Бесплатная электронная книга

УЧУСЬ apache-flink

Free unaffiliated eBook created from
Stack Overflow contributors.

#apache-flink

Около

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

глава 1: Начало работы с apache-flink

замечания

В этом разделе представлен обзор того, что такое apache-flink, и почему разработчик может захотеть его использовать.

Следует также упомянуть о любых крупных предметах в apache-flink и ссылаться на связанные темы. Поскольку Documentation для apache-flink является новым, вам может потребоваться создать начальные версии этих связанных тем.

Examples

Обзор и требования

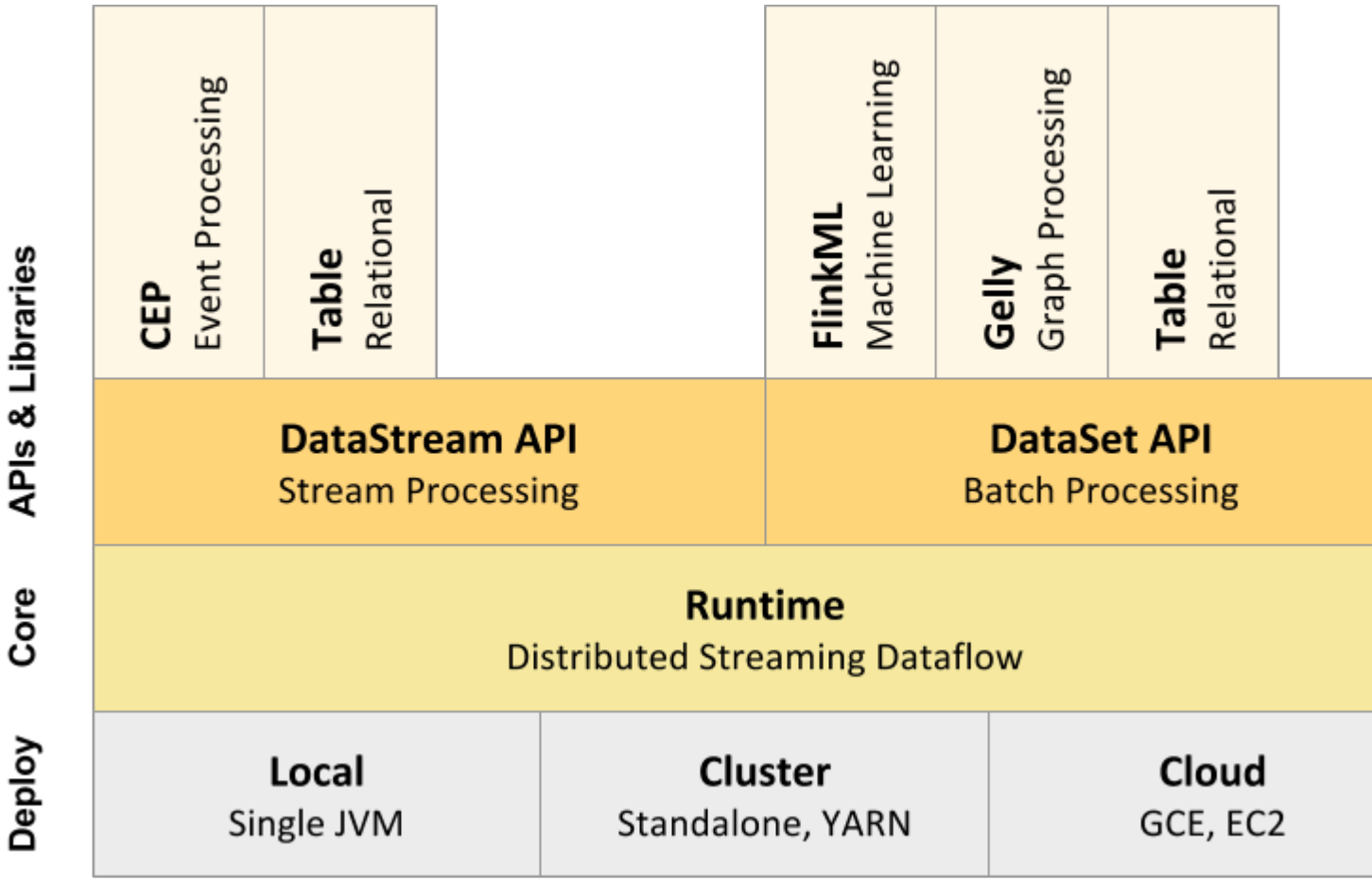
Что такое Флинк

Как Apache Hadoop и Apache Spark , Apache Flink - это среда с открытым исходным кодом для распределенных Big Data Analytics. Написанная на Java, Flink имеет API для Scala, Java и Python, что позволяет анализировать потоковые потоки в реальном времени и в реальном времени.

