Kostenloses eBook

LERNEN

apache-flink

Free unaffiliated eBook created from
Stack Overflow contributors.

#apache-flink

Inhaltsverzeichnis

Über

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

Kapitel 1: Erste Schritte mit Apache-Flink

Bemerkungen

In diesem Abschnitt erhalten Sie einen Überblick darüber, was Apache-Flink ist und warum ein Entwickler es verwenden möchte.

Es sollte auch alle großen Themen innerhalb von Apache-Flink erwähnen und auf die verwandten Themen verweisen. Da die Dokumentation für Apache-Flink neu ist, müssen Sie möglicherweise erste Versionen dieser verwandten Themen erstellen.

Examples

Übersicht und Anforderungen

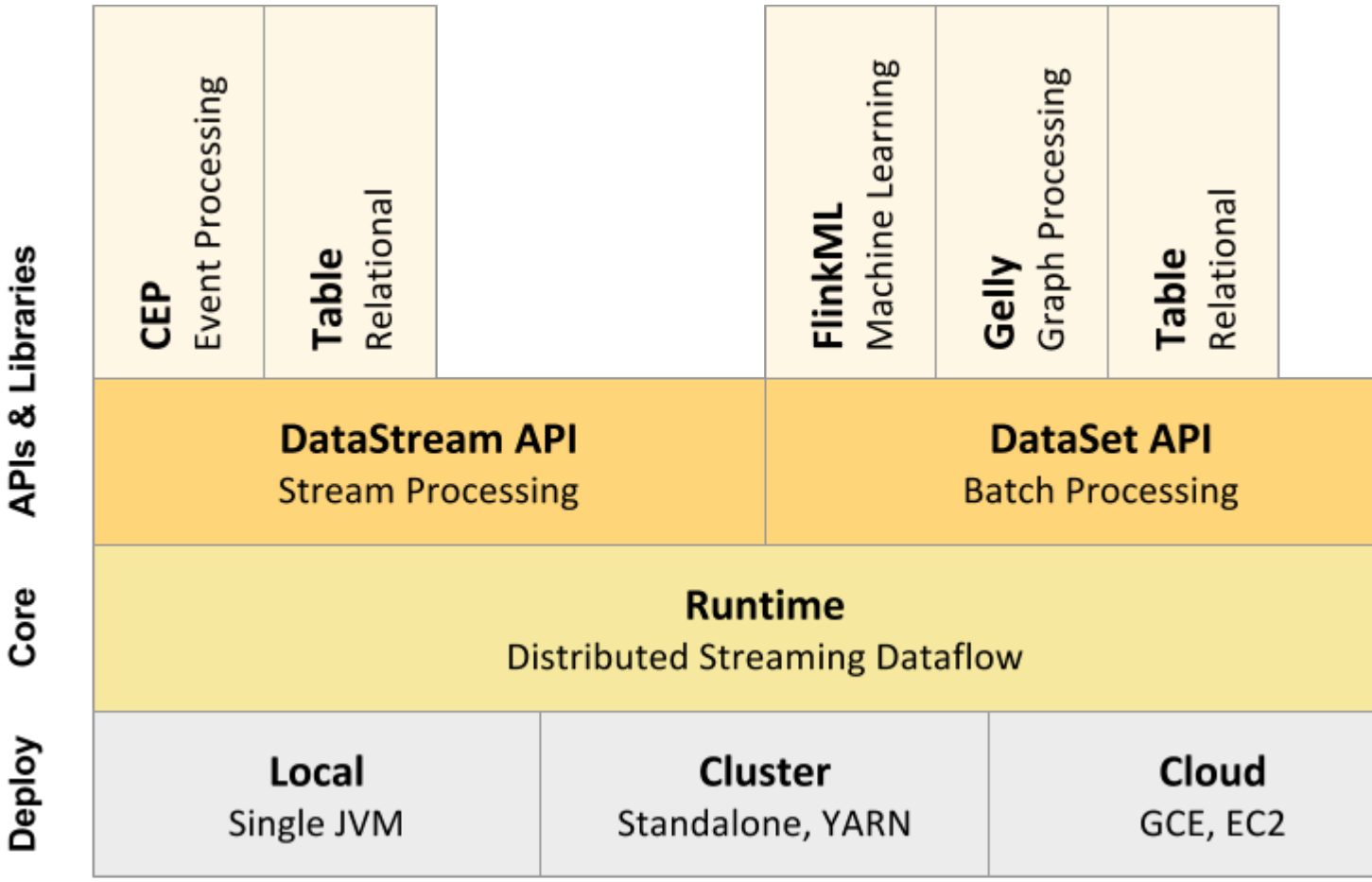
Was ist Flink?

Wie Apache Hadoop und Apache Spark ist Apache Flink ein von der Community gesteuertes Open Source-Framework für verteilte Big Data Analytics. Flink wurde in Java geschrieben und verfügt über APIs für Scala, Java und Python, wodurch Batch- und Echtzeit-Streaming-Analysen möglich sind.

Bedarf

- eine UNIX-ähnliche Umgebung wie Linux, Mac OS X oder Cygwin;
- Java 6.X oder höher;
- [optional] Maven 3.0.4 oder höher.

Stapel



Ausführungsumgebungen

> Apache Flink ist ein Datenverarbeitungssystem und **eine Alternative zur MapReduce-Komponente von Hadoop** . Es wird mit einer *eigenen Laufzeit geliefert,* anstatt auf MapReduce aufzubauen. Somit kann es völlig unabhängig vom Hadoop-Ökosystem arbeiten.

