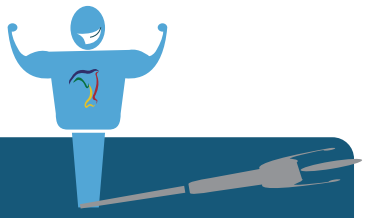
autentia

Avenida de Castilla,1 - Edificio Best Point - Oficina 21B
28830 San Fernando de Henares (Madrid)
tel./fax: +34 91 675 33 06
info@autentia.com - **www.autentia.com**

¿**Qué ofrece** Autentia Real Business Solutions S.L?

Somos su empresa de **Soporte a Desarrollo Informático**.
Ese apoyo que siempre quiso tener...

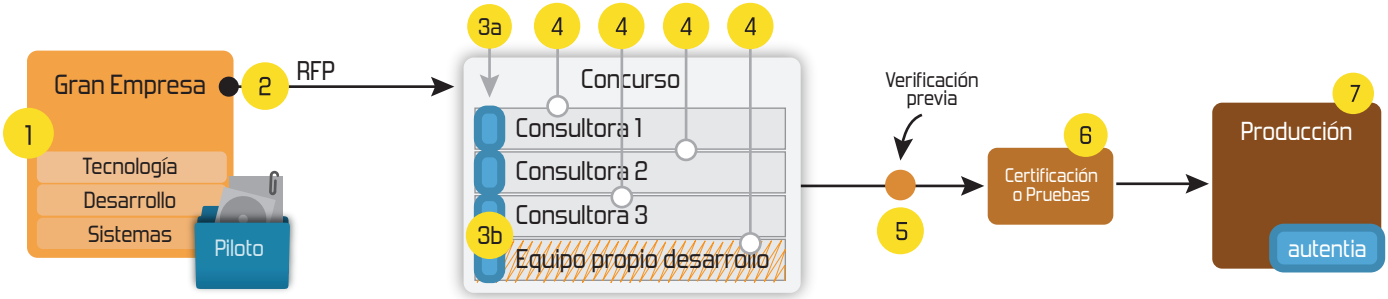
1. Desarrollo de componentes y proyectos a medida



2. Auditoría de código y recomendaciones de mejora

3. Arranque de proyectos basados en nuevas tecnologías

1. Definición de frameworks corporativos.
2. Transferencia de conocimiento de nuevas arquitecturas.
3. Soporte al arranque de proyectos.
4. Auditoría preventiva periódica de calidad.
5. Revisión previa a la certificación de proyectos.
6. Extensión de capacidad de equipos de calidad.
7. Identificación de problemas en producción.



4. Cursos de formación (impartidos por desarrolladores en activo)

- Spring MVC, JSF-PrimeFaces /RichFaces, HTML5, CSS3, JavaScript-jQuery
- Gestor portales (Liferay)
- Gestor de contenidos (Alfresco)
- Aplicaciones híbridas
- Control de autenticación y acceso (Spring Security)
- UDDI
- Web Services
- Rest Services
- Social SSO
- SSO (Cas)
- JPA-Hibernate, MyBatis
- Motor de búsqueda empresarial (Solr)
- ETL (Talend)
- Dirección de Proyectos Informáticos.
- Metodologías ágiles
- Patrones de diseño
- TDD
- Tareas programadas (Quartz)
- Gestor documental (Alfresco)
- Inversión de control (Spring)
- BPM (jBPM o Bonita)
- Generación de informes (JasperReport)
- ESB (Open ESB)

Compartimos nuestro conocimiento en:
www.adictosaltrabajo.com

Para más información visítenos en:
www.autentia.com

Juan Alonso Ramos

Consultor tecnológico de desarrollo de proyectos informáticos.

Ingeniero en Informática, especialidad en Ingeniería del Software

Puedes encontrarme en Autentia: Ofrecemos de servicios soporte a desarrollo, factoría y formación

Somos expertos en Java/J2EE

Ver todos los tutoriales del autor

Fecha de publicación del tutorial: 2014-10-01

Tutorial visitado 9 veces Descargar en PDF

Trident, un compañero de viaje para tratar con Storm

0. Índice de contenidos.

