一、 课程计划

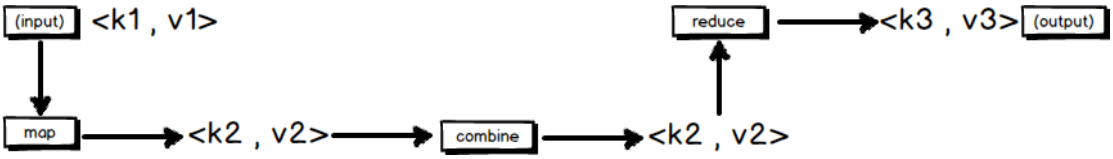
目录

二、 深入 MapReduce

1. MapReduce 的输入和输出

MapReduce 框架运转在<key,value>键值对上，也就是说，框架把作业的输入看成是一组<key,value>键值对，同样也产生一组<key,value>键值对作为作业的输出，这两组键值对可能是不同的。

一个 MapReduce 作业的输入和输出类型如下图所示：可以看出在整个标准的流程中，会有三组<key,value>键值对类型的存在。



2. MapReduce 的处理流程解析

2.1. Mapper 任务执行过程详解

- **第一阶段**是把输入目录下文件按照一定的标准逐个进行逻辑切片，形成切片规划。默认情况下，Split size = Block size。每一个切片由一个 MapTask 处理。（getSplits）
- **第二阶段**是对切片中的数据按照一定的规则解析成<key,value>对。默认规则是把每一行文本内容解析成键值对。key 是每一行的起始位置(单位是字节)，value 是本行的文本内容。（TextInputFormat）
- **第三阶段**是调用 Mapper 类中的 map 方法。上阶段中每解析出来的一个<k,v>，调用一次 map 方法。每次调用 map 方法会输出零个或多个键值对。
- **第四阶段**是按照一定的规则对第三阶段输出的键值对进行分区。默认是只有一个区。分区的数量就是 Reducer 任务运行的数量。默认只有一个 Reducer 任务。
- **第五阶段**是对每个分区中的键值对进行排序。首先，按照键进行排序，对于键相同的键值对，按照值进行排序。比如三个键值对<2,2>、<1,3>、<2,1>，键和值分别是整数。那么排序后的结果是<1,3>、<2,1>、<2,2>。如果有第六阶段，那么进入第六阶段；如果没有，直接输出到文件中。
- **第六阶段**是对数据进行局部聚合处理，也就是 combiner 处理。键相等的键值对会调用一次 reduce 方法。经过这一阶段，数据量会减少。**本阶段默认是没有的。**

2.2. Reducer 任务执行过程详解

- **第一阶段**是 Reducer 任务会主动从 Mapper 任务复制其输出的键值对。Mapper 任务可能会有很多，因此 Reducer 会复制多个 Mapper 的输出。
- **第二阶段**是把复制到 Reducer 本地数据，全部进行合并，即把分散的数据合并成一个大的数据。再对合并后的数据排序。
- **第三阶段**是对排序后的键值对调用 reduce 方法。键相等的键值对调用一次 reduce 方法，每次调用会产生零个或者多个键值对。最后把这些输出的键值对写入到 HDFS 文件中。

在整个 MapReduce 程序的开发过程中，我们最大的工作量是覆盖 map 函数和覆盖 reduce 函数。

三、 MapReduce 的序列化

1. 概述

序列化（Serialization）是指把结构化对象转化为字节流。

反序列化(Deserialization)是序列化的逆过程。把字节流转为结构化对象。

当要在进程间传递对象或持久化对象的时候，就需要序列化对象成字节流，反之当要将接收到或从磁盘读取的字节流转换为对象，就要进行反序列化。

Java 的序列化（Serializable）是一个重量级序列化框架，一个对象被序列化后，会附带很多额外的信息（各种校验信息，header，继承体系…），不便于在网络中高效传输；所以，hadoop 自己开发了一套序列化机制（Writable），精简，高效。不用像 java 对象类一样传输多层的父子关系，需要哪个属性就传输哪个属性值，大大的减少网络传输的开销。

Writable 是 Hadoop 的序列化格式,hadoop 定义了这样一个 Writable 接口。一个类要支持可序列化只需实现这个接口即可。

```
1. public interface Writable {
2.   void write(DataOutput out) throws IOException;
3.   void readFields(DataInput in) throws IOException;
4. }
```

2. Writable 序列化接口

如需要将自定义的 bean 放在 key 中传输，则还需要实现 comparable 接口，因为 mapreduce 框中的 shuffle 过程一定会对 key 进行排序,此时，自定义的 bean 实现的接口应该是：

public class FlowBean implements WritableComparable<FlowBean>

需要自己实现的方法是：

```
/**
 * 反序列化的方法，反序列化时，从流中读取到的各个字段的顺序应该与序列化时写出去的顺序保持一致
 */
@Override
```

```
public void readFields(DataInput in) throws IOException {

        upflow = in.readLong();
        dflow = in.readLong();
        sumflow = in.readLong();

}
/**
 * 序列化的方法
 */
@Override
public void write(DataOutput out) throws IOException {

        out.writeLong(upflow);
        out.writeLong(dflow);
        out.writeLong(sumflow);
}

@Override
public int compareTo(FlowBean o) {

        //实现按照 sumflow 的大小倒序排序
        return sumflow>o.getSumflow()?-1:1;
}
```

compareTo 方法用于将当前对象与方法的参数进行比较。

如果指定的数与参数相等返回 0。

如果指定的数小于参数返回 -1。

如果指定的数大于参数返回 1。

例如：o1.compareTo(o2);

