quindi, è possibile specificare l'intervallo e la modalità dei punti di controllo in base alle singole applicazioni.

---

backend

Backend disponibili

Dove i checkpoint sono memorizzati dipende dal back-end configurato:

- `MemoryStateBackend` : in memoria, backup nella memoria di JobManager / ZooKeeper. Dovrebbe essere utilizzato solo per lo stato minimo (predefinito fino a un massimo di 5 MB, per esempio per l'archiviazione degli offset Kafka) o per il test e il debugging locale.
- `FsStateBackend` : lo stato viene mantenuto in memoria sui TaskManager e le istantanee di stato (ad esempio i checkpoint) sono archiviate in un file system (HDFS, DS3, file system locale, ...). Questa configurazione è consigliata per grandi stati o finestre lunghe e per configurazioni ad alta disponibilità.
- `RocksDBStateBackend` : detiene i dati in-flight in un database RocksDB che è (per impostazione

predefinita) memorizzato nelle directory dei dati TaskManager. Al momento del checkpoint, l'intero database RocksDB viene scritto su un file (come sopra). Rispetto a FsStateBackend, consente gli stati più grandi (limitati solo dallo spazio su disco rispetto alla dimensione della memoria del task manager), ma il throughput sarà inferiore (i dati non sono sempre in memoria, devono essere caricati dal disco).

Si noti che qualunque sia il backend, i metadati (numero di checkpoint, localizzazione, ecc.) Sono sempre memorizzati nella memoria del job manager e i punti di controllo **non persistono dopo la terminazione / cancellazione dell'applicazione** .

Specifica del back-end

Specifica il backend nel metodo `main` del tuo programma usando:

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

Oppure imposta il backend predefinito in `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
```

Abilitazione dei checkpoint

Ogni applicazione deve abilitare esplicitamente i checkpoint:

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

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

È possibile specificare facoltativamente una *modalità checkpoint* . In caso contrario, si imposta automaticamente *esattamente una volta* :

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

La modalità checkpoint definisce quale coerenza garantisce il sistema in presenza di guasti. Quando viene attivato il checkpoint, i flussi di dati vengono riprodotti in modo tale da ripetere le parti perse del processo. Con `EXACTLY_ONCE` , il sistema disegna i checkpoint in modo tale che un

recupero si comporti come se gli operatori / funzioni vedessero ogni record "esattamente una volta". Con `AT_LEAST_ONCE` , i checkpoint vengono disegnati in modo più semplice, che in genere riscontra alcuni duplicati al momento del recupero.

Testare i checkpoint

Il codice

Ecco una semplice applicazione di flink che utilizza un mappatore stateful con uno stato gestito `Integer` . Puoi giocare con le variabili `checkpointEnable` , `checkpointInterval` e `checkpointMode` per vedere il loro effetto:

```
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);
        }
    }
}
```

Esecuzione dell'esempio e simulazione del fallimento

Per poter controllare i checkpoint, è necessario avviare un `cluster` . Il modo più semplice è utilizzare lo script `start-cluster.sh` nella directory `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.
```

Ora, impacchetta la tua app e invialo a flink:

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

Crea alcuni dati:

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

L'output dovrebbe essere disponibile in `flink/logs/flink-<user>-jobmanager-0-<host>.out` . Per esempio:

```
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)
```

Per testare i checkpoint, basta semplicemente uccidere il task manager (questo emulerà un errore), produrre alcuni dati e avviarne uno nuovo:

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

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

Nota: all'avvio di un nuovo taskmanager, verrà utilizzato un altro file di log, ovvero `flink/logs/flink-<user>-jobmanager-1-<host>.out` (notare l'incremento intero).