Требования

- UNIX-подобная среда, такая как Linux, Mac OS X или Cygwin;
- Java 6.X или новее;
- [необязательно] Maven 3.0.4 или новее.

стек



Условия выполнения

Apache Flink - это система обработки данных и **альтернатива компоненту MapReduce от Hadoop** . Он поставляется со своей *собственной версией,* а не поверх MapReduce. Таким образом, он может работать совершенно независимо от экосистемы Hadoop.

`ExecutionEnvironment` - это контекст, в котором выполняется программа. В зависимости от ваших потребностей вы можете использовать различные среды.

1. *Среда JVM* : Flink может работать на одной виртуальной машине Java, позволяя пользователям тестировать и отлаживать программы Flink непосредственно из их IDE. При использовании этой среды все, что вам нужно, это правильные зависимости от maven.

2. *Локальная среда* : чтобы иметь возможность запускать программу в текущем экземпляре Flink (не из вашей среды IDE), вам необходимо установить Flink на свой компьютер. См. Локальную настройку .

3. *Окружение кластера* : работа Flink в полностью распределенном режиме требует автономного или кластера пряжи. Для получения дополнительной информации см. Страницу настройки кластера или это слайд-шоу . mportant__: `2.11` в имени артефакта

- это *версия scala* , обязательно соответствующая той, что у вас есть в вашей системе.

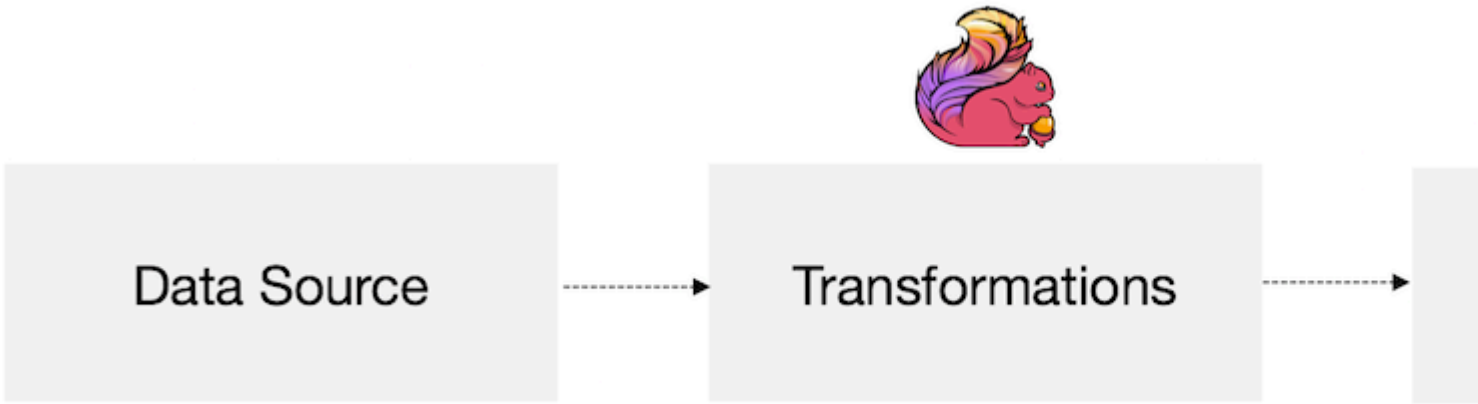
API-интерфейсы

Flink может использоваться для потоковой или пакетной обработки. Они предлагают три API:

- **DataStream API** : потоковая обработка, т. Е. Преобразования (фильтры, временные окна, агрегации) на неограниченные потоки данных.
- **API DataSet** : пакетная обработка, т. Е. Преобразования на наборах данных.
- **Table API** : SQL-подобный язык выражений (например, dataframes в Spark), который может быть встроен как в пакетные, так и потоковые приложения.

Строительные блоки

На самом базовом уровне Flink состоит из источника (ов), трансформаций (-ов) и приемников (-ов).



