用户综合分析平台-day8笔记

复习

从业务系统采集日志信息，对日志信息进行处理

1. 如果是登录的日志信息，就应该生成评估数据对象，执行评估链生成评估报告
2. 如果是登录成功的日志信息，就应该生成登录成功数据对象，执行更新链把数据更新到历史数据中

1. 责任链设计模式的应用

   所有的功能被当作责任。把所有的责任连在一起就形成了责任链

   - 父类类型--》为了能够把所欲的功能到放在一起，在链条上，在编码的时候，能够用父类类型调取到需要的功能
   - 需要把所有的责任放在一起--》list<父类>
   - 需要有一个位置标记
   - 在每一个功能执行完成之后，都应该交给链条进行处理

2. 评估因子（指标、维度）

   基于7个维度完成评估

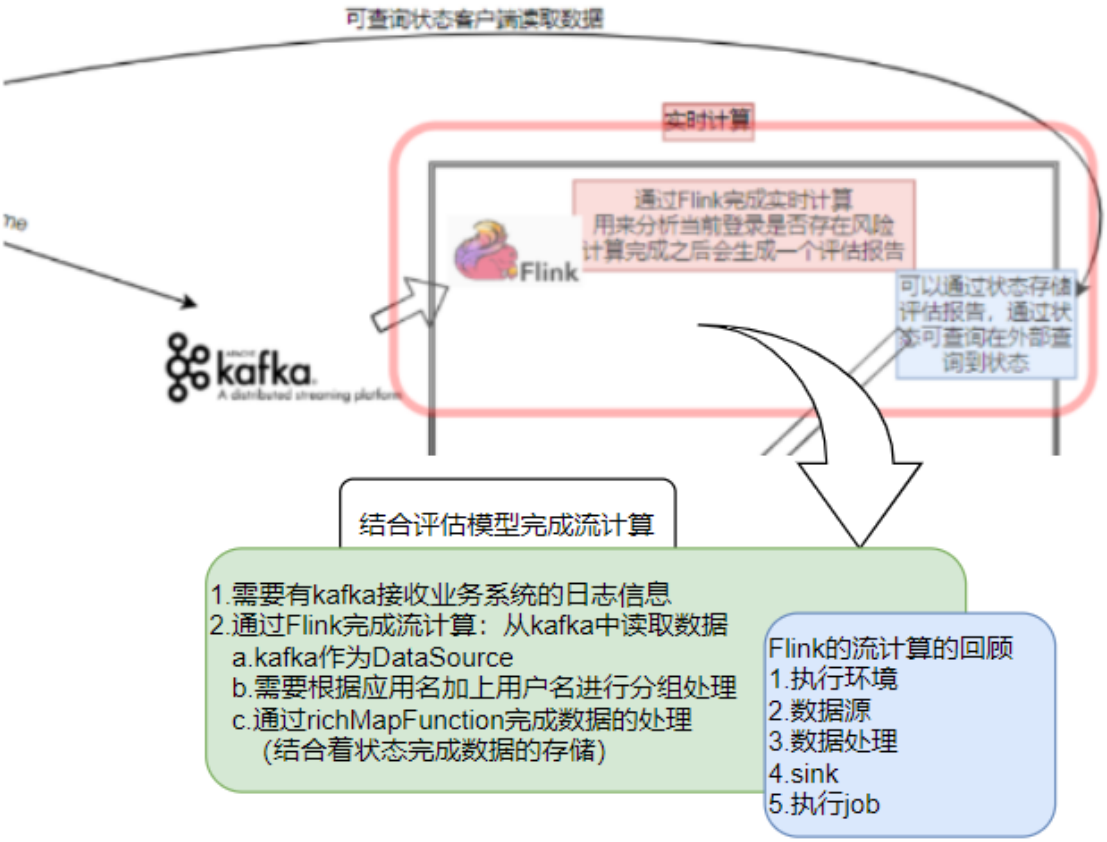
   - 登录地
   - 设备
   - 登录习惯
   - 输入特征：通过欧式距离
   - 密码相似性：余弦相似性
   - 位移速度：球面距离公式
   - 当天累计登录次数