1. Introducción.

En un tutorial anterior vimos una introducción a Apache Storm como un sistema para procesamiento de streams en tiempo real. En este tutorial vamos a ver el API Trident, una abstracción por encima de Storm para facilitar la creación de las topologías. Se trata de una aproximación a lo que es Pig o Cascading para Hadoop, salvando las distancias.

Con Trident podemos configurar una topología que procese una entrada de dato asemejándola a una query SQL, es decir a partir de una fuente de datos podemos manipularla seleccionando y procesando los datos que necesitemos y finalmente persistimos los resultados en alguna unidad de almacenamiento ya sea el HDFS o una base de datos NoSQL.

Puedes descargarte el código del tutorial desde aquí.

2. Entorno.

El tutorial se ha realizado con el siguiente entorno:

- MacBook Pro 15' (2.4 GHz Intel Core i5, 8GB DDR3 SDRAM).
- Oracle Java SDK 1.7.0_60
- Apache Storm 0.9.2-incubating
- Twitter4j 4.0.1

3. Introducción a Trident

Trident es una API para manejar streams de datos. Sobre estos datos podemos realizar funciones de extracción, formateo, agrupamiento, sumas, cuentas, etc. para finalmente persistir el resultado en algún sistema de almacenamiento. A continuación explicamos las funciones más importantes:

3.1 Function

En programación una función recibe una serie de datos en la entrada, realiza algún cálculo con esos datos y devuelve un resultado. Una función en Trident es exáctamente lo mismo. Recibirá un conjunto de campos en la entrada y emitirá en la salida una o más tuplas. Las funciones en Trident serían el equivalente a los Bolts de Storm, digamos que se colocan en mitad del flujo de datos pudiéndoles llegar parte de esos datos. Una vez procesados dentro de la función los emiten pasando a formar parte del flujo original.

Un ejemplo de función es la siguiente:

```
import storm.trident.operation.BaseFunction;
import storm.trident.operation.TridentCollector;
import storm.trident.tuple.TridentTuple;
import backtype.storm.tuple.Values;

public class ParamSplitter extends BaseFunction {
    public void execute(TridentTuple tuple, TridentCollector collector) {
        final String[] values = tuple.getString(0).split(",");

        for (String value : values) {
            collector.emit(new Values(value));
        }
    }
}
```

Imagina que nos llega una cadena de texto con diferentes valores separados por comas, cada uno representa un valor diferente. Por ejemplo si fuera un dato que nos devuelve un sensor de temperatura podría mandarnos una cadena con la fecha, hora, ciudad, temperatura, etc. Cada dato podría ser tratado de forma independiente por lo que antes de nada lo partimos y lo metemos de nuevo en el flujo para que otras funciones lo puedan tratar convenientemente. Recordar que cuando diseñamos este tipo de sistemas debemos tener en mente en todo momento la escabilidad por lo que Storm ya está pensado para que cada función se pueda ejecutar en máquinas independientes.

3.2 Filter

Un filtro es una función que establece una condición para que el dato pueda o no continuar la cadena si cumple la regla definida en el filtro.

```
import storm.trident.operation.BaseFilter;
import storm.trident.tuple.TridentTuple;

public class PositiveFilter extends BaseFilter {
    public boolean isKeep(TridentTuple tuple) {
        return tuple.getInteger(0) > 0;
    }
}
```

3.3 Aggregator

Una operación muy común cuando procesamos datos de un mismo tipo es la función de agregación por lo que Trident dispone de un método para crear nuestros propios agregadores o bien nos proporciona los más comunes como la suma o cuenta de valores:

Existen 3 tipos de agregadores: CombinerAggregator, ReducerAggregator y Aggregator.

Un CombinerAggregator debe devolver una tupla con un único campo de salida. Durante la ejecución del flujo se llamará al agregador con cada tupla de entrada y se irán combinando valores hasta el final del stream. Es útil también para combinar resultados en un único nodo antes de transferirlo por la red optimizando los recursos.

```
package storm.trident.operation.builtin;

import clojure.lang.Numbers;
import storm.trident.operation.CombinerAggregator;
import storm.trident.tuple.TridentTuple;

public class Sum implements CombinerAggregator<Number> {

    @Override
    public Number init(TridentTuple tuple) {
        return (Number) tuple.getValue(0);
    }

    @Override
    public Number combine(Number val1, Number val2) {
        return Numbers.add(val1, val2);
    }

    @Override
    public Number zero() {
        return 0;
    }
}
```

Storm llamara al método init() con cada tupla y posteriormente al combine hasta que la partición sea procesada. Los valores que se le pasan al combine() son parcialmente agregados.