На самом базовом уровне программа Flink состоит из:

- **Источник данных** : входящие данные, которые связаны с процессами Flink
- **Трансформации** : шаг обработки, когда Flink изменяет входящие данные
- **Приемник данных** : когда Flink отправляет данные после обработки

Источники и приемники могут быть локальными / файлами HDFS, базами данных, очередями сообщений и т. Д. Существует уже много сторонних разъемов, или вы можете легко создать свои собственные.

Локальная настройка времени выполнения

0. убедитесь, что у вас есть java 6 или выше, и что задана переменная среды `JAVA_HOME` .

1. скачать последнюю FLiNK бинарного здесь :

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

   Если вы не планируете работать с Hadoop, выберите версию hadoop 1. Также обратите внимание на загружаемую версию scala, поэтому вы можете добавить правильные зависимости maven в своих программах.

2. start flink:

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

   Flink уже настроен для запуска локально. Чтобы обеспечить работу флинка, вы можете проверять журналы в `flink/log/` или открывать интерфейс jobManager, работающий на `http://localhost:8081` .

3. stop flink:

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

Настройка окружения Flink

Чтобы запустить программу flink из вашей IDE (мы можем использовать либо Eclipse, либо Intellij IDEA (preffered)), вам нужны две зависимости: `flink-java` / `flink-scala` и `flink-clients` (по состоянию на февраль 2016 г.). Эти JARS могут быть добавлены с использованием Maven и SBT (если вы используете scala).

- **специалист**

```
<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>
```

- Название **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"
)
```

важно : `2.11` в имени артефакта - *версия scala* , обязательно соответствующая той, что у вас есть в вашей системе.

API WordCount - таблицы

Этот пример совпадает с *WordCount* , но использует Table API. Подробные сведения о выполнении и результатах см. В *WordCount* .

специалист

Чтобы использовать API таблиц, добавьте `flink-table` в качестве зависимости maven:

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

Код

```
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();
    }
}
```

Примечание . Для версии с использованием Java <8 замените лямбда на анонимный класс:

```
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

специалист

Добавьте зависимости `flink-java` и `flink-client` (как описано в примере *настройки среды JVM*).

Код

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

Если вы используете Java 8, вы можете заменить `.flatmap(new LineSplitter())` выражением лямбда:

```
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 );
```

выполнение

Из среды IDE : просто нажмите *пробег* в своей среде IDE. Flink создаст среду внутри JVM.

Из командной строки flink : для запуска программы с использованием автономной локальной среды выполните следующие действия:

1. убедитесь, что фланг запущен (`flink/bin/start-local.sh`);
2. создать файл jar (`maven package`);

3. используйте `flink` командной строки `flink` (в папке `bin` вашей установки flink), чтобы запустить программу:

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

   Параметр `-c` позволяет указать класс для запуска. Это не обязательно, если jar является исполняемым / определяет основной класс.

Результат

```
(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 - потоковый API

Этот пример совпадает с *WordCount* , но использует Table API. Подробные сведения о выполнении и результатах см. В *WordCount* .

специалист

Чтобы использовать Streaming API, добавьте `flink-streaming` как зависимость от maven:

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

Код

```
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();
    }
}
```

Прочитайте Начало работы с apache-flink онлайн: https://riptutorial.com/ru/apache-flink/topic/5798/начало-работы-с-apache-flink

глава 2: API таблицы

Examples

Зависимости Maven

Чтобы использовать API таблиц, добавьте `flink-table` в качестве зависимости maven (в дополнение к `flink-clients flink-core` и `flink-core`):

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

Убедитесь, что версия scala (здесь 2.11) совместима с вашей системой.

Простая агрегация из CSV

Учитывая CSV-файл `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
```

Мы хотим считать людей по странам и по странам + пол:

```
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();
    }
}
```

Результаты:

```
Germany,2
United States,18

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

Пример использования таблиц

В дополнение к `peoples.csv` (см. *Простое объединение из CSV*) у нас есть еще два CSV, представляющих продукты и продажи.

`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, price):

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

Мы хотим получить имя и продукт для каждой продажи более 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
```

Обратите внимание, что для каждого столбца важно использовать разные имена, иначе flink будет жаловаться на «неоднозначные имена в соединении».

Результат:

```
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
```

Использование внешних стоков

Таблица может быть записана в TableSink, который является общим интерфейсом для поддержки различных форматов и файловых систем. `BatchTableSink` таблица может быть записана только в `BatchTableSink` , а для потоковой таблицы требуется `StreamTableSink` .