3. 更新：把登录成功的数据更新到历史数据里面

环境准备

今天的任务就是flink实时计算：业务系统提交过来的日志信息在flink中做风险评估，然后把评估报告提交给业务系统

- Hadoop环境==》在虚拟机安装Hadoop
- Kafka环境==》需要zookeeper环境；在虚拟机中安装zookeeper，然后安装kafka
- Flink环境==》在虚拟机安装Flink
- Flume环境==》在虚拟机安装Flume
- 开发环境需要有Scala==》在Windows系统中安装Scala，在开发工具中集成Scala



创建Module-UserRiskEvaluate

```
<!--
1.需要通过scala编程语言完成Flink的代码===》需要添加scala环境
2.需要使用之前写好的风险评估模型==》以依赖的方式，把之前的module引入进来
3.需要通过Flink-scala的API完成开发==》需要添加flink流计算依赖
4.还需要flink对kafka的集成、flink对hdfs的集成===》添加对应的依赖
5.打包==》把打包插件添加进来
6.按照思路开发代码（从最基础的开始，一步步完成）
-->
```

```scala
/**
 * 用户风险评估模型job
 */
object UserRiskEvaluateJob {
  def main(args: Array[String]): Unit = {
    //1.执行环境
    //2.数据源: 暂时使用socket,方便测试
    //3.数据处理
      //3.1 把数据进行映射处理
      //3.2 根据应用名和用户名进行分组
      //3.3 通过richMapFunction实现功能
    //4.sink
    //5.执行job

  }

}
```

```scala
/**
 * 用户风险评估模型job
 */
object UserRiskEvaluateJob {
  def main(args: Array[String]): Unit = {
    //1.执行环境
    //2.数据源：暂时使用socket,方便测试
    //3.数据处理
      //3.1 把数据进行映射处理
      //3.2 根据应用名和用户名进行分组
      //3.3 通过richMapFunction实现功能
    //4.sink
    //5.执行job
    val environment: StreamExecutionEnvironment = StreamExecutionEnvironment.getExecutionEnvironment

    //需要在接下来的升级处理时，从kafka中获取数据
    val dataStream: DataStream[String] = environment.socketTextStream( hostname = "hadoop10",  port = 9999)

    //处理完成之后会有一个result
    //1.读取到的数据，一定是评估数据或者登录成功的数据-->filter算子完成
    //2.把读取到的数据log，映射成元组(应用名:用户名,log)
    //3.根据元组中的第一个元素分组 --> keyby算子完成
    //4.通过map算子，加载richMapFunction实现评估报告的生成或者历史数据的更新

//    result.print();//sink；需要在接下来的升级处理时，转换成写入到hdfs

    environment.execute( jobName = "UserRiskEvaluateJob")

  }

}
```

1. 创建Module，添加Scala环境
2. 添加依赖

```xml
<dependencies>

    <!--把写好的评估模型以依赖的方式引入到这个项目中-->
    <!--下面这个依赖到自己的评估模型的pom.xml中复制-->
    <dependency>
        <groupId>com.baizhi.evaluate</groupId>
        <artifactId>EvaluateModule</artifactId>
        <version>1.0-SNAPSHOT</version>
    </dependency>

    <!--flink stream-->
    <!-- https://mvnrepository.com/artifact/org.apache.flink/flink-streaming-scala -->
    <dependency>
        <groupId>org.apache.flink</groupId>
        <artifactId>flink-streaming-scala_2.11</artifactId>
        <version>1.10.0</version>
    </dependency>

    <!--flink集成kafka-->
    <dependency>

        <groupId>org.apache.flink</groupId>
```

```
        <artifactId>flink-connector-kafka_2.11</artifactId>
        <version>1.10.0</version>
    </dependency>
</dependencies>
```