返回正数的话，当前对象（调用 compareTo 方法的对象 o1）要排在比较对象（compareTo 传参对象 o2）后面，返回负数的话，放在前面。

四、 Mapreduce 的排序初步

1. 需求

在得出统计每一个用户（手机号）所耗费的总上行流量、下行流量，总流量结果的基础之上再加一个需求：将统计结果按照总流量倒序排序。

2. 分析

基本思路：

实现自定义的 bean 来封装流量信息，并将 bean 作为 map 输出的 key 来传输

MR 程序在处理数据的过程中会对数据排序(map 输出的 kv 对传输到 reduce 之前，会排序)，排序的依据是 map 输出的 key。所以，我们如果要实现自己需要的排序规则，则可以考虑将排序因素放到 key 中，让 key 实现接口：WritableComparable，然后重写 key 的 compareTo 方法。

3. 实现

自定义的 bean

```
public class FlowBean implements WritableComparable<FlowBean>{
        private long upFlow;
        private long downFlow;
        private long sumFlow;

        //这里反序列的时候会用到
        public FlowBean() {
        }

        public FlowBean(long upFlow, long downFlow, long sumFlow) {
            this.upFlow = upFlow;
            this.downFlow = downFlow;
            this.sumFlow = sumFlow;
        }

        public FlowBean(long upFlow, long downFlow) {
            this.upFlow = upFlow;
            this.downFlow = downFlow;
            this.sumFlow = upFlow+downFlow;
```

```
}

public void set(long upFlow, long downFlow) {
	this.upFlow = upFlow;
	this.downFlow = downFlow;
	this.sumFlow = upFlow+downFlow;
}

@Override
public String toString() {
	return upFlow+"\t"+downFlow+"\t"+sumFlow;
}

/这里是序列化方法
@Override
public void write(DataOutput out) throws IOException {
		out.writeLong(upFlow);
		out.writeLong(downFlow);
		out.writeLong(sumFlow);
}

/这里是反序列化方法

@Override
public void readFields(DataInput in) throws IOException {
	//注意反序列化的顺序跟序列化的顺序一致
    this.upFlow = in.readLong();
    this.downFlow = in.readLong();
    this.sumFlow = in.readLong();

}

//这里进行 bean 的自定义比较大小
@Override
public int compareTo(FlowBean o) {
	//实现按照 sumflow 的大小倒序排序
	return this.sumFlow>o.getSumFlow()?-1:1;
}
}
```

```
public class FlowSumMapper extends Mapper<LongWritable, Text, Text, FlowBean>{
	Text k = new Text();
	FlowBean v = new FlowBean();

	@Override
	protected void map(LongWritable key, Text value,Context context)
			throws IOException, InterruptedException {

			String line = value.toString();
			String[] fields = line.split("\t");
			String phoneNum = fields[1];
			long upFlow = Long.parseLong(fields[fields.length-3]);
			long downFlow = Long.parseLong(fields[fields.length-2]);
			k.set(phoneNum);
			v.set(upFlow, downFlow);

			context.write(k, v);
	}
}

public class FlowSumReducer extends Reducer<Text, FlowBean, Text, FlowBean> {
	FlowBean v = new FlowBean();

	@Override
	protected void reduce(Text key, Iterable<FlowBean> values, Context context)
			throws IOException, InterruptedException {

		long upFlowCount = 0;
		long downFlowCount = 0;
		for (FlowBean bean : values) {
			upFlowCount += bean.getUpFlow();
			downFlowCount += bean.getDownFlow();
		}
		v.set(upFlowCount, downFlowCount);
		context.write(key, v);
	}

}
```

五、 Mapreduce 的分区—Partitioner

1. 需求

将流量汇总统计结果按照手机归属地不同省份输出到不同文件中。

2. 分析

Mapreduce 中会将 map 输出的 kv 对，按照相同 key 分组，然后分发给不同的 reducetask。

默认的分发规则为：根据 key 的 hashcode%reducetask 数来分发

所以：如果要按照我们自己的需求进行分组，则需要改写数据分发（分组）组件 Partitioner，自定义一个 CustomPartitioner 继承抽象类：Partitioner，然后在 job 对象中，设置自定义 partitioner：job.setPartitionerClass(CustomPartitioner.class

3. 实现

```
public class ProvincePartitioner extends Partitioner<Text, FlowBean>
    public static HashMap<String, Integer>    provinceMap = new HashMap<String, Integer>();

    static{
        provinceMap.put("134", 0);
        provinceMap.put("135", 1);
        provinceMap.put("136", 2);
        provinceMap.put("137", 3);
        provinceMap.put("138", 4);
    }

    @Override
    public int getPartition(Text key, FlowBean value, int numPartitions) {
        Integer code = provinceMap.get(key.toString().substring(0, 3));

        if (code != null) {
            return code;
        }
```

```
            return 5;
        }

}
```

```
public class FlowSumProvince {

 public static class FlowSumProvinceMapper extends Mapper<LongWritable, Text, Text, FlowBean>{

        Text k = new Text();
        FlowBean    v = new FlowBean();

        @Override
        protected void map(LongWritable key, Text value,Context context)
                throws IOException, InterruptedException {
                //拿取一行文本转为 String
                String line = value.toString();
                //按照分隔符\t 进行分割
                String[] fileds = line.split("\t");
                //获取用户手机号
                String phoneNum = fileds[1];

                long upFlow = Long.parseLong(fileds[fileds.length-3]);
                long downFlow = Long.parseLong(fileds[fileds.length-2]);

                k.set(phoneNum);
                v.set(upFlow, downFlow);

                context.write(k,v);

            }

        }
```

```
    public static class FlowSumProvinceReducer extends Reducer<Text, FlowBean, Text, FlowBean>{

        FlowBean    v    = new FlowBean();

        @Override
        protected void reduce(Text key, Iterable<FlowBean> flowBeans,Context context) throws IOException, InterruptedException {

            long upFlowCount = 0;
            long downFlowCount = 0;

            for (FlowBean flowBean : flowBeans) {

                upFlowCount += flowBean.getUpFlow();

                downFlowCount += flowBean.getDownFlow();

            }
            v.set(upFlowCount, downFlowCount);

            context.write(key, v);
    }


    public static void main(String[] args) throws Exception{


        Configuration conf = new Configuration();
        conf.set("mapreduce.framework.name", "local");

        Job job = Job.getInstance(conf);

        //指定我这个 job 所在的 jar 包位置
        job.setJarByClass(FlowSumProvince.class);

        //指定我们使用的 Mapper 是那个类  reducer 是哪个类
        job.setMapperClass(FlowSumProvinceMapper.class);
```

```
        job.setReducerClass(FlowSumProvinceReducer.class);
//          job.setCombinerClass(FlowSumProvinceReducer.class);

        // 设置我们的业务逻辑 Mapper 类的输出 key 和 value 的数据类型
        job.setMapOutputKeyClass(Text.class);
        job.setMapOutputValueClass(FlowBean.class);

        // 设置我们的业务逻辑 Reducer 类的输出 key 和 value 的数据类型
        job.setOutputKeyClass(Text.class);
        job.setOutputValueClass(FlowBean.class);



        //这里设置运行 reduceTask 的个数
        //getPartition 返回的分区个数 = NumReduceTasks    正常执行
        //getPartition 返回的分区个数 > NumReduceTasks    报错：Illegal partition
        //getPartition 返回的分区个数 < NumReduceTasks    可以执行 ，多出空白文件
        job.setNumReduceTasks(10);



        //这里指定使用我们自定义的分区组件
        job.setPartitionerClass(ProvincePartitioner.class);



        FileInputFormat.setInputPaths(job, new Path("D:\\flowsum\\input"));
        // 指定处理完成之后的结果所保存的位置
        FileOutputFormat.setOutputPath(job, new Path("D:\\flowsum\\outputProvince"));

        boolean res = job.waitForCompletion(true);
        System.exit(res ? 0 : 1);

    }

 }
}
```