Die `ExecutionEnvironment` ist der Kontext, in dem ein Programm ausgeführt wird. Je nach Ihren Anforderungen können Sie verschiedene Umgebungen verwenden.

1. *JVM-Umgebung* : Flink kann auf einer einzelnen Java Virtual Machine ausgeführt werden, sodass Benutzer Flink-Programme direkt von ihrer IDE aus testen und debuggen können. Wenn Sie diese Umgebung verwenden, benötigen Sie nur die richtigen Abhängigkeiten von Maven.

2. *Lokale Umgebung* : Um ein Programm auf einer laufenden Flink-Instanz (nicht in Ihrer IDE) ausführen zu können, müssen Sie Flink auf Ihrem Computer installieren. Siehe lokales Setup .

3. *Cluster-Umgebung* : Um Flink vollständig verteilt auszuführen, ist ein Standalone- oder ein Garn-Cluster erforderlich. Weitere Informationen finden Sie auf der Cluster-Setup-Seite oder in dieser Diashow . mportant__: Die *Version* `2.11` im Artefaktnamen ist die *Scala-Version* . Vergewissern Sie sich, dass Sie mit der *Version* auf Ihrem System übereinstimmen.

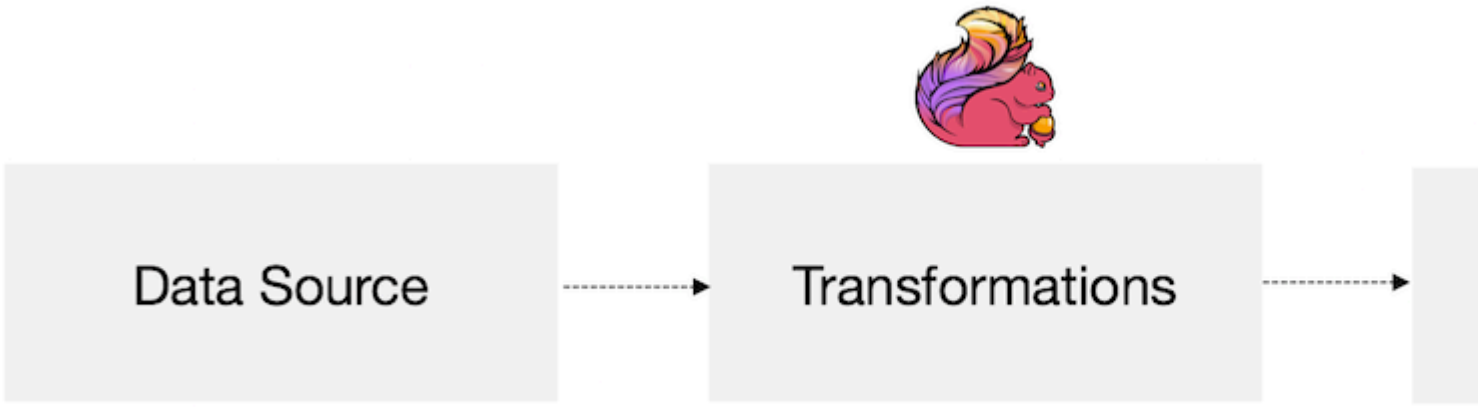
APIs

Flink kann entweder für die Stream- oder Stapelverarbeitung verwendet werden. Sie bieten drei APIs an:

- **DataStream-API** : Stream-Verarbeitung, dh Transformationen (Filter, Zeitfenster, Aggregationen) für unbegrenzte Datenflüsse.
- **DataSet-API** : Stapelverarbeitung, dh Umwandlungen von Datensätzen.
- **Tabellen-API** : Eine SQL-ähnliche Ausdruckssprache (wie Dataframes in Spark), die in Batch- und Streaming-Anwendungen eingebettet werden kann.

Bausteine

Grundsätzlich besteht Flink aus Quelle (n), Transformation (en) und Senke (Senken).



Grundsätzlich besteht ein Flink-Programm aus:

- **Datenquelle** : Eingehende Daten, die Flink verarbeitet
- **Transformationen** : Der Verarbeitungsschritt, wenn Flink ankommende Daten ändert
- **Datensenke:** Wo Flink Daten nach der Verarbeitung sendet

Quellen und Senken können lokale / HDFS-Dateien, Datenbanken, Nachrichtenwarteschlangen usw. sein. Es sind bereits viele Connectoren von Drittanbietern verfügbar, oder Sie können leicht eigene erstellen.

Lokale Laufzeiteinrichtung

0. `JAVA_HOME` Sie sicher, dass Sie über Java 6 oder höher verfügen und die Umgebungsvariable `JAVA_HOME` gesetzt ist.

1. Laden Sie die neueste Flink-Binärdatei hier herunter:

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

Wenn Sie nicht mit Hadoop arbeiten möchten, wählen Sie die hadoop 1-Version. Notieren Sie sich auch die heruntergeladene Scala-Version, damit Sie die richtigen Abhängigkeiten in Ihren Programmen hinzufügen können.

2. Startflink:

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

Flink ist bereits für die lokale Ausführung konfiguriert. Um sicherzustellen, dass flink ausgeführt wird, können Sie die Protokolle in `flink/log/` überprüfen oder die Schnittstelle des flink-jobManagers unter `http://localhost:8081` öffnen.