Cosa aspettarsi

- *checkpoint disabilitati* : se si producono dati durante l'errore, saranno definitivamente persi. Ma sorprendentemente, i contatori avranno ragione!
- *checkpoint abilitati* : nessuna perdita di dati (e contatori corretti).
- *punti di controllo con almeno una volta modalità* : è possibile visualizzare i duplicati, soprattutto se si imposta un intervallo di checkpoint su un numero elevato e si uccide il task manager più volte

Leggi checkpointing online: https://riptutorial.com/it/apache-flink/topic/9465/checkpointing

Capitolo 3: Come definire uno schema di serializzazione personalizzato (de)

introduzione

Gli schemi sono usati da alcuni connettori (Kafka, RabbitMQ) per trasformare i messaggi in oggetti Java e viceversa.

Examples

Esempio di schema personalizzato

Per utilizzare uno schema personalizzato, tutto ciò che devi fare è implementare un'interfaccia `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 classe `MyMessage` è definita come segue:

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

Leggi Come definire uno schema di serializzazione personalizzato (de) online: https://riptutorial.com/it/apache-flink/topic/9004/come-definire-uno-schema-di-serializzazione-personalizzato--de-

Capitolo 4: Consuma i dati da Kafka

Examples

Esempio di KafkaConsumer

`FlinkKafkaConsumer` consente di utilizzare i dati di uno o più argomenti di kafka.

versioni

Il consumatore da usare dipende dalla tua distribuzione di kafka.

- `FlinkKafkaConsumer08` : utilizza la vecchia API `SimpleConsumer` di Kafka. Gli offset vengono gestiti da Flink e assegnati a Zookeeper.
- `FlinkKafkaConsumer09` : utilizza la nuova API Consumer di Kafka, che gestisce gli offset e il ribilanciamento automatico.
- `FlinkKafkaProducer010` : questo connettore supporta i messaggi Kafka con timestamp sia per produrre che per consumare (utile per le operazioni con le finestre).

uso

I binari non fanno parte del core del flink, quindi è necessario importarli:

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

Il costruttore prende tre argomenti:

- uno o più argomenti da leggere
- uno schema di deserializzazione che dice a Flink come interpretare / decodificare i messaggi
- Proprietà di configurazione del consumatore kafka. Sono gli stessi di un consumatore di kafka "normale". Il minimo richiesto sono:
  - `bootstrap.servers` : un elenco separato da virgole di broker Kafka nella forma ip: port. Per la versione 8, invece, utilizzare `zookeeper.connect` (elenco dei server zookeeper)
  - `group.id` : l'id del gruppo di consumatori (vedi la documentazione di kafka per maggiori dettagli)

In 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));
```

In 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 lo sviluppo, è possibile utilizzare le proprietà `enable.auto.commit=false` e `auto.offset.reset=earliest` per riconsiderare gli stessi dati ogni volta che si avvia il pogramma.

Tolleranza ai guasti

Come spiegato nei documenti ,

> Con il checkpoint abilitato di Flink, il Flink Kafka Consumer consumerà record da un argomento e periodicamente controllerà tutti i suoi offset di Kafka, insieme allo stato di altre operazioni, in modo coerente. In caso di fallimento di un lavoro, Flink ripristinerà il programma di streaming nello stato dell'ultimo checkpoint e riutilizzerà i record di Kafka, a partire dagli offset che sono stati memorizzati nel checkpoint.
>
> L'intervallo dei punti di controllo del disegno definisce quindi quanto il programma potrebbe dover tornare al massimo, in caso di errore.

Per utilizzare i consumatori con tolleranza agli errori di Kafka, è necessario abilitare il checkpoint nell'ambiente di esecuzione utilizzando il metodo `enableCheckpointing` :

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

Schemi di deserializzazione incorporati

SimpleStringSchema : `SimpleStringSchema` deserializza il messaggio come una stringa. Nel caso in cui i tuoi messaggi abbiano chiavi, quest'ultimo verrà ignorato.

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

JSONDeserializationSchema

`JSONDeserializationSchema` deserializza i messaggi json-formattati usando *jackson* e restituisce un flusso di oggetti `com.fasterxml.jackson.databind.node.ObjectNode` . È quindi possibile utilizzare il metodo `.get("property")` per accedere ai campi. Ancora una volta, le chiavi vengono ignorate.

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

JSONKeyValueDeserializationSchema

`JSONKeyValueDeserializationSchema` è molto simile a quello precedente, ma tratta i messaggi con valori E di chiavi con codifica json.

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

Il `ObjectNode` restituito contiene i seguenti campi:

- `key` : tutti i campi presenti nella chiave
- `value` : tutti i campi del messaggio
- (facoltativo) `metadata` : espone l' `offset` , la `partition` e l' `topic` del messaggio (passare `true` al costruttore per recuperare anche i metadati).

Per esempio:

```
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
```

Verrà decodificato come:

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

Partizioni di Kafka e parallelismo di Flink

In kafka, ad ogni utente dello stesso gruppo di consumatori viene assegnata una o più partizioni. Si noti che non è possibile per due consumatori consumare dalla stessa partizione. Il numero di consumatori di flink dipende dal parallelismo del flink (valore predefinito 1).

Ci sono tre casi possibili:

1. **kafka partitions == flink parallelism** : questo caso è l'ideale, dato che ogni utente si prende cura di una partizione. Se i tuoi messaggi sono bilanciati tra le partizioni, il lavoro sarà distribuito uniformemente tra gli operatori di flink;

2. **kafka partitions <parallelismo di flink** : alcune istanze di flink non riceveranno alcun

messaggio. Per evitare ciò, è necessario richiamare il `rebalance` sul flusso di input *prima di qualsiasi operazione* , che provoca la ri-partizione dei dati:

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

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

3. **kafka partitions> flink parallelism** : in questo caso, alcune istanze gestiranno più partizioni. Ancora una volta, è possibile utilizzare il `rebalance` per diffondere i messaggi in modo uniforme tra i lavoratori.

Leggi Consuma i dati da Kafka online: https://riptutorial.com/it/apache-flink/topic/9003/consuma-i-dati-da-kafka

Capitolo 5: Punti di salvataggio e punti di controllo esterni

introduzione

I punti di salvataggio sono checkpoint *"grassi", memorizzati esternamente* che ci consentono di riprendere un programma di fluttuazione stateful dopo un errore permanente, una cancellazione o un aggiornamento del codice. Prima di Flink 1.2 e dell'introduzione di *checkpoint esterni* , i *punti di* salvataggio dovevano essere attivati in modo esplicito.

Examples

Punti di salvataggio: requisiti e note preliminari

Un punto di salvataggio memorizza due cose: (a) le posizioni di tutte le origini dati, (b) gli stati degli operatori. I punti di salvataggio sono utili in molte circostanze:

- lievi aggiornamenti del codice dell'applicazione
- Aggiornamento Flink
- cambiamenti nel parallelismo
- ...

A partire dalla **versione 1.3** (valido anche per la versione precedente):

- il punto di controllo **deve essere abilitato** affinché i punti di salvataggio siano possibili. Se si dimentica di abilitare esplicitamente il checkpoint usando:

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

  otterrete:

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

- quando si usano le operazioni con le finestre, è fondamentale utilizzare il tempo di evento (rispetto all'ingestione o al tempo di elaborazione) per ottenere risultati corretti;

- per poter aggiornare un programma e riutilizzare i punti di salvataggio, è **necessario impostare uid manuale** . Questo perché, per impostazione predefinita, Flink cambia l'UID dell'operatore dopo ogni modifica nel loro codice;

- Gli operatori concatenati sono identificati dall'ID della prima attività. Non è possibile assegnare manualmente un ID a un'attività intermedia concatenata, ad esempio nella catena [a -> b -> c] solo a può essere assegnato manualmente il proprio ID, ma non b o c. Per

ovviare a questo, è possibile definire manualmente le catene di attività. Se ti affidi all'assegnazione automatica dell'ID, un cambiamento nel comportamento di concatenazione cambierà anche gli ID (vedi punto sopra).

Maggiori informazioni sono disponibili nelle FAQ .

punti di salvataggio

Configurazione

La configurazione è nel file `flink/conf/flink-conf.yaml` (in Mac OSX tramite homebrew, è `/usr/local/Cellar/apache-flink/1.1.3/libexec/conf/flink-conf.yaml`).

Flink <1.2 : la configurazione è molto simile alla configurazione dei checkpoint (argomento disponibile). L'unica differenza è che non ha senso definire un backend di salvataggio in memoria, poiché abbiamo bisogno che i punti di salvataggio continuino dopo l'arresto di 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 : se non si specifica un back-end, il backend predefinito è *jobmanager* , il che significa che i punti di *salvataggio* spariranno una volta che il cluster è stato arrestato. Questo è utile solo per il debug.

Flink 1.2+ : come spiegato in questo biglietto jira , lasciare che un punto di salvataggio sia salvato nella memoria del job manager ha poco senso. Dal momento che Flink 1.2, i punti di salvataggio sono necessariamente memorizzati in file. La configurazione di cui sopra è stata sostituita da:

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

uso

Ottenere l'ID del lavoro

Per attivare un punto di salvataggio, tutto ciò che serve è l'ID del lavoro dell'applicazione. L'ID del lavoro viene stampato nella riga di comando quando si avvia il lavoro o può essere recuperato in un secondo momento utilizzando l' `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.
```