3. 写flink最基础的代码

   从之前写的代码中复制：kafka作为数据源的代码；修改kafka相关的配置信息

```
import java.util.Properties

import com.baizhi.evaluate.util.EvaluateUtil
import org.apache.flink.api.common.serialization.SimpleStringSchema
import org.apache.flink.streaming.api.scala.StreamExecutionEnvironment
import org.apache.flink.streaming.connectors.kafka.FlinkKafkaConsumer
import org.apache.flink.streaming.api.scala._

object UserRiskEvaluate {
  def main(args: Array[String]): Unit = {
    //1.创建执行环境
    val environment = StreamExecutionEnvironment.getExecutionEnvironment

    val properties = new Properties()
    //需要修改kafka主机地址
    properties.setProperty("bootstrap.servers", "projectCentOS.baizhiedu.com:9092")

    //需要注意使用的top的名字
    var text = environment
      .addSource(new FlinkKafkaConsumer[String]("user-evaluate", new SimpleStringSchema(),
properties));


    //3.对获取到的数据进行转换
    val result = text.print()

    //4.执行job
    environment.execute("UserRiskEvaluate")
  }

}
```

4. 启动kafka，运行flink代码
5. 往kafka对应的主题中写入数据，看flink的运行结果
6. 对从kafka中读取的数据进行分析处理

```
//3.对获取到的数据进行转换
    val result = text
      .filter(EvaluateUtil.isValid(_))
      .map(log=>
(EvaluateUtil.getApplicationName(log)+":"+EvaluateUtil.getUserIdentify(log),log))//把一行数据转换成(qq:zhangsan,日志信息)
      .keyBy(t=>t._1)
      .map(new UserRiskEvaluateMapFunction())
      .filter(evaluateReport=>evaluateReport.getApplicationName!=null)//如果从kafka中读取到的数据是登录成功的数据，就不在这里打印（用另外一句话讲，是评估数据的时候，才打印）
      .print()
```