3. Stop Flink:

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

Flink-Umgebung einrichten

Um ein Flink-Programm von Ihrer IDE aus auszuführen (wir können entweder Eclipse oder Intellij IDEA (bevorzugt) verwenden), benötigen Sie zwei Abhängigkeiten: `flink-java` / `flink-scala` und `flink-clients` (Stand Februar 2016). Diese JARS können mit Maven und SBT hinzugefügt werden (wenn Sie Scala verwenden).

- **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>
```

- **SBT-** Name: = ""

```
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"
)
```

Wichtig : Die `2.11` im Artefaktnamen ist die *Scala-Version* . Vergewissern Sie sich, dass Sie mit der auf Ihrem System vorhandenen übereinstimmen.

WordCount - Tabellen-API

Dieses Beispiel ist identisch mit *WordCount* , verwendet jedoch die Tabellen-API. *Weitere* Informationen zu Ausführung und Ergebnissen finden Sie unter *WordCount* .

Maven

Um die Tabellen-API zu verwenden, fügen Sie `flink-table` als Maven-Abhängigkeit hinzu:

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

Der 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();
    }
}
```

Hinweis : Ersetzen Sie bei einer Version mit Java <8 das Lambda durch eine anonyme Klasse:

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

Wortzahl

Maven

Fügen Sie die Abhängigkeiten `flink-java` und `flink-client` (wie im *Setup*- Beispiel für die *JVM-Umgebung* erläutert).

Der 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 ) );
            }
        }
    }
}
```

Wenn Sie Java 8 verwenden, können Sie `.flatmap(new LineSplitter())` durch einen Lambda-Ausdruck ersetzen:

```
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 );
```

Ausführung

Von der IDE : **Klicken Sie** einfach auf *Run* in Ihrer IDE. Flink erstellt eine Umgebung in der JVM.

Von der Flink-Befehlszeile aus : Führen Sie die folgenden **Schritte aus** , um das Programm in einer lokalen Umgebung auszuführen:

1. Stellen Sie sicher, dass der Flink läuft (`flink/bin/start-local.sh`);
2. Erstellen Sie eine JAR-Datei (`maven package`).
3. Verwenden Sie das `flink` (im Ordner `bin` Ihrer Flink-Installation), um das Programm zu starten:

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

   Mit der Option `-c` können Sie die Klasse angeben, die ausgeführt werden soll. Es ist nicht erforderlich, wenn die JAR-Datei ausführbar ist / eine Hauptklasse definiert.

Ergebnis

```
(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

Dieses Beispiel ist identisch mit *WordCount* , verwendet jedoch die Tabellen-API. *Weitere* Informationen zu Ausführung und Ergebnissen finden Sie unter *WordCount* .

Maven

Um die Streaming-API zu verwenden, fügen Sie `flink-streaming` als Maven-Abhängigkeit hinzu:

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

Der 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();
    }
}
```

Erste Schritte mit Apache-Flink online lesen: https://riptutorial.com/de/apache-flink/topic/5798/erste-schritte-mit-apache-flink

Kapitel 2: Daten von Kafka verbrauchen

Examples

KafkaConsumer-Beispiel

`FlinkKafkaConsumer` Sie Daten aus einem oder mehreren Kafka-Themen verwenden.

Versionen

Der zu verwendende Verbraucher hängt von Ihrer Kafka-Distribution ab.

- `FlinkKafkaConsumer08` : verwendet die alte `SimpleConsumer` API von Kafka. Offsets werden von Flink abgewickelt und dem Zoowächter übergeben.
- `FlinkKafkaConsumer09` : Verwendet die neue Consumer-API von Kafka, die Offsets und Ausgleichszahlungen automatisch übernimmt.
- `FlinkKafkaProducer010` : Dieser Connector unterstützt Kafka-Nachrichten mit Zeitstempeln zum Produzieren und Konsumieren (nützlich für Fensteroperationen).

Verwendungszweck

Die Binärdateien sind nicht Teil des Flink-Kerns, daher müssen Sie sie importieren:

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

Der Konstruktor akzeptiert drei Argumente:

- ein oder mehrere themen zum lesen
- ein Deserialisierungsschema, das Flink mitteilt, wie die Nachrichten interpretiert / dekodiert werden sollen
- kafka Consumer-Konfigurationseigenschaften. Das ist dasselbe wie ein "normaler" Kafka-Verbraucher. Die Mindestanforderungen sind:
  - `bootstrap.servers` : Eine durch Kommas getrennte Liste von Kafka-Brokern im Format ip: port. Verwenden `zookeeper.connect` für Version 8 stattdessen `zookeeper.connect` (Liste der Zoookeeper-Server)
  - `group.id` : Die ID der Verbrauchergruppe (weitere Informationen finden Sie in der Kafka-Dokumentation).

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))
```

Während der Entwicklung können Sie die kafka-Eigenschaften `enable.auto.commit=false` und `auto.offset.reset=earliest` um bei jedem Start Ihres Programms die gleichen Daten wiederherzustellen.

Fehlertoleranz

Wie in den Dokumenten erklärt ,

> Wenn Flinks Checkpointing aktiviert ist, konsumiert der Flink Kafka-Consumer Datensätze eines Themas und prüft regelmäßig alle seine Kafka-Offsets sowie den Status anderer Operationen auf konsistente Weise. Im Falle eines Auftragsfehlers stellt Flink das Streaming-Programm auf den Status des letzten Prüfpunkts zurück und verwendet die Datensätze von Kafka erneut, beginnend mit den im Prüfpunkt gespeicherten Offsets.
>
> Das Intervall für das Zeichnen von Kontrollpunkten definiert daher, um wie viel das Programm im Fehlerfall möglicherweise höchstens zurückgehen muss.

