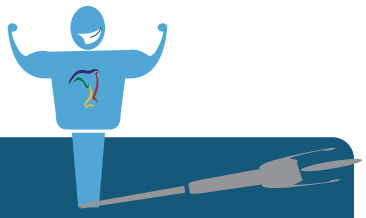
autentia

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...

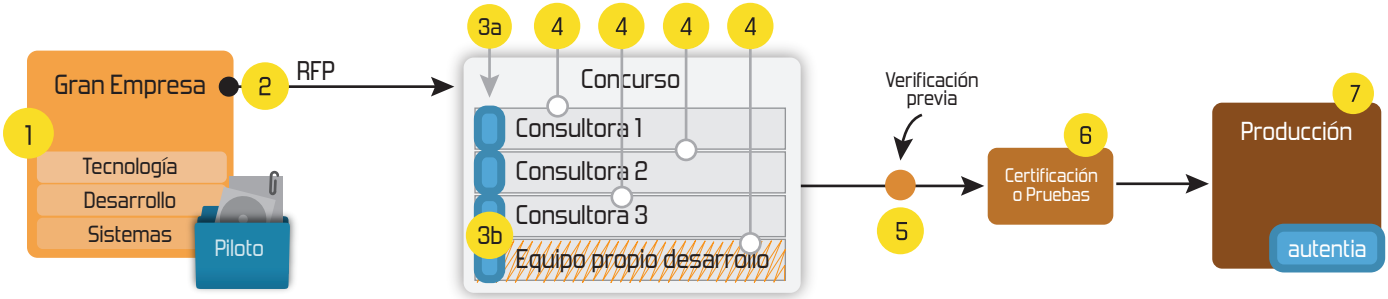
1. Desarrollo de componentes y proyectos a medida



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: 2014-03-20 Tutorial visitado 2 veces Descargar en PDF

Implementando tu propio Writable en Hadoop

0. Índice de contenidos.

1. Introducción.

En el anterior tutorial de primeros pasos con MapReduce vimos la forma de implementar un algoritmo MapReduce con Hadoop para calcular el valor medio de los niveles de monóxido de carbono (CO) de cada una de las nueve provincias de Castilla y León. El algoritmo era muy sencillo, nos bastó con emitir en el mapper como clave la provincia y como valor la medida del nivel de CO. El reducer recibía para cada provincia una lista de todos sus niveles y se encargaba de calcular la media.

En este tutorial vamos a extraer algo más de información del dataset, por ejemplo vamos a sacar por cada año la provincia donde se ha registrado el nivel más alto de una sustancia de las que vienen recogidas en el fichero csv. Como en este caso necesitamos tres valores de cada registro, año, provincia y valor de la medida necesitamos crear un registro compuesto para componer la clave que contendrá provincia y año. Para ello vamos a necesitar implementar nuestro propio Writable.

Hadoop por defecto ya nos proporciona unos Writables para los tipos básicos:

- Text para serializar String
- IntWritable para serializar Integer
- FloatWritable (Float), LongWritable (Long), ByteWritable (Byte), DoubleWritable (Double)
- NullWritable para emitir nulos.

Para casos sencillos como vamos a ver ahora, únicamente recogemos 3 valores, implementar nuestro propio Writable nos soluciona el problema. Si por el contrario necesitamos recoger muchos más campos tendríamos que recurrir a otras estrategias de serialización utilizando librerías como Avro o Thrift de Apache.

Puedes descargarte el código del tutorial desde mi repositorio de github pinchando aquí.

2. Entorno.

El tutorial se ha realizado con el siguiente entorno:

- Ubuntu 12.04 64 bits
- Oracle Java SDK 1.6.0_27
- Apache Hadoop 2.2.0
- Apache Maven 3.1.1

3. El Writable

Vamos a partir del proyecto que creamos en el tutorial de primeros pasos con MapReduce. En este caso vamos a implementar nuestro propio Writable que utilizaremos para almacenar la clave compuesta por el año y la provincia donde se tomó la medida. Para crear nuestro Writable customizado vamos a crear una clase que implemente el interfaz **WritableComparable**.