六、 Mapreduce 的 combiner

每一个 map 都可能会产生大量的本地输出，Combiner 的作用就是对 map 端的输出先做一次合并，以减少在 map 和 reduce 节点之间的数据传输量，以提高网络 IO 性能，是 MapReduce 的一种优化手段之一。

- combiner 是 MR 程序中 Mapper 和 Reducer 之外的一种组件
- combiner 组件的父类就是 Reducer
- combiner 和 reducer 的区别在于运行的位置：

  Combiner 是在每一个 maptask 所在的节点运行

  Reducer 是接收全局所有 Mapper 的输出结果；
- combiner 的意义就是对每一个 maptask 的输出进行局部汇总，以减小网络传输量
- 具体实现步骤：

  1、自定义一个 combiner 继承 Reducer，重写 reduce 方法

  2、在 job 中设置： job.setCombinerClass(CustomCombiner.class)
- combiner 能够应用的前提是不能影响最终的业务逻辑，而且，combiner 的输出 kv 应该跟 reducer 的输入 kv 类型要对应起来

七、 Apache Flume

1. 概述

Flume 是 Cloudera 提供的一个高可用的，高可靠的，分布式的海量日志采集、聚合和传输的软件。

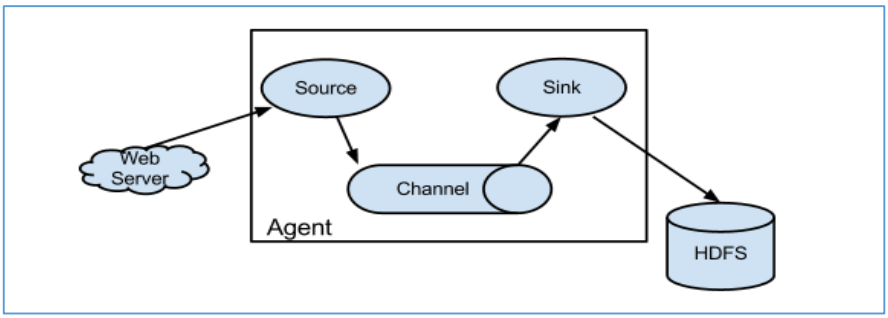
Flume 的核心是把数据从数据源(source)收集过来，再将收集到的数据送到指定的目的地(sink)。为了保证输送的过程一定成功，在送到目的地(sink)之前，会先缓存数据(channel)，待数据真正到达目的地(sink)后，flume 在删除自己缓存的数据。

Flume 支持定制各类数据发送方，用于收集各类型数据；同时，Flume 支持定制各种数据接受方，用于最终存储数据。一般的采集需求，通过对 flume 的简单配置即可实现。针对特殊场景也具备良好的自定义扩展能力。因此，flume 可以适用于大部分的日常数据采集场景。

当前 Flume 有两个版本。Flume 0.9X 版本的统称 Flume OG（original generation），Flume1.X 版本的统称 Flume NG（next generation）。由于 Flume NG 经过核心组件、核心配置以及代码架构重构，与 Flume OG 有很大不同，使用时请注意区分。改动的另一原因是将 Flume 纳入 apache 旗下，Cloudera Flume 改名为 Apache Flume。