Um fehlertolerante Kafka-Verbraucher verwenden zu können, müssen Sie Checkpointing in der Ausführungsumgebung mithilfe der `enableCheckpointing` Methode `enableCheckpointing` :

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

Eingebaute Deserialisierungsschemata

SimpleStringSchema : `SimpleStringSchema` deserialisiert die Nachricht als String. Falls Ihre Nachrichten Schlüssel enthalten, werden diese ignoriert.

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

JSONDeserializationSchema

`JSONDeserializationSchema` deserialisiert json-formatierte Nachrichten mit *jackson* und gibt einen

Stream von `com.fasterxml.jackson.databind.node.ObjectNode` Objekten zurück. Sie können dann die `.get("property")` Methode `.get("property")` verwenden, um auf Felder zuzugreifen. Wieder werden die Schlüssel ignoriert.

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

JSONKeyValueDeserializationSchema

`JSONKeyValueDeserializationSchema` ist dem vorherigen sehr ähnlich, behandelt jedoch Nachrichten mit json-codierten Schlüsseln und Werten.

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

Der zurückgegebene `ObjectNode` enthält die folgenden Felder:

- `key` : Alle Felder im Schlüssel
- `value` : alle Nachrichtenfelder
- (optional) `metadata` : macht den `offset` , die `partition` und das `topic` der Nachricht verfügbar (übergeben Sie `true` an den Konstruktor, um auch Metadaten abzurufen).

Zum Beispiel:

```
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
```

Wird entschlüsselt als:

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

Kafka-Partitionen und Flink-Parallelismus

In kafka wird jedem Verbraucher derselben Verbrauchergruppe eine oder mehrere Partitionen zugewiesen. Beachten Sie, dass zwei Verbraucher nicht von derselben Partition verbrauchen können. Die Anzahl der Flink-Consumer hängt von der Flink-Parallelität ab (Standardeinstellung 1).

Es gibt drei mögliche Fälle:

1. **kafka Partitionen == Flink-Parallelität** : Dieser Fall ist ideal, da sich jeder Verbraucher um eine Partition kümmert. Wenn Ihre Nachrichten zwischen den Partitionen verteilt sind, wird die Arbeit gleichmäßig auf die Flink-Operatoren verteilt.

2. **kafka Partitionen <Parallelität von Flink** : Einige Flink-Instanzen empfangen keine Nachrichten. Um dies zu vermeiden, müssen Sie *vor jeder Operation eine* `rebalance` des Eingabestreams aufrufen, wodurch die Daten neu partitioniert werden:

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

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

3. **kafka-Partitionen> Parallelität mit Flink** : In diesem Fall werden einige Partitionen mit mehreren Partitionen behandelt. Wiederum können Sie die Funktion " `rebalance` , um Nachrichten gleichmäßig über die Mitarbeiter zu verteilen.

Daten von Kafka verbrauchen online lesen: https://riptutorial.com/de/apache-flink/topic/9003/daten-von-kafka-verbrauchen

Kapitel 3: Protokollierung

Einführung

In diesem Thema wird die Verwendung und Konfiguration der Protokollierung (log4j) in Flink-Anwendungen beschrieben.

Examples

Verwenden eines Loggers in Ihrem Code

Fügen Sie der `pom.xml` die Abhängigkeit von `slf4j` :

```
<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>
```

Erstellen Sie ein Logger-Objekt zur Verwendung in Ihrer Klasse:

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

`RichMapFunction` Sie nicht, in Klassen, die serialisiert werden müssen, wie beispielsweise Unterklassen von `RichMapFunction` , `LOGGER` als `transient` zu deklarieren:

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

Verwenden `LOGGER` in Ihrem Code wie üblich `LOGGER` . Verwenden Sie Platzhalter (`{}`) zum Formatieren von Objekten.

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

Protokollierung der Konfiguration

Lokalbetrieb

Im lokalen Modus, zum Beispiel wenn Sie Ihre Anwendung von einer IDE aus `log4j.properties` , können Sie `log4j` wie üblich konfigurieren, indem `log4j.properties` im Klassenpfad eine `log4j.properties` verfügbar machen. In maven können Sie ganz einfach `log4j.properties` im Ordner `src/main/resources` erstellen. Hier ist ein Beispiel:

```
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
```

Standalone-Modus

> Im Standalone-Modus ist die tatsächlich verwendete Konfiguration nicht die in Ihrer `jar` Datei. Das liegt daran, dass Flink eigene Konfigurationsdateien hat, die Vorrang vor Ihren eigenen haben.

Standarddateien : Flink wird mit den folgenden Standardeigenschaftendateien geliefert:

- `log4j-cli.properties` : Wird vom Flink-Befehlszeilen-Client verwendet (z. B. `flink run`) (kein Code, der im Cluster ausgeführt wird)
- `log4j-yarn-session.properties` : Wird vom Flink-Befehlszeilenclient beim Starten einer YARN-Sitzung verwendet (`yarn-session.sh`)
- `log4j.properties` : JobManager / Taskmanager-Protokolle (sowohl Standalone als auch YARN)

Beachten Sie, dass `${log.file}` standardmäßig `flink/log` . Es kann in `flink-conf.yaml` , indem `env.log.dir` wird.

```
env.log.dir
```

definiert das Verzeichnis, in dem die Flink-Protokolle gespeichert werden. Es muss ein absoluter Pfad sein.