7. 写UserRiskEvaluateMapFunction
   - 创建一个scala类UserRiskEvaluateMapFunction集成RichMapFunction，重写里面的map方法
   - 需要状态管理，所以需要重写open方法==》在open方法中创建对应的状态对象

```
import java.util
import com.baizhi.evaluate.entity.{EvaluateData, EvaluateReport, HistoryData, LoginSuccessData}
import com.baizhi.evaluate.evaluate.Evaluate
import com.baizhi.evaluate.evaluate.impl._
import com.baizhi.evaluate.update.Updater
import com.baizhi.evaluate.update.impl._
import com.baizhi.evaluate.util.{EvaluateChain, EvaluateUtil, UpdaterChain}
import org.apache.flink.api.common.functions.{ReduceFunction, RichMapFunction, RuntimeContext}
import org.apache.flink.api.common.state.{ReducingState, ReducingStateDescriptor, ValueState, ValueStateDescriptor}
import org.apache.flink.configuration.Configuration
import org.apache.flink.streaming.api.scala.createTypeInformation

/**
  * 自定义RichMapFunction
  * 在这个里面完成数据的处理以及状态的管理
  *
  * 关于状态，在写代码的时候，首先创建stateDescriptor，然后由stateDescriptor创建state对象
  * 上面这一句话对应的代码，应该写在open方法里面
  *
  * 常用的state
  *   ValueState：存储单一值
  *   ListState：存储集合
  *   MapState：存储key,value
  *   ReducingState：存储单一值，该状态会通过调用用户提供的ReduceFunction，将添加的元素和历史状态自动做运算
  *   AggregatingState：同ReducingState,但是输入数据类型和输出数据类型可以不同
  */
class UserRiskEvaluateMapFunction extends RichMapFunction[(String,String),EvaluateReport]{

  //历史数据状态：用来存储历史数据的
```

```
  var historyDataState:ValueState[HistoryData]=_;

  //当天登录次数状态：设置了TTL之后就可以表示当天的登录次数
  var currentDayLoginCountState:ReducingState[Int]=_;

  //评估报告状态：用来存储评估报告
  var evaluateReportState:ValueState[EvaluateReport]=_;

  //执行map方法之前，执行一次这个open方法
  override def open(parameters: Configuration): Unit = {
    val runtimeContext: RuntimeContext = getRuntimeContext

    //创建currentDayLoginCountState；需要在这里设置TTL，设置为1
天? ? ? ? ? ? ? ? ? ? ? ? ? ? ?
    var currentDayLoginCountStateDescriptor:ReducingStateDescriptor[Int]=new
ReducingStateDescriptor[Int]("currentDayLoginCountState",new ReduceFunction[Int] {
      override def reduce(value1: Int, value2: Int): Int = {
        value1+value2
      }
    },createTypeInformation[Int])

 currentDayLoginCountState=runtimeContext.getReducingState(currentDayLoginCountStateDes
criptor);

    //创建historyDataState
    var historyDataStateDescriptor:ValueStateDescriptor[HistoryData]=new
ValueStateDescriptor[HistoryData]
("historyDataState",createTypeInformation[HistoryData]);
    historyDataState=runtimeContext.getState(historyDataStateDescriptor);

    //创建evaluateReportState
    var evaluateReportStateDescriptor:ValueStateDescriptor[EvaluateReport]=new
ValueStateDescriptor[EvaluateReport]
("evaluateReportState",createTypeInformation[EvaluateReport]);
    //评估报告需要可查询
    evaluateReportStateDescriptor.setQueryable("evaluateReportState")
    evaluateReportState=runtimeContext.getState(evaluateReportStateDescriptor);
  }

  override def map(value: (String, String)): EvaluateReport = {

    //到状态中获取历史数据
    var historyData: HistoryData = historyDataState.value()
    if(historyData==null){
      historyData=new HistoryData();
    }

    println(historyData+"**************************old")

    var evaluateReport:EvaluateReport=new EvaluateReport();
    if(EvaluateUtil.isEvaluateData(value._2)){
      //做评估操作，生成评估报告
```

```
      //评估因子
      var evaluates:util.ArrayList[Evaluate] = new util.ArrayList[Evaluate]
      evaluates.add(new AreaEvaluate)
      evaluates.add(new DeviceEvaluate)
      evaluates.add(new InputfeatureEvaluate)
      evaluates.add(new SimilarityEvaluate(0.9))
      evaluates.add(new SpeedEvaluate(750.0))
      evaluates.add(new TimeSlotEvaluate(1))
      evaluates.add(new TotalEvaluate(2))

      //解析评估数据
      val evaluateData: EvaluateData = EvaluateUtil.getEvaluateData(value._2)

      //准备一个评估报告
      evaluateReport = new EvaluateReport(evaluateData.getApplicationName,
evaluateData.getUserIdentify, evaluateData.getLoginSequence,
evaluateData.getEvaluateTime, evaluateData.getCityName, evaluateData.getGeoPoint)

      //评估链执行完毕之后，所有的评估因子都执行一次
      val evaluateChain = new EvaluateChain(evaluates)
      evaluateChain.doChain(evaluateData, historyData, evaluateReport)

      //evaluateReport=新值(生成出来的评估报告)
    }else{
      //做更新历史数据操作
      var updaters:util.ArrayList[Updater] = new util.ArrayList[Updater];
      updaters.add(new CitiesUpdates)
      updaters.add(new DeviceUpdates(3))
      updaters.add(new LatestInputFeatures)
      updaters.add(new LastLoginGeoPoint)
      updaters.add(new LastLoginTime)
      updaters.add(new PasswordsUpdates)
      updaters.add(new TimeSlotUpdater)

      //更新链
      val updaterChain = new UpdaterChain(updaters)
      val loginSuccessData: LoginSuccessData =
EvaluateUtil.getLoginSuccessData(value._2)
      updaterChain.doChain(loginSuccessData, historyData)

      //每过来一个登录成功数据，登录次数都应该加1
      currentDayLoginCountState.add(1)
      //从状态中获取到当天的登录次数
      val currentDayLoginCount: Int = currentDayLoginCountState.get()
      historyData.setCurrentDayLoginCount(currentDayLoginCount)

      println(historyData+"========================new")
      //更改状态
      historyDataState.update(historyData);
    }



    return evaluateReport;
```