2. 运行机制

Flume 系统中核心的角色是 agent，agent 本身是一个 Java 进程，一般运行在日志收集节点。



每一个 agent 相当于一个数据传递员，内部有三个组件：

Source：采集源，用于跟数据源对接，以获取数据；

Sink：下沉地，采集数据的传送目的，用于往下一级 agent 传递数据或者往最终存储系统传递数据；

Channel：agent 内部的数据传输通道，用于从 source 将数据传递到 sink；

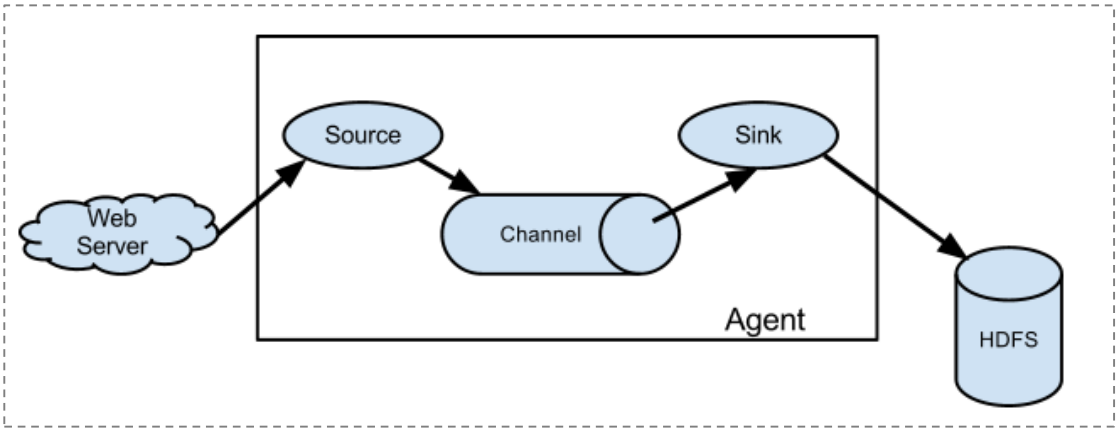
在整个数据的传输的过程中，流动的是 event，它是 Flume 内部数据传输的最基本单元。event 将传输的数据进行封装。如果是文本文件，通常是一行记录，event 也是事务的基本单位。event 从 source，流向 channel，再到 sink，本身为一个字节数组，并可携带 headers(头信息)信息。event 代表着一个数据的最小完整单元，从外部数据源来，向外部的目的地去。

一个完整的 event 包括：event headers、event body、event 信息，其中 event 信息就是 flume 收集到的日记记录。

3. Flume 采集系统结构图

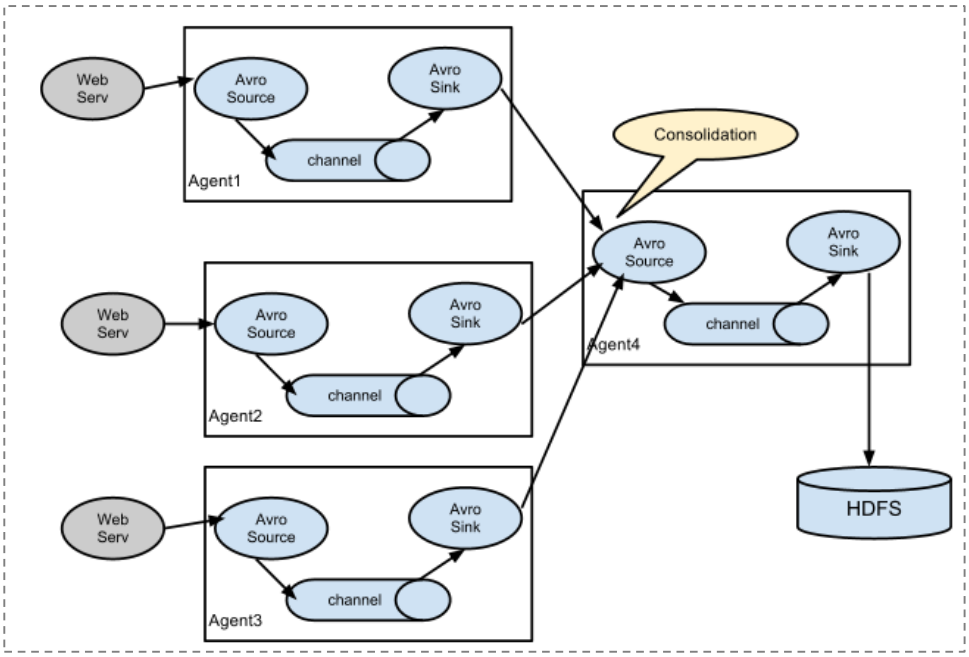
3.1. 简单结构

单个 agent 采集数据



3.2. 复杂结构

多级 agent 之间串联



八、 Flume 安装部署

- Flume 的安装非常简单

上传安装包到数据源所在节点上

然后解压 tar -zxvf apache-flume-1.6.0-bin.tar.gz

然后进入 flume 的目录，修改 conf 下的 flume-env.sh，在里面配置 JAVA_HOME

- 根据数据采集需求**配置采集方案**，描述在配置文件中(文件名可任意自定义)
- **指定采集方案配置文件**，在相应的节点上启动 flume agent

先用一个最简单的例子来测试一下程序环境是否正常

1、先在 flume 的 conf 目录下新建一个文件

vi netcat-logger.conf

```
# 定义这个 agent 中各组件的名字
a1.sources = r1
a1.sinks = k1
a1.channels = c1

# 描述和配置 source 组件：r1
a1.sources.r1.type = netcat
a1.sources.r1.bind = localhost
a1.sources.r1.port = 44444

# 描述和配置 sink 组件：k1
a1.sinks.k1.type = logger

# 描述和配置 channel 组件，此处使用是内存缓存的方式
a1.channels.c1.type = memory
a1.channels.c1.capacity = 1000
a1.channels.c1.transactionCapacity = 100

# 描述和配置 source  channel   sink 之间的连接关系
a1.sources.r1.channels = c1
a1.sinks.k1.channel = c1
```

2、启动 agent 去采集数据

```
bin/flume-ng agent -c conf -f conf/netcat-logger.conf -n a1    -Dflume.root.logger=INFO,console
```

-c conf　　指定 flume 自身的配置文件所在目录

-f conf/netcat-logger.con　指定我们所描述的采集方案

-n a1　指定我们这个 agent 的名字

3、测试

先要往 agent 采集监听的端口上发送数据，让 agent 有数据可采。

随便在一个能跟 agent 节点联网的机器上：

telnet anget-hostname　port　　（telnet localhost 44444）

九、 Flume 简单案例

1. 采集目录到 HDFS

采集需求：**服务器的某特定目录下，会不断产生新的文件，每当有新文件出现，就需要把文件采集到 HDFS 中去**

根据需求，首先定义以下 3 大要素

- 采集源，即 source——监控文件目录 ： **spooldir**
- 下沉目标，即 sink——HDFS 文件系统 ： **hdfs sink**
- source 和 sink 之间的传递通道——channel，可用 file channel 也可以用内存 channel