Protokollspeicherort : Die Protokolle sind *lokal* , dh sie werden auf den Maschinen erstellt, auf denen die JobManager / Taskmanager ausgeführt werden.

Yarn : Wenn Sie Flink auf Yarn ausführen, müssen Sie sich auf die Protokollierungsfunktionen von Hadoop YARN verlassen. Die nützlichste Funktion dafür ist die YARN-Protokollaggregation . Um es zu aktivieren, setzen Sie die `yarn.log-aggregation-enable` in der `yarn-site.xml file` auf `true` . Wenn dies aktiviert ist, können Sie alle Protokolldateien einer (fehlgeschlagenen) YARN-Sitzung mit folgendem Befehl abrufen:

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

Leider sind Protokolle nur verfügbar , *nachdem eine Sitzung beendet wurde* , beispielsweise nach einem Fehler.

Verwenden unterschiedlicher Konfigurationen für jede Anwendung

Falls Sie für Ihre verschiedenen Anwendungen andere Einstellungen benötigen, gibt es (ab Flink 1.2) keine einfache Möglichkeit, dies zu tun.

Wenn Sie den Modus " *Ein-Garn-Cluster-pro-Job"* für Flink verwenden (dh Sie starten Ihre Skripts mit: `flink run -m yarn-cluster` ...), ist hier eine Problemumgehung:

1. Erstellen Sie ein `conf` Verzeichnis in der Nähe Ihres Projekts
2. Erstelle `flink/conf` für alle Dateien in `flink/conf` :

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

3. Ersetzen Sie die symlink `log4j.properties` (oder jede andere Datei, die Sie ändern möchten) durch Ihre eigene Konfiguration
4. Bevor Sie Ihren Job starten, starten Sie ihn

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

Abhängig von Ihrer Version von flink müssen Sie möglicherweise die Datei `flink/bin/config.sh` . Wenn Sie über diese Linie laufen:

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

Ändern Sie es mit:

```
if [ -z "$FLINK_CONF_DIR" ]; then
```

```
    FLINK_CONF_DIR=$FLINK_ROOT_DIR_MANGLED/conf;
fi
```

Flink-on-Yarn-Workaround: Mit rsyslog Protokolle in Echtzeit abrufen

Garne fassen standardmäßig keine Protokolle zusammen, bevor eine Anwendung abgeschlossen ist. Dies kann bei Streaming-Jobs, die nicht einmal beendet werden, problematisch sein.

Eine `rsyslog` besteht in der Verwendung von `rsyslog` , das auf den meisten Linux-Maschinen verfügbar ist.

Zulassen Sie zunächst eingehende udp-Anforderungen, indem Sie die folgenden Zeilen in `/etc/rsyslog.conf` :

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

Bearbeiten Sie Ihre `log4j.properties` (siehe die anderen Beispiele auf dieser Seite), um `SyslogAppender` zu verwenden:

```
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
```

Das Layout ist wichtig, da rsyslog eine neue Zeile als neuen Protokolleintrag behandelt. Oben werden Zeilenumbrüche (z. B. in Stacktraces) übersprungen. Wenn Sie wirklich wollen, dass mehrzeilige / tabellierte Protokolle "normal" `rsyslog.conf` , bearbeiten Sie `rsyslog.conf` und fügen Sie `rsyslog.conf` hinzu:

```
$EscapeControlCharactersOnReceive off
```

Die Verwendung von `myTag:` am Anfang des `conversionPattern` ist hilfreich, wenn Sie alle Ihre Protokolle in eine bestimmte Datei umleiten möchten. Bearbeiten Sie dazu die `rsyslog.conf` und fügen Sie die folgende Regel hinzu:

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

Protokollierung online lesen: https://riptutorial.com/de/apache-flink/topic/9713/protokollierung

Kapitel 4: Prüfpunkt

Einführung

(getestet mit Flink 1.2 und darunter)

Jede Funktion, Quelle oder jeder Operator in Flink kann stateful sein. Prüfpunkte ermöglichen es Flink, Status und Positionen in den Streams wiederherzustellen, um der Anwendung die gleiche Semantik wie eine fehlerfreie Ausführung zu geben. Es ist der Mechanismus hinter den Garantien der *Fehlertoleranz* und *der einmaligen* Verarbeitung.

Lesen Sie diesen Artikel , um die Interna zu verstehen.

Bemerkungen

Prüfpunkte sind nur nützlich, wenn im Cluster ein Fehler auftritt, beispielsweise wenn ein Taskmanager ausfällt. Sie bleiben nicht bestehen, nachdem der Job selbst fehlgeschlagen ist oder abgebrochen wurde.

Um einen Stateful-Job nach einem Ausfall / Abbruch wieder aufnehmen zu können, sollten Sie sich **Sicherungspunkte** oder **externe Kontrollpunkte (Flink 1.2+) ansehen** .

Examples

Konfiguration und Einrichtung

Die Checkpoint-Konfiguration erfolgt in zwei Schritten. Zuerst müssen Sie ein *Backend* auswählen. Anschließend können Sie das Intervall und den Modus der Prüfpunkte anwendungsspezifisch angeben.