Attivare un punto di salvataggio

Per attivare un punto di salvataggio, utilizzare il punto di `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.
```

Nota che puoi anche fornire una directory di destinazione come secondo argomento, sovrascriverà quella predefinita definita in `flink/bin/flink-conf.yaml` .

In Flink 1.2+, è anche possibile cancellare un lavoro E fare un punto di salvataggio allo stesso tempo, usando l'opzione `-s` :

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

Nota : è possibile spostare un punto di salvataggio, ma non rinominarlo!

Riprendendo da un punto di salvataggio

Per riprendere da un punto di salvataggio specifico, utilizzare l'opzione `-s [savepoint-dir]` del comando di `flink run` del `flink run` :

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

Specifica UID dell'operatore

Per poter riprendere da un punto di salvataggio dopo una modifica del codice, è necessario assicurarsi che il nuovo codice utilizzi lo stesso UID per l'operatore. Per assegnare manualmente un UID, chiamare la funzione `.uid(<name>)` subito dopo l'operatore:

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

Punti di controllo esterni (Flink 1.2+)

Prima di 1.2, l'unico modo per mantenere lo stato / mantenere un checkpoint dopo una interruzione / cancellazione / persistente di un processo avveniva tramite un punto di salvataggio, che viene attivato manualmente. La versione 1.2 ha introdotto punti di controllo persistenti.

I punti di controllo persistenti si comportano molto come i normali punti di controllo periodici ad eccezione delle seguenti differenze:

1. Persistono i loro metadati in una memoria persistente (come i punti di salvataggio).
2. Non vengono scartati quando il lavoro proprietario fallisce in modo permanente. Inoltre, possono essere configurati per non essere scartati quando il lavoro viene annullato.

È quindi molto simile ai punti di salvataggio; infatti, i punti di salvataggio sono solo checkpoint esterni con un po 'più di informazioni.

Nota importante : al momento, il coordinatore del checkpoint di Flink conserva solo l'ultimo checkpoint completato con successo. Ciò significa che ogni volta che viene completato un nuovo checkpoint, l'ultimo checkpoint completato verrà scartato. Questo vale anche per i checkpoint esterni.

Configurazione

Dove i metadati relativi ai punti di controllo [esternalizzati] sono archiviati sono configurati in `flink-conf.yaml` (e non possono essere ignorati tramite codice):

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

Si noti che questa directory *contiene solo i metadati del checkpoint* richiesti per ripristinare il checkpoint. I file di checkpoint effettivi sono ancora memorizzati nella loro directory configurata (cioè proprietà `state.bachend.fs.checkpointdir`).

uso

È necessario abilitare esplicitamente i checkpoint esterni nel codice utilizzando il metodo `getCheckpointConfig()` dell'ambiente di streaming:

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

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

Le modalità disponibili di `ExternalizedCheckpointCleanup` sono:

- `RETAIN_ON_CANCELLATION` : l'ultimo checkpoint e i relativi metadati vengono conservati all'annullamento del lavoro; è tua responsabilità ripulire in seguito.

- `DELETE_ON_CANCELLATION` : l'ultimo checkpoint viene cancellato all'annullamento, il che significa che è disponibile solo se l'applicazione fallisce.

Per riprendere da un checkpoint esterno, utilizzare la sintassi savepoint. Per esempio:

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

Leggi Punti di salvataggio e punti di controllo esterni online: https://riptutorial.com/it/apache-flink/topic/9466/punti-di-salvataggio-e-punti-di-controllo-esterni

Capitolo 6: registrazione

introduzione

Questo argomento mostra come utilizzare e configurare la registrazione (log4j) nelle applicazioni Flink.

Examples

Usando un logger nel tuo codice

Aggiungi la dipendenza `slf4j` al tuo `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>
```