配置文件编写：

```
# Name the components on this agent
a1.sources = r1
a1.sinks = k1
a1.channels = c1

# Describe/configure the source
##注意：不能往监控目中重复丢同名文件
a1.sources.r1.type = spooldir
a1.sources.r1.spoolDir = /root/logs
a1.sources.r1.fileHeader = true

# Describe the sink
a1.sinks.k1.type = hdfs
a1.sinks.k1.hdfs.path = /flume/events/%y-%m-%d/%H%M/
a1.sinks.k1.hdfs.filePrefix = events-
a1.sinks.k1.hdfs.round = true
a1.sinks.k1.hdfs.roundValue = 10
a1.sinks.k1.hdfs.roundUnit = minute
a1.sinks.k1.hdfs.rollInterval = 3
a1.sinks.k1.hdfs.rollSize = 20
a1.sinks.k1.hdfs.rollCount = 5
a1.sinks.k1.hdfs.batchSize = 1
a1.sinks.k1.hdfs.useLocalTimeStamp = true
```

```
#生成的文件类型，默认是 Sequencefile，可用 DataStream，则为普通文本
a1.sinks.k1.hdfs.fileType = DataStream

# Use a channel which buffers events in memory
a1.channels.c1.type = memory
a1.channels.c1.capacity = 1000
a1.channels.c1.transactionCapacity = 100

# Bind the source and sink to the channel
a1.sources.r1.channels = c1
a1.sinks.k1.channel = c1
```

Channel 参数解释：

capacity：默认该通道中最大的可以存储的 event 数量

trasactionCapacity：每次最大可以从 source 中拿到或者送到 sink 中的 event 数量

2. 采集文件到 HDFS

采集需求：**比如业务系统使用 log4j 生成的日志，日志内容不断增加，需要把追加到日志文件中的数据实时采集到 hdfs**

根据需求，首先定义以下 3 大要素

- 采集源，即 source——监控文件内容更新 ： exec ‘tail -F file’
- 下沉目标，即 sink——HDFS 文件系统 ： hdfs sink
- Source 和 sink 之间的传递通道——channel，可用 file channel 也可以用内存 channel

配置文件编写：

```
# Name the components on this agent
a1.sources = r1
a1.sinks = k1
a1.channels = c1
```

```
# Describe/configure the source
a1.sources.r1.type = exec
a1.sources.r1.command = tail -F /root/logs/test.log
a1.sources.r1.channels = c1

# Describe the sink
a1.sinks.k1.type = hdfs
a1.sinks.k1.hdfs.path = /flume/tailout/%y-%m-%d/%H%M/
a1.sinks.k1.hdfs.filePrefix = events-
a1.sinks.k1.hdfs.round = true
a1.sinks.k1.hdfs.roundValue = 10
a1.sinks.k1.hdfs.roundUnit = minute
a1.sinks.k1.hdfs.rollInterval = 3
a1.sinks.k1.hdfs.rollSize = 20
a1.sinks.k1.hdfs.rollCount = 5
a1.sinks.k1.hdfs.batchSize = 1
a1.sinks.k1.hdfs.useLocalTimeStamp = true
#生成的文件类型，默认是 Sequencefile，可用 DataStream，则为普通文本
a1.sinks.k1.hdfs.fileType = DataStream

# Use a channel which buffers events in memory
a1.channels.c1.type = memory
a1.channels.c1.capacity = 1000
a1.channels.c1.transactionCapacity = 100

# Bind the source and sink to the channel
a1.sources.r1.channels = c1
a1.sinks.k1.channel = c1
```

参数解析：

- rollInterval

默认值：30

hdfs sink 间隔多长将临时文件滚动成最终目标文件，单位：秒；

如果设置成 0，则表示不根据时间来滚动文件；

注：滚动（roll）指的是，hdfs sink 将临时文件重命名成最终目标文件，并新打开一个临时文件来写入数据；

- rollSize

默认值：1024

当临时文件达到该大小（单位：bytes）时，滚动成目标文件；

如果设置成 0，则表示不根据临时文件大小来滚动文件；

- rollCount

默认值：10

当 events 数据达到该数量时候，将临时文件滚动成目标文件；

如果设置成 0，则表示不根据 events 数据来滚动文件；

- round

默认值：false

是否启用时间上的“舍弃”，这里的“舍弃”，类似于“四舍五入”。

- roundValue

默认值：1

时间上进行“舍弃”的值；

- roundUnit

默认值：seconds

时间上进行“舍弃”的单位，包含：second,minute,hour

示例：

a1.sinks.k1.hdfs.path = /flume/events/%y-%m-%d/%H%M/%S

a1.sinks.k1.hdfs.round = true

a1.sinks.k1.hdfs.roundValue = 10

a1.sinks.k1.hdfs.roundUnit = minute

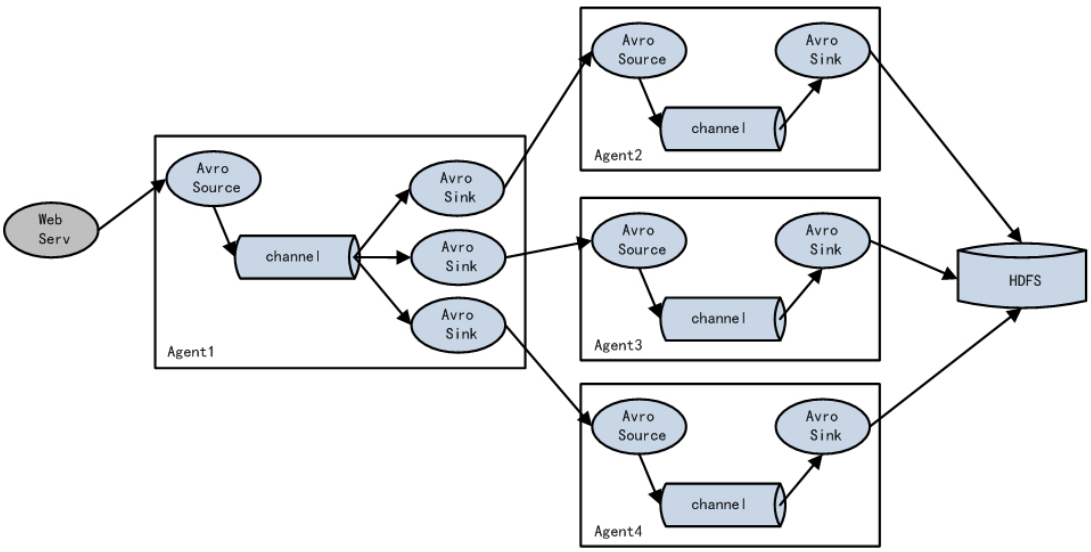
当时间为 2015-10-16 17:38:59 时候，hdfs.path 依然会被解析为：