Backends

Verfügbare Backends

Wo die Checkpoints gespeichert werden, hängt vom konfigurierten Backend ab:

- `MemoryStateBackend` : In-Memory-Zustand, Sicherung im JobManager / ZooKeeper-Speicher. Sollte nur für minimale Zustände (Standardeinstellung: max. 5 MB, zum Speichern von Kafka-Offsets) oder für das Testen und lokales Debuggen verwendet werden.
- `FsStateBackend` : Der Status wird auf den TaskManagers im Arbeitsspeicher gehalten, und Status-Snapshots (dh Prüfpunkte) werden in einem Dateisystem (HDFS, DS3, lokales Dateisystem, ...) gespeichert. Diese Einrichtung wird für große Zustände oder lange Fenster und für Hochverfügbarkeitseinstellungen empfohlen.

- `RocksDBStateBackend` : enthält In-Flight-Daten in einer RocksDB-Datenbank, die (standardmäßig) in den TaskManager-Datenverzeichnissen gespeichert ist. Beim Checkpointing wird die gesamte RocksDB-Datenbank in eine Datei geschrieben (wie oben). Verglichen mit dem FsStateBackend sind größere Zustände möglich (nur durch den Festplattenspeicherplatz im Vergleich zur Größe des Taskmanager-Speichers begrenzt), der Durchsatz ist jedoch geringer (Daten sind nicht immer im Speicher, müssen von der Festplatte geladen werden).

Unabhängig vom Backend werden Metadaten (Anzahl der Kontrollpunkte, Lokalisierung usw.) immer im Jobmanager-Speicher gespeichert, und Kontrollpunkte **bleiben nach dem Beenden / Abbrechen der Anwendung nicht bestehen** .

Backend angeben

Sie geben das Backend in der `main` Ihres Programms an mit:

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

Oder setzen Sie das Standard-Backend 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
```

Aktivieren von Checkpoints

Jede Anwendung muss explizit Kontrollpunkte aktivieren:

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

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

Sie können optional einen *Checkpoint-Modus* angeben. Wenn nicht, wird standardmäßig *genau einmal festgelegt* :

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

Der Checkpointing-Modus definiert, welche Konsistenz das System bei Fehlern garantiert. Wenn

Checkpointing aktiviert ist, werden die Datenströme so abgespielt, dass verlorene Teile der Verarbeitung wiederholt werden. Mit `EXACTLY_ONCE` zeichnet das System Prüfpunkte so, dass sich eine Wiederherstellung so verhält, als würden die Operatoren / Funktionen jeden Datensatz "genau einmal" sehen. Mit `AT_LEAST_ONCE` werden die Prüfpunkte auf eine einfachere Weise gezeichnet, die bei der Wiederherstellung normalerweise auf einige Duplikate trifft.

Prüfpunkte testen

Der Code

Hier ist eine einfache Flink-Anwendung, die einen Stateful-Mapper mit einem verwalteten `Integer` Status verwendet. Sie können mit den Variablen `checkpointEnable` , `checkpointInterval` und `checkpointMode` , um ihre Auswirkungen zu sehen:

```
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);
        }
    }
}
```

Das Beispiel ausführen und Fehler simulieren

Um die Checkpoints überprüfen zu können, müssen Sie einen `cluster` starten. Der einfachere Weg ist die Verwendung des Skripts `start-cluster.sh` im `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.
```

Packen Sie jetzt Ihre App und senden Sie sie an flink:

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

Erstellen Sie einige Daten:

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

Die Ausgabe sollte in `flink/logs/flink-<user>-jobmanager-0-<host>.out` . Zum Beispiel:

```
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)
```

Um die Checkpoints zu testen, beenden Sie einfach den Taskmanager (dadurch wird ein Fehler emuliert), erzeugen Sie einige Daten und starten Sie einen neuen:

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

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

Hinweis: Beim Starten eines neuen Taskmanagers wird eine andere Protokolldatei verwendet, nämlich `flink/logs/flink-<user>-jobmanager-1-<host>.out` (beachten Sie das Ganzzahlinkrement).

Was zu erwarten ist

- *Checkpoints deaktiviert* : Wenn Sie während des Fehlers Daten erzeugen, gehen diese definitiv verloren. Aber überraschenderweise haben die Zähler recht!
- *Checkpoints aktiviert* : Kein Datenverlust mehr (und korrekte Zähler).
- *Checkpoints mit dem Modus "Mindestens einmal"* : Möglicherweise werden Duplikate angezeigt, insbesondere wenn Sie ein Checkpoint-Intervall auf eine hohe Anzahl setzen und den Taskmanager mehrmals beenden

Prüfpunkt online lesen: https://riptutorial.com/de/apache-flink/topic/9465/prufpunkt

Kapitel 5: Sicherungspunkte und externisierte Kontrollpunkte

Einführung

Sicherungspunkte sind *"fette", extern gespeicherte Prüfpunkte* , die es uns ermöglichen, ein Stateful-Flink-Programm nach einem dauerhaften Ausfall, einem Abbruch oder einer Code-Aktualisierung fortzusetzen. Vor Flink 1.2 und der Einführung *externalisierter Checkpoints* mussten Savepoints explizit ausgelöst werden.

Examples

Sicherungspunkte: Anforderungen und vorläufige Hinweise

Ein Sicherungspunkt speichert zwei Dinge: (a) die Positionen aller Datenquellen, (b) die Zustände der Operatoren. Sicherungspunkte sind in vielen Situationen nützlich:

- leichte Anwendungscode-Updates
- Flink-Update
- Änderungen in der Parallelität
- ...