В настоящее время flink предлагает только интерфейс `CsvTableSink` .

использование

В приведенных выше примерах замените:

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

с:

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

`/tmp/results` - это папка, потому что flink выполняет параллельные операции. Следовательно, если у вас 4 процессора, у вас, скорее всего, будет 4 файла в папке с результатами.

Кроме того, обратите внимание, что мы явно вызываем `env.execute()` : это необходимо для запуска задания flink, но в предыдущих примерах `print()` сделал это для нас.

Прочитайте API таблицы онлайн: https://riptutorial.com/ru/apache-flink/topic/8966/api-таблицы

глава 3: Checkpointing

Вступление

(проверено на Flink 1.2 и ниже)

Каждая функция, источник или оператор в Flink может быть сдержанной. Контрольные точки позволяют Flink восстанавливать состояние и позиции в потоках, чтобы придать приложению ту же семантику, что и безотказное выполнение. Это механизм за гарантиями *отказоустойчивости* и *точно-раз* обработки.

Прочтите эту статью, чтобы понять внутренности.

замечания

Контрольные точки полезны только тогда, когда в кластере происходит сбой, например, когда сбой диспетчера задач. Они не сохраняются после того, как сама работа потерпела неудачу или была отменена.

Чтобы иметь возможность возобновить работу с состоянием после отказа / отмены, взгляните на **точки сохранения** или **внешние контрольные точки (flink 1.2+)** .

Examples

Конфигурация и настройка

Конфигурация контрольной точки выполняется в два этапа. Во-первых, вам нужно выбрать *бэкэнд* . Затем вы можете указать интервал и режим контрольных точек для каждого приложения.

Backends

Доступные бэкэнд

Если контрольные точки хранятся, зависит от сконфигурированного бэкэнд:

- `MemoryStateBackend` : состояние в памяти, резервное копирование в память JobManager / ZooKeeper. Должен использоваться только для минимального состояния (по умолчанию не более 5 МБ, для хранения смещений Кафки, например) или тестирования и локальной отладки.
- `FsStateBackend` : состояние хранится в памяти TaskManagers, а состояния снимков (т.е.

контрольные точки) хранятся в файловой системе (HDFS, DS3, локальная файловая система, ...). Эта настройка рекомендуется для больших состояний или длинных окон и для высокодоступных установок.

- `RocksDBStateBackend` : хранит данные в полете в базе данных RocksDB, которая (по умолчанию) хранится в каталогах данных TaskManager. При контрольной точке вся база данных RocksDB записывается в файл (например, выше). По сравнению с FsStateBackend он позволяет использовать более крупные состояния (ограниченные только дисковым пространством и размером памяти диспетчера задач), но пропускная способность будет ниже (данные не всегда в памяти должны загружаться с диска).

Обратите внимание, что независимо от бэкэнд, метаданные (количество контрольных точек, локализация и т. Д.) Всегда сохраняются в памяти диспетчера заданий, а контрольные точки **не будут сохраняться после завершения / отмены заявки** .

Указание бэкэнд

Вы указываете бэкэнд в `main` методе своей программы, используя:

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

Или установите бэкэнда по умолчанию в `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
```

Включение контрольных точек

Каждое приложение должно явно включать контрольные точки:

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

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

Вы можете указать *режим контрольной точки* . Если нет, то он по умолчанию *точно один раз* :

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

Режим контрольной точки определяет, какую последовательность гарантирует система при наличии сбоев. Когда контрольная точка активируется, потоки данных воспроизводятся так, что потерянные части обработки повторяются. С `EXACTLY_ONCE` система рисует контрольные точки таким образом, что восстановление ведет себя так, как если бы операторы / функции `EXACTLY_ONCE` каждую запись «ровно один раз». С помощью `AT_LEAST_ONCE` контрольные точки рисуются более простым способом, который обычно сталкивается с некоторыми дубликатами при восстановлении.

Тестирование контрольных точек

Код

Вот простое приложение flink, использующее stateful mapper с управляемым состоянием `Integer` . Вы можете играть с переменными `checkpointEnable` , `checkpointInterval` и `checkpointMode` чтобы увидеть их эффект:

```
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);
        }
    }
}
```

Запуск примера и моделирование сбоя

Чтобы проверить контрольные точки, вам нужно запустить `cluster` . Более простой способ - использовать скрипт `start-cluster.sh` в `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.
```

Теперь, упакуйте приложение и отправьте его, чтобы включить:

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

Создайте несколько данных:

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

Выход должен быть доступен в `flink/logs/flink-<user>-jobmanager-0-<host>.out` . Например:

```
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)
```

Чтобы проверить контрольные точки, просто уберите диспетчер задач (это будет эмулировать сбой), создайте некоторые данные и запустите новый:

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

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

Примечание: при запуске нового диспетчера задач он будет использовать другой файл журнала, а именно `flink/logs/flink-<user>-jobmanager-1-<host>.out` (обратите внимание на приращение целых чисел).