/flume/events/20151016/17:30/00

因为设置的是舍弃 10 分钟内的时间，因此，该目录每 10 分钟新生成一个。

十、 Flume 的 load-balance、failover

负载均衡是用于解决一台机器(一个进程)无法解决所有请求而产生的一种算法。Load balancing Sink Processor 能够实现 load balance 功能，如下图 Agent1 是一个路由节点，负责将 Channel 暂存的 Event 均衡到对应的多个 Sink 组件上，而每个 Sink 组件分别连接到一个独立的 Agent 上，示例配置，如下所示：




```
a1.sinkgroups = g1
a1.sinkgroups.g1.sinks = k1 k2 k3
a1.sinkgroups.g1.processor.type = load_balance
a1.sinkgroups.g1.processor.backoff = true   #如果开启，则将失败的 sink 放入黑名单
a1.sinkgroups.g1.processor.selector = round_robin   # 另外还支持 random
a1.sinkgroups.g1.processor.selector.maxTimeOut=10000 #在黑名单放置的超时时间，超时结束时，若仍然无法接收，则超时时间呈指数增长
```

Failover Sink Processor 能够实现 failover 功能，具体流程类似 load balance，但是内部处理机制与 load balance 完全不同。

Failover Sink Processor 维护一个优先级 Sink 组件列表，只要有一个 Sink 组件可用，Event 就被传递到下一个组件。故障转移机制的作用是将失败的 Sink 降级到一个池，在这些池中它们被分配一个冷却时间，随着故障的连续，在重试之前冷却时间增加。一旦 Sink 成功发送一个事件，它将恢复到活动池。 Sink 具有与之相关的优先级，数量越大，优先级越高。

例如，具有优先级为 100 的 sink 在优先级为 80 的 Sink 之前被激活。如果在发送事件时汇聚失败，则接下来将尝试下一个具有最高优先级的 Sink 发送事件。如果没有指定优先级，则根据在配置中指定 Sink 的顺序来确定优先级。

示例配置如下所示：

```
a1.sinkgroups = g1
a1.sinkgroups.g1.sinks = k1 k2 k3
a1.sinkgroups.g1.processor.type = failover
a1.sinkgroups.g1.processor.priority.k1 = 5    #优先级值, 绝对值越大表示优先级越高
a1.sinkgroups.g1.processor.priority.k2 = 7
a1.sinkgroups.g1.processor.priority.k3 = 6
a1.sinkgroups.g1.processor.maxpenalty = 20000    #失败的 Sink 的最大回退期（millis）
```