Este interfaz del API de Hadoop es una subinterfaz de la interfaz **Writable** de Hadoop y la interfaz **Comparable** de toda la vida. Todo Writable debe tener un constructor por defecto para que el framework MapReduce pueda instanciarlo. Debemos implementar los métodos **write** utilizado por Hadoop para serializar los valores del objeto a la salida del map y el método **readFields** de donde los leerá posteriormente para pasárselos a la tarea reduce. Es recomendable también implementar los métodos hashCode e equals.

Al implementar también de Comparable debemos implementar el método compareTo utilizado para ordenar las claves en la fase de suffle and sort.

```
1  public class MeasureWritable implements WritableComparable<MeasureWritable> {
2
3      private String year;
4      private String province;
5
6      public MeasureWritable() {
7
8      }
9
10     public MeasureWritable(String year, String province) {
11         this.year = year;
12         this.province = province;
13     }
14
15     @Override
16     public void write(DataOutput out) throws IOException {
17         Text.writeString(out, year);
18         Text.writeString(out, province);
19     }
20
21     @Override
22     public void readFields(DataInput in) throws IOException {
23         year = Text.readString(in);
24         province = Text.readString(in);
25     }
26
27     public String getYear() {
28         return this.year;
29     }
30
31     @Override
32     public int hashCode() {
33         return new HashCodeBuilder().append(province).append(year).toHashCode();
34     }
35
36     @Override
37     public boolean equals(Object o) {
38         if (!(o instanceof MeasureWritable)) {
39             return false;
40         }
41
42         final MeasureWritable other = (MeasureWritable) o;
43         return new EqualsBuilder().append(province, other.province).append(year, other.y
44     }
```

Por eficiencia es muy recomendable implementar adicionalmente un comparador que pueda comparar los registros de nuestro Writable sin necesidad de deserializarlos en objetos Java.

```
1  public static class Comparator extends WritableComparator {
2      private static final Text.Comparator TEXT_COMPARATOR = new Text.Comparator();
3
4      public Comparator() {
5          super(MeasureWritable.class);
6      }
7
8      @Override
9      public int compare(byte[] b1, int s1, int l1, byte[] b2, int s2, int l2) {
10         try {
11             int firstL1 = WritableUtils.decodeVIntSize(b1[s1]) + readVInt(b1, s1);
12             int firstL2 = WritableUtils.decodeVIntSize(b2[s2]) + readVInt(b2, s2);
13             return TEXT_COMPARATOR.compare(b1, s1, firstL1, b2, s2, firstL2);
14         } catch (IOException e) {
15             throw new IllegalArgumentException(e);
16         }
17     }
18 }
19
20 static {
21     WritableComparator.define(MeasureWritable.class, new Comparator());
22 }
```

El bloque estático se encarga de registrar el comparador implementado para ser usado por defecto en la clase MeasureWritable. Es muy eficiente este comparador ya que funciona a nivel de byte.

4. Mapper

» Como configurar CloudFlare en nuestra web

Últimos Tutoriales del Autor

» Primeros pasos de MapReduce con Hadoop

» Primeros pasos con Hadoop: instalación y configuración en Linux

» Trabajando con Mule ESB

» Crear un proyecto de Mule ESB con Mule Studio

» Crear un proyecto de Mule ESB con Maven

Categorías del Tutorial

Big Data

Una vez que tenemos nuestro propio Writable encargado de almacenar el año y la provincia vamos a usarlo desde nuestro mapper.

```
public static class AirQualityMapper extends Mapper<Object, Text, MeasureWritable, Flo ?:
    private static final String DATE_SEPARATOR = "/";
    private String measureType;

    @Override
    protected void setup(Context context) throws IOException, InterruptedException {
        this.measureType = context.getConfiguration().get(MEASURE_TYPE);
    }

    public void map(Object key, Text value, Context context) throws IOException, Interru
        final String[] values = value.toString().split(SEPARATOR);
        final MeasureWritable measure = getMeasure(values, measureType);
        final String measureValue = format(values[MeasureType.getOrder(measureType)]);

        if (measure != null && NumberUtils.isNumber(measureValue)) {
            context.write(measure, new FloatWritable(Float.valueOf(measureValue)));
        }
    }

    private MeasureWritable getMeasure(String[] values, String measureType) {
        MeasureWritable measureWritable = null;

        final String date = format(values[DATE_ORDER]);

        if (isValidData(date)) {
            final String year = date.split(DATE_SEPARATOR)[2];
            final String province = format(values[PROVINCE_ORDER]);

            measureWritable = new MeasureWritable(year, province);
        }

        return measureWritable;
    }

    private boolean isValidData(final String date) {
        return date.contains(DATE_SEPARATOR);
    }

    private String format(String value) {
        return value.trim();
    }
}
```