Что ожидать

- *контрольные точки отключены* : если вы производите данные во время сбоя, они будут определенно потеряны. Но удивительно, что счетчики будут правы!
- *контрольные точки* : нет потери данных больше (и правильные счетчики).
- *контрольные точки с режимом «по крайней мере один раз»* : вы можете видеть дубликаты, особенно если вы установите интервал контрольной точки на большое число и несколько раз убейте taskmanager

Прочитайте Checkpointing онлайн: https://riptutorial.com/ru/apache-flink/topic/9465/checkpointing

глава 4: Как определить схему пользовательской (де) сериализации

Вступление

Схемы используются некоторыми коннекторами (Kafka, RabbitMQ) для преобразования сообщений в объекты Java и наоборот.

Examples

Пример пользовательской схемы

Чтобы использовать настраиваемую схему, все, что вам нужно сделать, это реализовать один из интерфейсов `SerializationSchema` или `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;
    }
}
```

Класс `MyMessage` определяется следующим образом:

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

Прочитайте Как определить схему пользовательской (де) сериализации онлайн: https://riptutorial.com/ru/apache-flink/topic/9004/как-определить-схему-пользовательской--де--сериализации

глава 5: Потребляйте данные от Kafka

Examples

Пример KafkaConsumer

`FlinkKafkaConsumer` позволяет вам использовать данные из одной или нескольких тем кафки.

версии

Потребляемый потребитель зависит от вашего распределения кафки.

- `FlinkKafkaConsumer08` : использует старый API `SimpleConsumer` для Kafka. Смещения обрабатываются Flink и передаются в zookeeper.
- `FlinkKafkaConsumer09` : использует новый потребительский API Kafka, который автоматически обрабатывает смещения и балансировку.
- `FlinkKafkaProducer010` : этот разъем поддерживает сообщения Kafka с отметками времени как для производства, так и для потребления (полезно для оконных операций).

использование

Бинарные файлы не являются частью ядра flink, поэтому вам необходимо импортировать их:

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

Конструктор принимает три аргумента:

- одна или несколько тем для чтения из
- схема десериализации, указывающая Flink, как интерпретировать / декодировать сообщения
- kafka. Это те же, что и «обычный» потребитель кафки. Необходимый минимум:
    - `bootstrap.servers` : список брокеров Kafka, разделенный запятыми, в виде ip: port. Для версии 8 вместо этого используйте `zookeeper.connect` (список серверов zookeeper)
    - `group.id` : идентификатор группы потребителей (более подробную информацию см. в документации `group.id`)

В 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));
```

В 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))
```

Во время разработки вы можете использовать свойства `enable.auto.commit=false` и `auto.offset.reset=earliest` чтобы повторно использовать одни и те же данные каждый раз при запуске вашей пограммы.

Отказоустойчивость

Как поясняется в документах ,

> При включенной контрольной точке Flink потребитель Flink Kafka будет потреблять записи из темы и периодически проверять все свои смещения Kafka вместе с состоянием других операций в последовательном порядке. В случае отказа задания Flink восстановит программу потоковой передачи до состояния последней контрольной точки и повторно запишет записи из Kafka, начиная с смещений, которые хранятся в контрольной точке.
>
> Таким образом, интервал контрольных точек рисования определяет, сколько программы, возможно, придется вернуться в лучшем случае, в случае сбоя.

Чтобы использовать отказоустойчивые потребители Kafka, вам необходимо включить контрольную точку в среде исполнения с `enableCheckpointing` метода `enableCheckpointing` :

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

Встроенные схемы десериализации

SimpleStringSchema : `SimpleStringSchema` десериализует сообщение как строку. Если ваши сообщения имеют ключи, последние будут игнорироваться.

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

JSONDeserializationSchema

`JSONDeserializationSchema` десериализует json-форматированные сообщения с помощью *jackson* и возвращает поток объектов `com.fasterxml.jackson.databind.node.ObjectNode` . Затем вы можете использовать метод `.get("property")` для доступа к полям. Опять же, клавиши игнорируются.

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

JSONKeyValueDeserializationSchema

`JSONKeyValueDeserializationSchema` очень похож на предыдущий, но имеет дело с сообщениями с json-закодированными ключами AND.

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

`ObjectNode` содержит следующие поля:

- `key` : все поля, присутствующие в ключе
- `value` : все поля сообщений
- (необязательные) `metadata` : предоставляет `offset` , `partition` и `topic` сообщения (передайте `true` для конструктора, чтобы также получить метаданные).

Например:

```
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
```

Будет декодирован как:

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

Разделы Кафки и параллельность Флинка

В kafka каждому потребителю из той же группы потребителей присваивается один или несколько разделов. Обратите внимание, что два потребителя не могут потреблять из одного раздела. Количество потребителей flink зависит от параллелизма flink (по умолчанию 1).

Возможны три случая:

1. **kafka partitions == flink parallelism** : этот случай идеален, поскольку каждый потребитель заботится о одном разделе. Если ваши сообщения сбалансированы между разделами, работа будет равномерно распределена между операторами flink;

2. **kafka partitions <flink parallelism** : некоторые флинковые экземпляры не получат никаких сообщений. Чтобы этого избежать, вам необходимо вызвать `rebalance` в вашем потоке ввода *перед любой операцией* , которая заставляет данные повторно разбивать:

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

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

3. **kafka partitions> flink parallelism** : в этом случае некоторые экземпляры обрабатывают несколько разделов. Еще раз, вы можете использовать `rebalance` для равномерного распространения сообщений среди рабочих.

Прочитайте Потребляйте данные от Kafka онлайн: https://riptutorial.com/ru/apache-flink/topic/9003/потребляйте-данные-от-kafka

глава 6: протоколирование

Вступление

В этом разделе показано, как использовать и настраивать ведение журнала (log4j) в приложениях Flink.

Examples

Использование регистратора в вашем коде

Добавьте зависимость `slf4j` к вашему `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>
```

Создайте объект журнала для использования в вашем классе:

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

В классах, которые должны быть сериализованы, например подклассы `RichMapFunction` , не забудьте объявить `LOGGER` как `transient` :

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

В своем коде используйте `LOGGER` как обычно. Использовать заполнители (`{}`) для форматирования объектов и т.

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

Конфигурация протоколирования

Локальный режим

В локальном режиме, например, при запуске приложения из среды IDE, вы можете настроить `log4j` как обычно, то есть сделать `log4j.properties` доступным в пути к классам. `log4j.properties` способом в maven является создание `log4j.properties` в `log4j.properties` `src/main/resources` . Вот пример:

```
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
```

Автономный режим

> В автономном режиме фактическая конфигурация используется не в файле `jar` . Это связано с тем, что у Flink есть собственные файлы конфигурации, которые имеют приоритет над вашими собственными.

Файлы по умолчанию : Flink поставляется со следующими файлами свойств по умолчанию:

- `log4j-cli.properties` : используется клиентом командной строки Flink (например, `flink run`) (не выполняется код в кластере)
- `log4j-yarn-session.properties` : используется клиентом командной строки Flink при запуске сеанса YARN (`yarn-session.sh`)
- `log4j.properties` : журналы JobManager / Taskmanager (как автономные, так и YARN)

Обратите внимание, что `${log.file}` умолчанию для `flink/log` . Его можно переопределить в

`flink-conf.yaml` , установив `env.log.dir` ,

> `env.log.dir` определяет каталог, в котором сохраняются журналы Flink. Это должен быть абсолютный путь.

Лог-локация : журналы являются *локальными* , т. Е. Они создаются на компьютере (-ах), на котором запущены JobManager (ы) / Taskmanager (s).

Пряжа : при использовании Flink on Yarn вы должны полагаться на возможности ведения журнала Hadoop YARN. Наиболее полезной особенностью этого является агрегирование журнала YARN . Чтобы включить его, установите для `yarn.log-aggregation-enable` значение `true` в `yarn-site.xml file` . После того, как это включено, вы можете получить все файлы журналов сеанса YARN (сбой), используя:

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

К сожалению, журналы доступны *только после прекращения работы сеанса* , например, после сбоя.