El ReducerAggregator es bastante similar al CombinerAggregator con la diferencia que Storm llamará al reduce() hasta que la partición sea procesada completamente.

Por último tenemos el interfaz Aggregate para poder implementar nuestros propios agregadores:

```
import storm.trident.tuple.TridentTuple;

public interface Aggregator<T> extends Operation {
    T init(Object batchId, TridentCollector collector);
    void aggregate(T val, TridentTuple tuple, TridentCollector collector);
    void complete(T val, TridentCollector collector);
}
```

Lo interesante de un Aggregator es que le podemos pasar a la función cualquier tipo por lo que es muy flexible pudiendo pasarle por ejemplo mapas, muy útiles para agrupar por distintas claves y poder hacer una cuenta sobre ellas.

3.4 Métodos para el procesamiento de streams

- **each**

  Este método se utiliza para indicar la función o filtro que vamos a utilizar para procesar una tupla del stream de datos.

```
stream.each(new Fields("str"), new ParamSplitter(), new Fields("date, hour, city,
```

formato europeo

Últimos Tutoriales del Autor

» Introducción a Apache Storm

» Primeros pasos con Neo4j

» Testing de Hadoop con MRUnit

» Introducción a Spring Data Hadoop

» Implementar una función UDF de Apache Pig

Categorías del Tutorial

Big Data

Indicamos que todas las tuplas del stream de entrada se pasen por la función ParamSplitter que como vimos antes se encargaba de dividir los campos separados por comas. La función emitirá tuplas en la salida pero es mediante el método each() donde configuramos el nombre de estas tuplas, similar a como cuándo configuramos los spouts y bols en una topología de Storm.

- **partitionAggregate**

  Ejecuta una función de agregación y su resultado reemplazará las tuplas de entrada. Se ejecuta con todas las tuplas de todos los nodos.

```
stream.partitionAggregate(new Fields("values"), new Sum(), new Fields("sum"))
```

- **aggregate**

  Ejecuta una función de agregación en un único nodo de forma aislada.

```
stream.partitionAggregate(new Fields("values"), new Count(), new Fields("count"))
```

- **project**

  El método project() sirve para mantener en el flujo únicamente los campos especificados en la operación. Por ejemplo si el flujo tuviera los campos ["a","b","c","d"], si únicamente queremos quedarnos con los campos "a" y "b" realizaríamos la proyección:

```
stream.project(new Fields("a", "b"))
```

- **parallelismHint**

  Configura la operación sobre una función para que sea ejecutada con el grado de paralelismo que le indiquemos.

```
stream.each(new Fields("str"), new ParamSplitter(), new Fields("date, hour, city,
```

- **partitionBy**

  Encamina las tuplas para que sean procesadas las del mismo tipo en el mismo nodo de destino.

```
stream.partitionBy(new Fields("date"))
```

- **shuffle**

  Utilizar el algoritmo round robin para redistribuir equitativamente las tuplas.

- **groupBy**

  Se utiliza para agrupar las tuplas por un tipo.

```
stream.groupBy(new Fields("hashtag"))
```

3.5 State

Para almacenar el estado de las operaciones Trident soporta múltiples fuentes, desde almacenarlo en memoria, persistir en HDFS o almacenar los resultados en una base de datos NoSQL como Cassandra, Memcached, Redis, etc.

La administración del estado es tolerante a fallos mediante el procesado de las tuplas por lotes más pequeños asignando identificadores únicos que a la hora de persistir son consultados por el estado para mantener la consistencia de los datos.

```
TridentTopology topology = new TridentTopology();
TridentState wordCounts = topology.newStream("spout", spout)
        .each(new Fields("sentence"), new Split(), new Fields("word"))
        .groupBy(new Fields("word"))
        .persistentAggregate(new MemoryMapState.Factory(), new Count(), new Fields("count
```

La lógica necesaria para gestionar las transacciones se realiza en la clase MemoryMapState proporcionada por el API de Trident.