Creare un oggetto logger da utilizzare nella classe:

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

Nelle classi che devono essere serializzate, come le sottoclassi di `RichMapFunction` , non dimenticare di registrare `LOGGER` come `transient` :

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

Nel tuo codice, usa `LOGGER` come al solito. Usa segnaposti (`{}`) per formattare oggetti e tali:

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

Registrazione della configurazione

Modalità locale

In modalità locale, ad esempio quando si esegue l'applicazione da un IDE, è possibile configurare `log4j` come al solito, ovvero rendendo disponibile `log4j.properties` nel classpath. Un modo semplice in Maven è creare `log4j.properties` nella `log4j.properties src/main/resources` . Ecco un esempio:

```
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
```

Modalità autonoma

> In modalità standalone, la configurazione effettiva utilizzata non è quella del file `jar` . Questo perché Flink ha i propri file di configurazione, che hanno la precedenza sul proprio.

File predefiniti : Flink viene fornito con i seguenti file di proprietà predefiniti:

- `log4j-cli.properties` : usato dal client della riga di comando di Flink (es. `flink run`) (non codice eseguito sul cluster)
- `log4j-yarn-session.properties` : usato dal client della riga di comando Flink all'avvio di una sessione `yarn-session.sh` (`yarn-session.sh`)
- `log4j.properties` : `log4j.properties` JobManager / Taskmanager (sia standalone che YARN)

Si noti che `${log.file}` predefinito per `flink/log` . Può essere sovrascritto in `flink-conf.yaml` , impostando `env.log.dir` ,

> `env.log.dir` definisce la directory in cui vengono salvati i registri Flink. Deve essere un percorso assoluto.

Posizione del registro : i registri sono *locali* , ovvero sono prodotti nella / e macchina / e che

eseguono il / i / i / i Taskmanager / i di JobManager.

Yarn : quando esegui Flink su Yarn, devi fare affidamento sulle funzionalità di logging di Hadoop YARN. La funzione più utile è l' aggregazione dei registri YARN . Per abilitarlo, imposta la proprietà `yarn.log-aggregation-enable` su `true` nel `yarn-site.xml file` . Una volta abilitato, è possibile recuperare tutti i file di registro di una sessione YARN (non riuscita) utilizzando:

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

Purtroppo, i registri sono disponibili *solo dopo l'* interruzione di *una sessione* , ad esempio dopo un errore.

Utilizzo di diverse configurazioni per ciascuna applicazione

Nel caso in cui abbiate bisogno di impostazioni diverse per le vostre varie applicazioni, non vi è (come in Flink 1.2) un modo semplice per farlo.