Ab **Version 1.3** (auch gültig für frühere Version):

- Kontrollpunkt **muss aktiviert sein,** damit die Sicherungspunkte möglich sind. Wenn Sie vergessen, den Checkpoint explizit zu aktivieren, verwenden Sie:

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

  Sie erhalten:

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

- Bei der Verwendung von Fensteroperationen ist es entscheidend, die Ereigniszeit (vs. Aufnahme- oder Verarbeitungszeit) zu verwenden, um korrekte Ergebnisse zu erzielen.
- **Um** ein Programm aktualisieren und Sicherungspunkte **wiederverwenden zu können, muss eine manuelle Benutzer-ID eingestellt werden** . Dies liegt daran, dass Flink standardmäßig die UID des Operators nach jeder Codeänderung ändert.
- Verkettete Operatoren werden anhand der ID der ersten Aufgabe identifiziert. Es ist nicht möglich, manuell eine ID einer zwischengeschalteten verketteten Task zuzuweisen, z. B. kann in der Kette [a -> b -> c] nur die ID manuell vergeben werden, nicht jedoch b oder c.

Um dies zu umgehen, können Sie die Task-Ketten manuell definieren. Wenn Sie sich auf die automatische ID-Zuweisung verlassen, werden durch eine Änderung des Verkettungsverhaltens auch die IDs geändert (siehe Punkt oben).

Weitere Informationen finden Sie in den FAQ .

Sicherungspunkte

Aufbau

Die Konfiguration befindet sich in der Datei `flink/conf/flink-conf.yaml` (unter Mac OSX über Homebrew ist dies `/usr/local/Cellar/apache-flink/1.1.3/libexec/conf/flink-conf.yaml`).

Flink <1.2 : Die Konfiguration ist der Checkpoint-Konfiguration (Thema verfügbar) sehr ähnlich. Der einzige Unterschied ist, dass es keinen Sinn macht, ein In-Memory-Savepoint-Backend zu definieren, da die Savepoints nach dem Herunterfahren von Flink bestehen bleiben müssen.

```
# 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
```

Hinweis : Wenn Sie kein Backend angeben, ist das Standard-Backend " *jobmanager". Dies* bedeutet, dass Ihre Sicherungspunkte verschwinden, sobald der Cluster heruntergefahren wird. Dies ist nur für das Debugging nützlich.

Flink 1.2+ : Wie in diesem Jira-Ticket erläutert, macht das Speichern eines Sicherungspunkts im Jobmanager wenig Sinn. Sicherungspunkte werden seit Flink 1.2 unbedingt in Dateien gespeichert. Die obige Konfiguration wurde ersetzt durch:

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

Verwendungszweck

Abrufen der Job-ID

Um einen Sicherungspunkt auszulösen, benötigen Sie lediglich die Job-ID der Anwendung. Die Job-ID wird beim Starten des Jobs in der Befehlszeile gedruckt oder kann später mithilfe der `flink list` abgerufen werden:

```
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.
```

Einen Savepoint auslösen

Um einen `flink savepoint <jobID>` auszulösen, verwenden Sie den `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.
```

Beachten Sie, dass Sie auch ein Zielverzeichnis als zweites Argument `flink/bin/flink-conf.yaml` . Das Standardverzeichnis in `flink/bin/flink-conf.yaml` .

In Flink 1.2+ ist es auch möglich, einen Job abzubrechen UND gleichzeitig einen Sicherungspunkt mit der Option `-s` erstellen:

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

Hinweis : Sie können einen Sicherungspunkt verschieben, aber nicht umbenennen!

Wiederaufnahme von einem Sicherungspunkt

Verwenden Sie die Option `-s [savepoint-dir]` des `flink run` Befehls, um von einem bestimmten Sicherungspunkt aus `flink run` :

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

Angabe der Operator-UID

Um nach einer Codeänderung von einem Sicherungspunkt fortfahren zu können, müssen Sie sicherstellen, dass der neue Code dieselbe UID für den Operator verwendet. Um eine UID manuell zuzuweisen, rufen Sie die Funktion `.uid(<name>)` direkt nach dem Operator auf:

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

Externalisierte Kontrollpunkte (Flink 1.2+)

Vor 1.2 bestand die einzige Möglichkeit, einen Kontrollpunkt nach einem Jobabbruch / -abbruch / -fehler auf Dauer zu halten / zu behalten, durch einen Sicherungspunkt, der manuell ausgelöst wird. In Version 1.2 wurden permanente Checkpoints eingeführt.

Persistente Kontrollpunkte verhalten sich sehr ähnlich wie regelmäßige periodische Kontrollpunkte, mit Ausnahme der folgenden Unterschiede:

1. Sie speichern ihre Metadaten in einem permanenten Speicher (wie Sicherungspunkten).
2. Sie werden nicht verworfen, wenn der Besitzjob dauerhaft fehlschlägt. Außerdem können sie so konfiguriert werden, dass sie nicht gelöscht werden, wenn der Job abgebrochen wird.

Es ist also Savepoints sehr ähnlich; Savepoints sind lediglich externe Kontrollpunkte mit etwas mehr Informationen.

Wichtiger Hinweis : Der Checkpoint-Koordinator von Flink behält momentan nur den zuletzt erfolgreich abgeschlossenen Checkpoint. Dies bedeutet, dass der letzte abgeschlossene Prüfpunkt verworfen wird, sobald ein neuer Prüfpunkt abgeschlossen ist. Dies gilt auch für ausgelagerte Kontrollpunkte.

Aufbau

Wo die Metadaten zu [externalisierten] Prüfpunkten gespeichert sind, wird in `flink-conf.yaml` konfiguriert (und kann nicht durch Code überschrieben werden):

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

Beachten Sie, dass dieses Verzeichnis *nur die Prüfpunktmetadaten enthält, die* zum Wiederherstellen des Prüfpunkts erforderlich sind. Die eigentlichen Prüfpunktdateien werden weiterhin in ihrem konfigurierten Verzeichnis gespeichert (`state.bachend.fs.checkpointdir` Eigenschaft `state.bachend.fs.checkpointdir`).

Verwendungszweck

Sie müssen externe Prüfpunkte mithilfe der `getCheckpointConfig()` Methode der Streaming-Umgebung explizit im Code `getCheckpointConfig()` :

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

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

Die verfügbaren `ExternalizedCheckpointCleanup` Modi sind:

- `RETAIN_ON_CANCELLATION` : Der letzte Prüfpunkt und seine Metadaten werden beim Abbruch des

Jobs beibehalten. Es liegt in Ihrer Verantwortung, danach aufzuräumen.

- `DELETE_ON_CANCELLATION` : Der letzte Prüfpunkt wird beim Abbruch gelöscht, dh er ist nur verfügbar, wenn die Anwendung fehlschlägt.

Verwenden Sie die Savepoint-Syntax, um von einem externisierten Prüfpunkt fortzufahren. Zum Beispiel:

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

Sicherungspunkte und externisierte Kontrollpunkte online lesen: https://riptutorial.com/de/apache-flink/topic/9466/sicherungspunkte-und-externisierte-kontrollpunkte

Kapitel 6: So definieren Sie ein benutzerdefiniertes (De) Serialisierungsschema

Einführung

Schemas werden von einigen Connectors (Kafka, RabbitMQ) verwendet, um Nachrichten in Java-Objekte umzuwandeln und umgekehrt.

Examples

Beispiel für ein benutzerdefiniertes Schema

Um ein benutzerdefiniertes Schema zu verwenden, müssen Sie `DeserializationSchema` Schnittstellen `SerializationSchema` oder `DeserializationSchema` implementieren.