La topología Storm creada a través de Trident se define mediante la clase TridentTopology donde se le indica el flujo de entrada de datos o spout. Este spout realiza la ingesta de datos metiendo streams de sentencias que serán procesadas por la función Split() que se encargará de separarlas por palabras. Una vez que se van separando se realiza una agrupación por cada palabra y una cuenta por número de apariciones que se almacenará en memoria.

4. Implementar un trending topics de Twitter con Trident.

Para ilustrar con un ejemplo lo que hemos visto sobre el API Trident vamos a construir una topología que se encargue de consumir los tweets recibidos de Twitter, extraiga los hashtags que tenga el tweet, si es que tiene alguno, y realice una cuenta para finalmente sacar una lista de los trending topics.

La topología será bastante parecida a la del tutorial de Storm. Para empezar vamos a crear un spout para recoger los tweets. En este caso nuestro spout implementa el interfaz IBatchSpout. A Twitter le vamos a pedir tweets que contengan información sobre equipos de fútbol para ver sobre qué se habla más en relación a los equipos de fútbol de España. Que me perdonen los aficionados del resto de equipos de fútbol pero por simplificar únicamente he puesto los 3 primeros en la clasificación de la temporada 13-14. Podéis probar metiendo diferentes topics si lo preferís ;)

```
import java.util.Map;
import java.util.concurrent.LinkedBlockingQueue;

import storm.trident.operation.TridentCollector;
import storm.trident.spout.IBatchSpout;
import twitter4j.FilterQuery;
import twitter4j.StallWarning;
import twitter4j.Status;
import twitter4j.StatusDeletionNotice;
import twitter4j.StatusListener;
import twitter4j.TwitterStream;
import twitter4j.TwitterStreamFactory;
```

```
import backtype.storm.Config;
import backtype.storm.task.TopologyContext;
import backtype.storm.tuple.Fields;
import backtype.storm.tuple.Values;
import backtype.storm.utils.Utils;

@SuppressWarnings({"serial", "rawtypes"})
public class TwitterConsumerBatchSpout implements IBatchSpout {

    private LinkedBlockingQueue<Status> queue;
    private TwitterStream twitterStream;

    @Override
    public void open(Map conf, TopologyContext context) {
        this.twitterStream = new TwitterStreamFactory().getInstance();
        this.queue = new LinkedBlockingQueue<Status>();

        final StatusListener listener = new StatusListener() {

            @Override
            public void onStatus(Status status) {
                queue.offer(status);
            }

            @Override
            public void onDeletionNotice(StatusDeletionNotice sdn) {
            }

            @Override
            public void onTrackLimitationNotice(int i) {
            }

            @Override
            public void onScrubGeo(long l, long l1) {
            }

            @Override
            public void onException(Exception e) {
            }

            @Override
            public void onStallWarning(StallWarning warning) {
            }
        };

        twitterStream.addListener(listener);

        final FilterQuery query = new FilterQuery();
        query.track(new String[]{"atleti", "fcbarcelona", "realmadrid"});
        query.language(new String[]{"es"});

        twitterStream.filter(query);
    }

    @Override
    public void emitBatch(long batchId, TridentCollector collector) {
        final Status status = queue.poll();
        if (status == null) {
            Utils.sleep(50);
        } else {
            collector.emit(new Values(status));
        }
    }

    @Override
    public void ack(long batchId) {
    }

    @Override
    public void close() {
        twitterStream.shutdown();
    }

    @Override
    public Map getComponentConfiguration() {
        return new Config();
    }

    @Override
    public Fields getOutputFields() {
        return new Fields("tweet");
    }
}
```