Se si utilizza la modalità di flink a *un filo-gruppo-per-lavoro* (cioè si lanciano i propri script con: `flink run -m yarn-cluster` ...), ecco una soluzione:

1. crea una directory `conf` vicino al tuo progetto
2. creare collegamenti simbolici per tutti i file in `flink/conf` :

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

3. sostituire il symlink `log4j.properties` (o qualsiasi altro file che si desidera modificare) con la propria configurazione
4. prima di iniziare il tuo lavoro, corri

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

A seconda della versione di flink, potrebbe essere necessario modificare il `flink/bin/config.sh` del file `flink/bin/config.sh` . Se la tua corsa attraversa questa linea:

```
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
```

Soluzione alternativa Flink-on-Yarn: ottieni i log in tempo reale con rsyslog

Per impostazione predefinita, il filato non esegue aggregati dei registri prima che un'applicazione finisca, il che può essere problematico con i lavori di streaming che non terminano nemmeno.

Una soluzione alternativa è usare `rsyslog` , che è disponibile sulla maggior parte delle macchine Linux.

Innanzitutto, consenti le richieste in entrata di udp decommentando le seguenti righe in `/etc/rsyslog.conf` :

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

Modifica il tuo `log4j.properties` (vedi gli altri esempi in questa pagina) per utilizzare `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
```

Il layout è importante, perché rsyslog considera una nuova riga come una nuova voce di registro. Sopra, i newline (in stacktraces per esempio) verranno saltati. Se vuoi davvero che i registri multilinea / a schede funzionino "normalmente", modifica `rsyslog.conf` e aggiungi:

```
$EscapeControlCharactersOnReceive off
```

L'uso di `myTag:` all'inizio della `conversionPattern` è utile se vuoi reindirizzare tutti i log in un file specifico. Per fare ciò, modifica `rsyslog.conf` e aggiungi la seguente regola:

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

Leggi registrazione online: https://riptutorial.com/it/apache-flink/topic/9713/registrazione

Capitolo 7: Tabella API

Examples

Dipendenze Maven

Per utilizzare l'API Table, aggiungi `flink-table` come dipendenza maven (oltre a `flink-clients` e `flink-core`):

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

Assicurati che la versione scala (qui 2.11) sia compatibile con il tuo sistema.

Semplice aggregazione da un CSV

Dato il file 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
```

Vogliamo contare le persone per paese e per nazione + genere:

```
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();
    }
}
```

I risultati sono:

```
Germany,2
United States,18

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

Unisciti alle tabelle di esempio

Oltre a `peoples.csv` (vedi *aggregazione semplice da un CSV*) abbiamo altri due CSV che rappresentano prodotti e vendite.

`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, nome, prezzo):

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

Vogliamo ottenere il nome e il prodotto per ogni vendita di oltre 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
```

Nota che è importante usare nomi diversi per ogni colonna, altrimenti il flink si lamenterà di "nomi ambigui in join".

Risultato:

```
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
```

Utilizzando lavandini esterni

Una tabella può essere scritta in un TableSink, che è un'interfaccia generica per supportare diversi formati e file system. Una tabella batch può essere scritta solo su un `BatchTableSink` , mentre una tabella di streaming richiede uno `StreamTableSink` .

Attualmente, Flink offre solo l'interfaccia `CsvTableSink` .

uso

Negli esempi sopra, sostituire:

```
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` è una cartella, perché flink esegue operazioni parallele. Quindi, se si dispone di 4 processori, è probabile che nella cartella dei risultati siano presenti 4 file.

Inoltre, si noti che chiamiamo esplicitamente `env.execute()` : questo è necessario per avviare un lavoro di flink, ma negli esempi precedenti `print()` lo ha fatto per noi.

Leggi Tabella API online: https://riptutorial.com/it/apache-flink/topic/8966/tabella-api

Titoli di coda

S. No	Capitoli	Contributors
1	Iniziare con apache-flink	Community, Derlin, vdep
2	checkpointing	Derlin
3	Come definire uno schema di serializzazione personalizzato (de)	Derlin
4	Consuma i dati da Kafka	alpinegizmo, Derlin
5	Punti di salvataggio e punti di controllo esterni	Derlin
6	registrazione	Derlin
7	Tabella API	Derlin