Использование различных конфигураций для каждого приложения

Если вам нужны разные настройки для различных приложений, нет (как и для Flink 1.2) простого способа сделать это.

Если вы используете режим « *один-пряжи-кластер-на-работу* » для флинга (т.е. вы запускаете свои скрипты с помощью: `flink run -m yarn-cluster` ...), это обходное решение:

1. создайте каталог `conf` где-нибудь рядом с вашим проектом
2. создавать символические `flink/conf` для всех файлов в `flink/conf` :

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

3. замените символическую `log4j.properties` (или любой другой файл, который вы хотите изменить) по вашей собственной конфигурации
4. перед запуском вашей работы, запустите

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

В зависимости от вашей версии flink вам может потребоваться отредактировать файл `flink/bin/config.sh` . Если ваш пробег проходит по этой линии:

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

измените его с помощью:

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

Обходное решение Flink-on-Yarn: получение журналов в режиме реального времени с помощью rsyslog

Пряжа по умолчанию не суммирует журналы до того, как приложение завершится, что может быть проблематичным при выполнении потоковых заданий, которые даже не заканчиваются.

`rsyslog` является использование `rsyslog` , который доступен на большинстве Linux-машин.

Во-первых, разрешите входящие запросы udp, раскомментируя следующие строки в `/etc/rsyslog.conf` :

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

Измените свои `log4j.properties` (см. Другие примеры на этой странице), чтобы использовать `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
```

Макет важен, поскольку rsyslog рассматривает новую строку как новую запись в журнале. Выше, новые строки (например, в stacktraces) будут пропущены. Если вы действительно

хотите, чтобы журналы из нескольких `rsyslog.conf` / вкладок работали «нормально», отредактируйте `rsyslog.conf` и добавьте:

```
$EscapeControlCharactersOnReceive off
```

Использование `myTag:` в начале `conversionPattern` полезно, если вы хотите перенаправить все ваши журналы в определенный файл. Для этого отредактируйте `rsyslog.conf` и добавьте следующее правило:

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

Прочитайте протоколирование онлайн: https://riptutorial.com/ru/apache-flink/topic/9713/протоколирование

глава 7: Сохраняющие точки и внешние контрольные точки

Вступление

Savepoints являются *«жирными», хранящимися извне контрольно-пропускными пунктами,* которые позволяют нам возобновить программу flink stateful после постоянного сбоя, отмены или обновления кода. Перед тем, как Flink 1.2 и введение *внешних контрольных точек* , точки сохранения должны были быть вызваны явно.

Examples

Savepoints: требования и предварительные замечания

В точке сохранения хранятся две вещи: (а) позиции всех источников данных, (б) состояния операторов. Сохраняемые точки полезны во многих условиях:

- небольшие обновления кода приложения
- Обновление Flink
- изменения в параллелизме
- ...

Начиная с **версии 1.3** (также для предыдущей версии):

- контрольная точка **должна быть включена,** чтобы точки сохранения были возможны. Если вы забыли явно включить контрольную точку, используя:

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

  ты получишь:

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

- при использовании оконных операций крайне важно использовать время события (против времени проглатывания или обработки) для получения правильных результатов;

- чтобы иметь возможность обновить программу и повторно использовать точки сохранения, **должен быть установлен ручной uid** . Это связано с тем, что по умолчанию Flink изменяет UID оператора после любого изменения своего кода;

- Связанные операторы идентифицируются по идентификатору первой задачи. Невозможно вручную назначить идентификатор промежуточной цепочке, например, в цепочке [a -> b -> c] только один может иметь свой идентификатор, назначенный вручную, но не b или c. Чтобы обойти это, вы можете вручную определить цепочки задач. Если вы полагаетесь на автоматическое присвоение идентификатора, изменение поведения цепочки также изменит идентификаторы (см. Пункт выше).

Дополнительная информация доступна в FAQ .

Точки сохранения

конфигурация

Конфигурация находится в файле `flink/conf/flink-conf.yaml` (в Mac OSX через homebrew это `/usr/local/Cellar/apache-flink/1.1.3/libexec/conf/flink-conf.yaml`).

Flink <1.2 : Конфигурация очень похожа на конфигурацию контрольных точек (тема доступна). Единственное различие заключается в том, что нет смысла определять бэкэнд памяти сохранения в памяти, так как нам нужны точки сохранения, которые сохраняются после выключения 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
```

Примечание . Если вы не укажете бэкэнд, *бэкендом* по умолчанию является *jobmanager* , а это означает, что ваши точки сохранения исчезнут после завершения кластера. Это полезно только для отладки.