Lo siguiente será crear la topología. Creamos un método main donde configuramos la topología, por simplicidad la arrancaremos en modo local. Construimos la topología trident pasándole el spout creado anteriormente.

```

import java.io.IOException;

import storm.trident.TridentTopology;
import storm.trident.operation.builtin.Count;
import storm.trident.operation.builtin.Debug;
import storm.trident.spout.IBatchSpout;
import storm.trident.testing.MemoryMapState;
import backtype.storm.Config;
import backtype.storm.LocalCluster;
import backtype.storm.generated.StormTopology;
import backtype.storm.tuple.Fields;

import com.autentia.tutoriales.functions.HashtagExtractor;
import com.autentia.tutoriales.spout.TwitterConsumerBatchSpout;

public class TrendingTopicsTridentTopology {

    public static StormTopology createTopology(IBatchSpout spout) throws IOException {
        final TridentTopology topology = new TridentTopology();

```

```
        topology.newStream("spout", spout)
                .each(new Fields("tweet"), new HashtagExtractor(), new Fields("hashtag")
                .groupBy(new Fields("hashtag"))
                .persistentAggregate(new MemoryMapState.Factory(), new Count(), new Fiel
                .newValuesStream()
                .each(new Fields("hashtag", "count"), new Debug());

        return topology.build();
    }

    public static void main(String[] args) {
        final Config conf = new Config();
        final LocalCluster local = new LocalCluster();
        final IBatchSpout spout = new TwitterConsumerBatchSpout();

        try {
            local.submitTopology("hashtag-count-topology", conf, createTopology(spout));
        } catch (IOException e) {
            throw new RuntimeException(e);
        }
    }
}
```

Toda topología debe recibir un stream de datos, en nuestro caso el TwitterConsumerBatchSpout que irá metiendo en el sistema los tweets recibidos, a continuación configuramos las operaciones. En primer lugar, cada tweet se procesa en la función HashtagExtractor para extraer únicamente los hashtags que son pasados de nuevo al flujo para posteriormente ser agrupados. Posteriormente a la agrupación se realiza una cuenta sobre ellos siendo necesario almacenar el estado de la misma, en este caso se realiza en memoria.

Para terminar y poder ver los resultados obtenidos pasamos los valores "hashtag" y "count" por la función Debug que los imprime por consola.

La función HashtagExtractor es muy sencilla:

```
import storm.trident.operation.BaseFunction;
import storm.trident.operation.TridentCollector;
import storm.trident.tuple.TridentTuple;
import twitter4j.HashtagEntity;
import twitter4j.Status;
import backtype.storm.tuple.Values;

@SuppressWarnings("serial")
public class HashtagExtractor extends BaseFunction {

    @Override
    public void execute(TridentTuple tuple, TridentCollector collector) {
        final Status status = (Status) tuple.get(0);

        for (HashtagEntity hashtag : status.getHashtagEntities()) {
            collector.emit(new Values(hashtag.getText()));
        }
    }
}
```

Recoge la tupla cero que contiene el tweet, extrae sus hashtags y los emite a la topología.

Si ejecutamos la clase TrendingTopicsTridentTopology veremos cómo van apareciendo hashtags por consola con un contador que irá incrementándose sucesivamente.

```
DEBUG: [FelizLunes, 5]
DEBUG: [futbol, 8]
DEBUG: [FCBarcelona, 15]
DEBUG: [Baloncesto, 2]
DEBUG: [HalaMadrid, 11]
DEBUG: [HM, 2]
DEBUG: [ReyesDeEuropa, 2]
DEBUG: [RealMadrid, 28]
DEBUG: [Atleti, 10]
DEBUG: [Cholismo, 3]
DEBUG: [ElChiringuitoDeNeox, 2]
DEBUG: [LigaBBVA, 2]
```

5. Conclusiones.

Storm va madurando poco a poco y se está posicionando en el mercado como uno de los mejores productos para procesamiento de datos en real time junto a otros como Spark.

Los usuarios de Storm estamos de enhorabuena ya que recientemente (el 29 de septiembre de 2014) Storm ha pasado a ser considerado como un Top-Level Project (TLP) dentro de Apache, muy buena noticia ya que supone la graduación de este fantástico framework de procesamiento en tiempo real.

Puedes descargarte el código del tutorial desde aquí.

Espero que te haya sido de ayuda.

Un saludo.

Juan

Anímate y coméntanos lo que pienses sobre este **TUTORIAL:**

» **Registrate** y accede a esta y otras ventajas «