La salida de nuestro mapper será la compuesta por la tupla **[MeasureWritable, FloatWritable]** que emitirá los valores de nuestro dataset registrando el año y la provincia de la medida como clave y el valor de la medida como valor que emitiremos al reduce. Para poder aprovechar mucho mejor el dataset con los valores de diferentes medidas de la contaminación, la entrada al mapper es parametrizable, es decir cogerá el valor que se le indique por la entrada estándar. Estos valores se corresponden con las posiciones de los datos presentes en el registro.

El método **setup** se llamará una única vez antes de ejecutar las tareas map y es muy utilizado para inicializar algún recurso. En este caso lo he utilizado para recoger el parámetro indicado para el tipo de medida a consultar. Este parámetro será añadido en el método run cuando se crea el Job.

Para que resulte un poco más sencillo se han guardado en un enumerado asociando el tipo de medida con su posición en la línea del fichero.

```
public enum MeasureType {                                                              ?

    CO("co", 1),
    NO("no", 2),
    NO2("no2", 3),
    O3("o3", 4),
    PM10("pm10", 5),
    SH2("sh2", 6),
    PM25("pm25", 7),
    PST("pst", 8),
    SO2("so2", 9);

    private final String type;
    private final int order;

    private MeasureType(String value, int order) {
        this.type = value;
        this.order = order;
    }

    public String getType() {
        return type;
    }

    public static int getOrder(String type) {

        for (MeasureType measureType : MeasureType.values()) {
            if (measureType.getType().equals(type)) {
                return measureType.order;
            }
        }

        // Value by default
        return MeasureType.CO.order;
    }
}
```

5. Reducer

El reducer es muy sencillo, recibirá como clave el writable que contiene por cada año y provincia la lista de medidas tomadas y se encargará de encontrar la mayor de todas. Una vez encontrada emitirá la salida.

```
public static class AirQualityReducer extends Reducer<MeasureWritable, FloatWritable,

    public void reduce(MeasureWritable key, Iterable<FloatWritable> values, Context cont
        float maxMeasure = 0f;
        for (FloatWritable measureValue : values) {
            maxMeasure = Math.max(maxMeasure, measureValue.get());
        }

        context.write(key, new FloatWritable(maxMeasure));
    }
}
```

El Driver encargado de la ejecución del MapReduce es el típico para una clase configurada con ToolRunner.

```
@Override
public int run(String[] args) throws Exception {

    if (args.length != 3) {
        System.err.println("AirQualityManager required params: <input file> <output dir>
        System.exit(2);
    }

    deleteOutputFileIfExists(args);

    final Configuration configuration = new Configuration();
    configuration.set(MEASURE_TYPE, args[2]);

    final Job job = new Job(configuration);

    job.setJarByClass(AirQualityManager.class);
    job.setInputFormatClass(TextInputFormat.class);
    job.setOutputFormatClass(TextOutputFormat.class);

    job.setMapOutputKeyClass(MeasureWritable.class);
    job.setMapOutputValueClass(FloatWritable.class);
    job.setOutputKeyClass(MeasureWritable.class);
    job.setOutputValueClass(FloatWritable.class);

    job.setMapperClass(AirQualityMapper.class);
    job.setReducerClass(AirQualityReducer.class);

    FileInputFormat.addInputPath(job, new Path(args[0]));
    FileOutputFormat.setOutputPath(job, new Path(args[1]));

    job.waitForCompletion(true);

    return 0;
}

private void deleteOutputFileIfExists(String[] args) throws IOException {
    final Path output = new Path(args[1]);
    FileSystem.get(output.toUri(), getConf()).delete(output, true);
}

public static void main(String[] args) throws Exception {
    ToolRunner.run(new AirQualityManager(), args);
}
```

Para pasarle el argumento de la entrada que indica el tipo de medida lo hacemos a través de la clase **Configuration** que recibe el Job cuando es creado.