十一、 Flume 实战案例

1. 日志的采集和汇总

1.1. 案例场景

A、B 两台日志服务机器实时生产日志主要类型为 access.log、nginx.log、web.log

现在要求：

把 A、B 机器中的 access.log、nginx.log、web.log 采集汇总到 C 机器上然后统一收集到 hdfs 中。

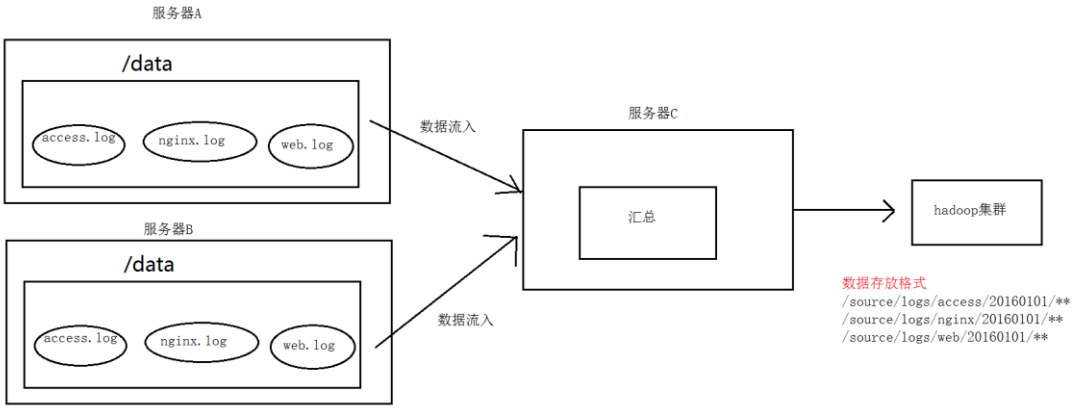
但是在 hdfs 中要求的目录为：

/source/logs/access/20160101/**

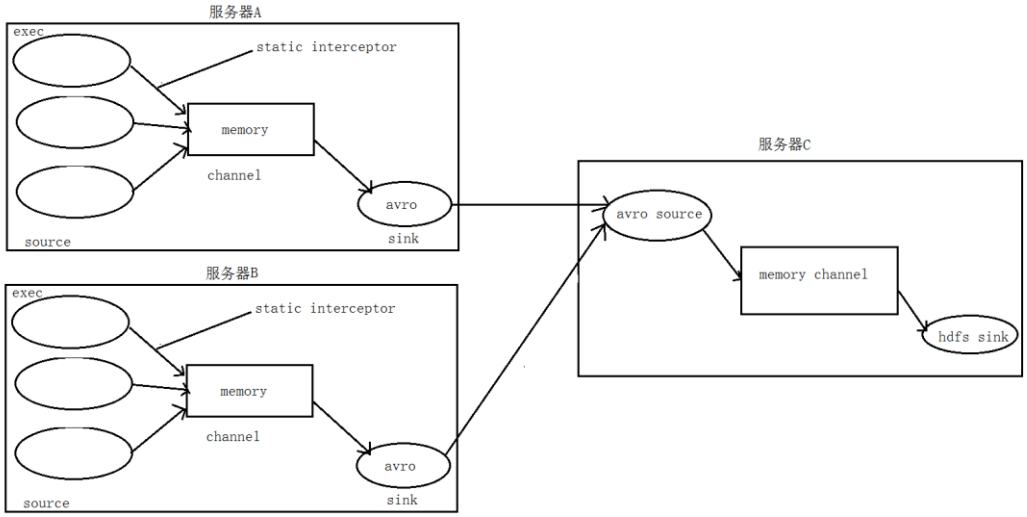
/source/logs/nginx/20160101/**

/source/logs/web/20160101/**

1.2. 场景分析



1.3. 数据流程处理分析



1.4. 功能实现

① 在服务器 A 和服务器 B 上
创建配置文件 exec_source_avro_sink.conf

```
# Name the components on this agent
a1.sources = r1 r2 r3
a1.sinks = k1
a1.channels = c1

# Describe/configure the source
a1.sources.r1.type = exec
a1.sources.r1.command = tail -F /root/data/access.log
a1.sources.r1.interceptors = i1
a1.sources.r1.interceptors.i1.type = static
##  static 拦截器的功能就是往采集到的数据的 header 中插入自
##  己定义的 key-value 对
a1.sources.r1.interceptors.i1.key = type
a1.sources.r1.interceptors.i1.value = access

a1.sources.r2.type = exec
a1.sources.r2.command = tail -F /root/data/nginx.log
a1.sources.r2.interceptors = i2
a1.sources.r2.interceptors.i2.type = static
a1.sources.r2.interceptors.i2.key = type
```

```
a1.sources.r2.interceptors.i2.value = nginx

a1.sources.r3.type = exec
a1.sources.r3.command = tail -F /root/data/web.log
a1.sources.r3.interceptors = i3
a1.sources.r3.interceptors.i3.type = static
a1.sources.r3.interceptors.i3.key = type
a1.sources.r3.interceptors.i3.value = web

# Describe the sink
a1.sinks.k1.type = avro
a1.sinks.k1.hostname = 192.168.200.101
a1.sinks.k1.port = 41414

# Use a channel which buffers events in memory
a1.channels.c1.type = memory
a1.channels.c1.capacity = 20000
a1.channels.c1.transactionCapacity = 10000

# Bind the source and sink to the channel
a1.sources.r1.channels = c1
a1.sources.r2.channels = c1
a1.sources.r3.channels = c1
a1.sinks.k1.channel = c1
```

② 在服务器 c 上创建配置文件avro_source_hdfs_sink.conf 文件内容为

```
#定义 agent 名， source、channel、sink 的名称
a1.sources = r1
a1.sinks = k1
a1.channels = c1

#定义 source
a1.sources.r1.type = avro
a1.sources.r1.bind = mini2
a1.sources.r1.port =41414

#添加时间拦截器
```

```
a1.sources.r1.interceptors = i1
a1.sources.r1.interceptors.i1.type =
org.apache.flume.interceptor.TimestampInterceptor$Builder

#定义 channels
a1.channels.c1.type = memory
a1.channels.c1.capacity = 20000
a1.channels.c1.transactionCapacity = 10000

#定义 sink
a1.sinks.k1.type = hdfs
a1.sinks.k1.hdfs.path=hdfs://192.168.200.101:9000/source/logs/%{ty
pe}/%Y%m%d
a1.sinks.k1.hdfs.filePrefix =events
a1.sinks.k1.hdfs.fileType = DataStream
a1.sinks.k1.hdfs.writeFormat = Text
#时间类型
a1.sinks.k1.hdfs.useLocalTimeStamp = true
#生成的文件不按条数生成
a1.sinks.k1.hdfs.rollCount = 0
#生成的文件按时间生成
a1.sinks.k1.hdfs.rollInterval = 30
#生成的文件按大小生成
a1.sinks.k1.hdfs.rollSize    = 10485760
#批量写入 hdfs 的个数
a1.sinks.k1.hdfs.batchSize = 10000
flume 操作 hdfs 的线程数（包括新建，写入等）
a1.sinks.k1.hdfs.threadsPoolSize=10
#操作 hdfs 超时时间
a1.sinks.k1.hdfs.callTimeout=30000

#组装 source、channel、sink
a1.sources.r1.channels = c1
a1.sinks.k1.channel = c1
```

③ 配置完成之后，在服务器 A 和 B 上的/root/data 有数据文件 access.log、nginx.log、web.log。先启动服务器 C 上的 flume，启动命令

在 flume 安装目录下执行 ：

bin/flume-ng agent -c conf -f conf/avro_source_hdfs_sink.conf -name a1 -

Dflume.root.logger=DEBUG,console

然后在启动服务器上的 A 和 B，启动命令

在 flume 安装目录下执行 ：

bin/flume-ng agent -c conf -f conf/exec_source_avro_sink.conf -name a1 -Dflume.root.logger=DEBUG,console

2. Flume 自定义拦截器（了解）

2.1. 案例背景介绍

Flume 是 Cloudera 提供的一个高可用的，高可靠的，分布式的海量日志采集、聚合和传输的系统，Flume 支持在日志系统中定制各类数据发送方，用于收集数据；同时，Flume 提供对数据进行简单处理，并写到各种数据接受方（可定制）的能力。**Flume 有各种自带的拦截器，比如：TimestampInterceptor、HostInterceptor、RegexExtractorInterceptor 等，通过使用不同的拦截器，实现不同的功能。**但是以上的这些拦截器，不能改变原有日志数据的内容或者对日志信息添加一定的处理逻辑，当一条日志信息有几十个甚至上百个字段的时候，在传统的 Flume 处理下，收集到的日志还是会有对应这么多的字段，也不能对你想要的字段进行对应的处理。