```
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;
    }
}
```

Die `MyMessage` Klasse ist wie folgt definiert:

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

So definieren Sie ein benutzerdefiniertes (De) Serialisierungsschema online lesen: https://riptutorial.com/de/apache-flink/topic/9004/so-definieren-sie-ein-benutzerdefiniertes--de--serialisierungsschema

Kapitel 7: Tabellen-API

Examples

Abhängigkeiten von Maven

Um die Table-API zu verwenden, fügen Sie `flink-table` als Maven-Abhängigkeit hinzu (zusätzlich zu `flink-clients` und `flink-core`):

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

Stellen Sie sicher, dass die Scala-Version (hier 2.11) mit Ihrem System kompatibel ist.

Einfache Aggregation aus einer CSV

In Anbetracht der CSV-Datei `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
```

Wir möchten die Leute nach Land und nach Land + Geschlecht zählen:

```
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();
    }
}
```

Die Ergebnisse sind:

```
Germany,2
United States,18

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

Beispiel für Tabellen

Neben `peoples.csv` (siehe *einfache Aggregation aus einem CSV*) haben wir zwei weitere CSVs, die Produkte und Verkäufe darstellen.

`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, name, preis):

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

Wir möchten den Namen und das Produkt für jeden Verkauf von mehr als 40 $ erhalten:

```
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
```

Beachten Sie, dass es wichtig ist, für jede Spalte andere Namen zu verwenden. Andernfalls beschwert sich Flink über "mehrdeutige Namen im Join".

Ergebnis:

```
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
```

Verwenden von externen Senken

Eine Tabelle kann in eine TableSink geschrieben werden, eine generische Schnittstelle zur Unterstützung verschiedener Formate und Dateisysteme. Eine Batch-Tabelle kann nur in eine `BatchTableSink` , während für eine Streaming-Tabelle eine `StreamTableSink` erforderlich `StreamTableSink` .

Derzeit bietet flink nur die `CsvTableSink` Schnittstelle.

Verwendungszweck

Ersetzen Sie in den obigen Beispielen Folgendes:

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

mit:

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

`/tmp/results` ist ein Ordner, da Flink parallele Operationen durchführt. Wenn Sie also über 4 Prozessoren verfügen, befinden sich wahrscheinlich 4 Dateien im Ergebnisordner.

Beachten Sie auch, dass wir explizit `env.execute()` aufrufen: Dies ist notwendig, um einen Flink-Job zu starten, aber in den vorherigen Beispielen hat `print()` dies für uns getan.

Tabellen-API online lesen: https://riptutorial.com/de/apache-flink/topic/8966/tabellen-api

Credits

S. No	Kapitel	Contributors
1	Erste Schritte mit Apache-Flink	Community, Derlin, vdep
2	Daten von Kafka verbrauchen	alpinegizmo, Derlin
3	Protokollierung	Derlin
4	Prüfpunkt	Derlin
5	Sicherungspunkte und externisierte Kontrollpunkte	Derlin
6	So definieren Sie ein benutzerdefiniertes (De) Serialisierungsschema	Derlin
7	Tabellen-API	Derlin