Ahora que ya tenemos todo el código de nuestro algoritmo lo ejecutamos indicando el tipo de medida que vamos a utilizar, por ejemplo CO. La salida una vez ejecutado el algoritmo queda de la siguiente manera:

```
(1997) - BURGOS     7.2
(1998) - SALAMANCA  9.0
(1999) - SALAMANCA  11.1
(2000) - LEóN       25.1
(2001) - ÁVILA      5.7
(2002) - ZAMORA     5.5
(2003) - SALAMANCA  8.6
(2004) - SALAMANCA  3.8
(2005) - SEGOVIA    4.3
(2006) - BURGOS     4.0
(2007) - BURGOS     2.8
(2008) - SALAMANCA  2.1
(2009) - SALAMANCA  1.4
(2010) - BURGOS     1.3
(2011) - SORIA      1.3
(2012) - PALENCIA   1.0
(2013) - BURGOS     1.1
```

Como se puede observar, nos muestra un registro por cada año, indicando la provincia que tuvo el mayor índice registrado de monóxido de carbono y su nivel.

Vamos a probar a sacar los datos de otras medidas que contiene el dataset.

Dióxido de azufre (SO2)

```
(1997) - ZAMORA     360.0
(1998) - ÁVILA      261.0
(1999) - LEóN       344.0
(2000) - ZAMORA     218.0
(2001) - LEóN       364.0
(2002) - SALAMANCA  176.0
(2003) - SEGOVIA    246.0
(2004) - BURGOS     326.0
(2005) - VALLADOLID 202.0
```

```
10  (2006) - PALENCIA   247.0
11  (2007) - LEóN       155.0
12  (2008) - LEóN       81.0
13  (2009) - VALLADOLID 78.0
14  (2010) - PALENCIA   76.0
15  (2011) - SEGOVIA    59.0
16  (2012) - VALLADOLID 67.0
17  (2013) - LEóN       45.0
```

Dióxido de azufre (NO2)

```
 1  (1997) - ÁVILA      228.0
 2  (1998) - PALENCIA   202.0
 3  (1999) - LEóN       214.0
 4  (2000) - BURGOS     211.0
 5  (2001) - LEóN       193.0
 6  (2002) - BURGOS     178.0
 7  (2003) - BURGOS     188.0
 8  (2004) - LEóN       121.0
 9  (2005) - SALAMANCA  116.0
10  (2006) - LEóN       147.0
11  (2007) - BURGOS     119.0
12  (2008) - BURGOS     128.0
13  (2009) - LEóN       150.0
14  (2010) - LEóN       101.0
15  (2011) - SALAMANCA  85.0
16  (2012) - LEóN       64.0
17  (2013) - VALLADOLID 65.0
```

Sin ser muy entendido en la materia a simple vista se puede sacar una conclusión muy importante después de analizar varias medidas tomadas durante varios años y es que los índices de contaminación en general van descendiendo año tras año. Una muy buena noticia!!!

6. Conclusiones.

En esta ocasión hemos aprovechado un poquito más el potencial de Hadoop para analizar un fichero con datos estructurados que nos han ayudado a extraer información importante. Es un pequeño ejemplo de lo mucho que se puede hacer con este framework ya que resulta sencillo programar algoritmos MapReduce y lo más importante que puedan escalar si fuera necesario.

Puedes descargarte el código del tutorial desde mi repositorio de github pinchando aquí.

Espero que te haya sido de ayuda.

Un saludo.

Juan

A continuación puedes evaluarlo:

Regístrate para evaluarlo

Por favor, vota +1 o compártelo si te pareció interesante

Share |

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

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