2.2. 自定义拦截器

根据实际业务的需求，为了更好的满足数据在应用层的处理，通过自定义 Flume 拦截器，过滤掉不需要的字段，并对指定字段加密处理，将源数据进行预处理。减少了数据的传输量，降低了存储的开销。

2.3. 功能实现

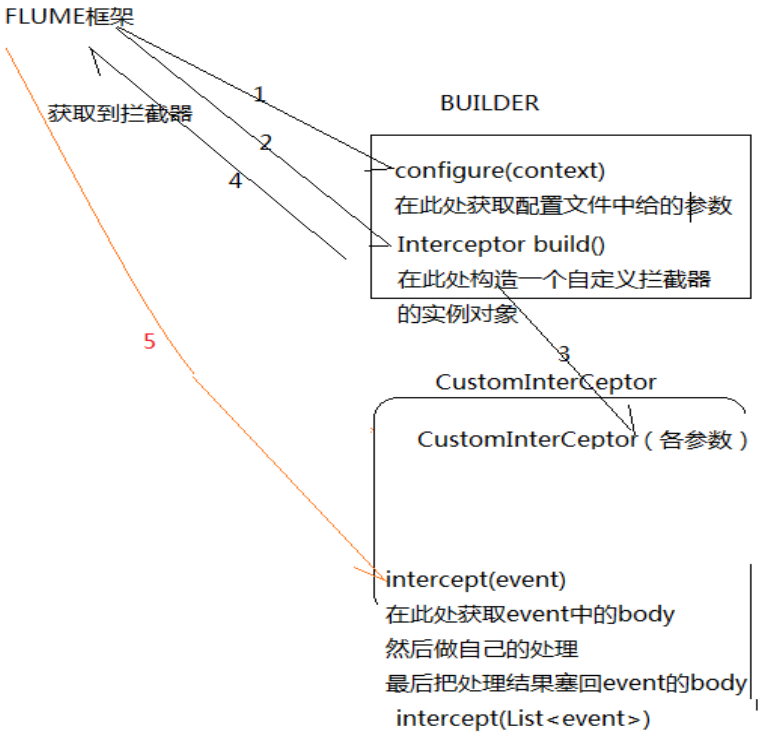
本技术方案核心包括二部分：

- **编写 java 代码，自定义拦截器**

内容包括：

1. 定义一个类 CustomParameterInterceptor 实现 Interceptor 接口。
2. 在 CustomParameterInterceptor 类中定义变量，这些变量是需要到 Flume 的配置文件中进行配置使用的。每一行字段间的分隔符(fields_separator)、通过分隔符分隔后，所需要列字段的下标（indexs)、多个下标使用的分隔符（indexs_separator)、多个下标使用的分隔符（indexs_separator)。

3. 添加 CustomParameterInterceptor 的有参构造方法。并对相应的变量进行处理。将配置文件中传过来的 unicode 编码进行转换为字符串。
4. 写具体的要处理的逻辑 intercept()方法，一个是单个处理的，一个是批量处理。
5. 接口中定义了一个内部接口 Builder，在 configure 方法中，进行一些参数配置。并给出，在 flume 的 conf 中没配置一些参数时，给出其默认值。通过其 builder 方法，返回一个 CustomParameterInterceptor 对象。
6. 定义一个静态类，类中封装 MD5 加密方法



7. 通过以上步骤，自定义拦截器的代码开发已完成，然后打包成 jar， 放到 Flume 的根目录下的 lib 中

- **修改 Flume 的配置信息**

新增配置文件 spool-interceptor-hdfs.conf，内容为：

```
a1.channels = c1
a1.sources = r1
a1.sinks = s1

#channel
```

```
a1.channels.c1.type = memory
a1.channels.c1.capacity=100000
a1.channels.c1.transactionCapacity=50000

#source
a1.sources.r1.channels = c1
a1.sources.r1.type = spooldir
a1.sources.r1.spoolDir = /root/data/
a1.sources.r1.batchSize= 50
a1.sources.r1.inputCharset = UTF-8

a1.sources.r1.interceptors =i1 i2
a1.sources.r1.interceptors.i1.type
=cn.itcast.interceptor.CustomParameterInterceptor$Builder
a1.sources.r1.interceptors.i1.fields_separator=\\u0009
a1.sources.r1.interceptors.i1.indexs =0,1,3,5,6
a1.sources.r1.interceptors.i1.indexs_separator =\\u002c
a1.sources.r1.interceptors.i1.encrypted_field_index =0

a1.sources.r1.interceptors.i2.type                                    =
org.apache.flume.interceptor.TimestampInterceptor$Builder


#sink
a1.sinks.s1.channel = c1
a1.sinks.s1.type = hdfs
a1.sinks.s1.hdfs.path =hdfs://192.168.200.101:9000/flume/%Y%m%d
a1.sinks.s1.hdfs.filePrefix = event
a1.sinks.s1.hdfs.fileSuffix = .log
a1.sinks.s1.hdfs.rollSize = 10485760
a1.sinks.s1.hdfs.rollInterval =20
a1.sinks.s1.hdfs.rollCount = 0
a1.sinks.s1.hdfs.batchSize = 1500
a1.sinks.s1.hdfs.round = true
a1.sinks.s1.hdfs.roundUnit = minute
a1.sinks.s1.hdfs.threadsPoolSize = 25
a1.sinks.s1.hdfs.useLocalTimeStamp = true
a1.sinks.s1.hdfs.minBlockReplicas = 1
```

```
a1.sinks.s1.hdfs.fileType =DataStream
a1.sinks.s1.hdfs.writeFormat = Text
a1.sinks.s1.hdfs.callTimeout = 60000
a1.sinks.s1.hdfs.idleTimeout =60
```

启动：

```
bin/flume-ng agent -c conf -f conf/spool-interceptor-hdfs.conf -name
a1 -Dflume.root.logger=DEBUG,console
```