Flink 1.2+ : как объясняется в этом билете jira , позволяя сохранить точку сохранения, сохраненную в памяти менеджера, имеет мало смысла. Поскольку Flink 1.2, точки сохранения обязательно сохраняются в файлах. Вышеупомянутая конфигурация была заменена на:

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

использование

Получение идентификатора задания

Для запуска точки сохранения все, что вам нужно, это идентификатор задания приложения. Идентификатор задания печатается в командной строке при запуске задания или может быть восстановлен позднее с использованием `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.
```

Запуск точки сохранения

Для запуска точки сохранения используйте `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.
```

Обратите внимание, что вы также можете `flink/bin/flink-conf.yaml` целевой каталог в качестве второго аргумента, он переопределит значение по умолчанию, определенное в `flink/bin/flink-conf.yaml` .

В Flink 1.2+ также можно отменить задание и одновременно сохранить точку сохранения, используя опцию `-s` :

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

Примечание : можно перемещать точку сохранения, но не переименовывать ее!

Возобновление с точки сохранения

Чтобы возобновить работу с определенной точки сохранения, используйте параметр `-s [savepoint-dir]` команды `flink run` :

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

Указание оператора UID

Чтобы возобновить работу с точки сохранения после изменения кода, вы должны убедиться, что новый код использует тот же UID для оператора. Чтобы вручную назначить UID, вызовите `.uid(<name>)` сразу после оператора:

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

Внешние контрольно-пропускные пункты (Flink 1.2+)

До версии 1.2 единственный способ сохранить состояние / сохранить контрольную точку после завершения / отмены / стойкого отказа работы через точку сохранения, которая запускается вручную. Версия 1.2 вводила постоянные контрольные точки.

Постоянные контрольно-пропускные пункты ведут себя очень похоже на регулярные периодические контрольно-пропускные пункты, за исключением следующих отличий:

1. Они сохраняют свои метаданные в постоянное хранилище (например, точки сохранения).
2. Они не отбрасываются, когда работа по трудоустройству завершается навсегда. Кроме того, они могут быть настроены так, чтобы их нельзя было отменить, когда задание отменено.

Таким образом, он очень похож на точки сохранения; Фактически, точки сохранения - это только внешние контрольные точки с немного дополнительной информацией.

Важное примечание . На данный момент координатор контрольной точки Флинка сохраняет только последнюю успешно пройденную контрольную точку. Это означает, что всякий раз, когда новая контрольная точка завершается, последняя завершенная контрольная точка будет отброшена. Это также относится к внешним контрольно-пропускным пунктам.

конфигурация

Где хранятся метаданные о [внешних] контрольных точках, настроен в `flink-conf.yaml` (и не может быть переопределен через код):

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

Обратите внимание, что этот каталог *содержит только метаданные контрольной точки,* необходимые для восстановления контрольной точки. Фактические файлы контрольных точек все еще хранятся в их сконфигурированном каталоге (т. `state.bachend.fs.checkpointdir` свойство).

использование

Вам необходимо явно включить внешние контрольные точки в коде, используя метод `getCheckpointConfig()` для потоковой среды:

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

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

Доступные режимы `ExternalizedCheckpointCleanup` :

- `RETAIN_ON_CANCELLATION` : последняя контрольная точка и ее метаданные сохраняются при аннулировании работы; это ваша ответственность за очистку после этого.
- `DELETE_ON_CANCELLATION` : последняя контрольная точка удаляется после отмены, то есть она доступна только в случае сбоя приложения.

Чтобы возобновить работу с внешней контрольной точки, используйте синтаксис точки сохранения. Например:

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

Прочитайте Сохраняющие точки и внешние контрольные точки онлайн: https://riptutorial.com/ru/apache-flink/topic/9466/сохраняющие-точки-и-внешние-контрольные-точки

кредиты

S. No	Главы	Contributors
1	Начало работы с apache-flink	Community, Derlin, vdep
2	API таблицы	Derlin
3	Checkpointing	Derlin
4	Как определить схему пользовательской (де) сериализации	Derlin
5	Потребляйте данные от Kafka	alpinegizmo, Derlin
6	протоколирование	Derlin
7	Сохраняющие точки и внешние контрольные точки	Derlin