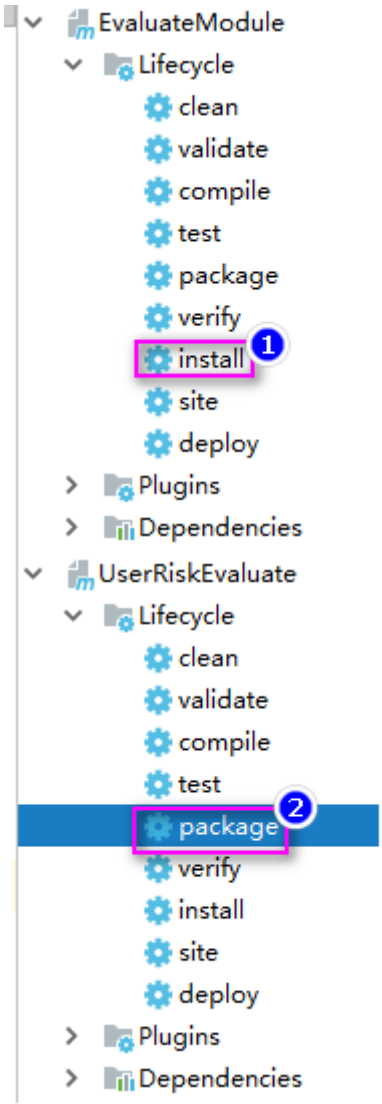
```
    }
}
```

8. 运行代码，通过测试数据完成测试==》确保代码没有问题

9. 添加fatjar插件

```
<build>
    <plugins>
        <!--scala编译插件-->
        <plugin>
            <groupId>net.alchim31.maven</groupId>
            <artifactId>scala-maven-plugin</artifactId>
            <version>4.0.1</version>
            <executions>
                <execution>
                    <id>scala-compile-first</id>
                    <phase>process-resources</phase>
                    <goals>
                        <goal>add-source</goal>
                        <goal>compile</goal>
                    </goals>
                </execution>
            </executions>
        </plugin>
        <!--创建fatjar插件-->
        <plugin>
            <groupId>org.apache.maven.plugins</groupId>
            <artifactId>maven-shade-plugin</artifactId>
            <version>2.4.3</version>
            <executions>
                <execution>
                    <phase>package</phase>
                    <goals>
                        <goal>shade</goal>
                    </goals>
                    <configuration>
                        <filters>
                            <filter>
                                <artifact>*:*</artifact>
                                <excludes>
                                    <exclude>META-INF/*.SF</exclude>
                                    <exclude>META-INF/*.DSA</exclude>
                                    <exclude>META-INF/*.RSA</exclude>
                                </excludes>
                            </filter>
                        </filters>
                    </configuration>
                </execution>
            </executions>
        </plugin>
    </plugins>
</build>
```

10. 打包成fatjar
    - 在EvaluateModule里面执行install
    - 然后再打包



11. 通过FlinkUI界面完成项目的部署

从Kafka消费数据

1. 需要把flink对kafka集成的依赖添加进来

```
 <!--flink集成kafka-->
<!-- https://mvnrepository.com/artifact/org.apache.flink/flink-connector-kafka -->
<dependency>
    <groupId>org.apache.flink</groupId>
    <artifactId>flink-connector-kafka_2.11</artifactId>
    <version>1.10.0</version>
</dependency>
```

2. 在代码，kafkaConsumer

```
//kafka的配置
var properties:Properties=new Properties()
properties.setProperty("bootstrap.servers","hadoop10:9092")

//反序列化机制
var deserializationSchema:DeserializationSchema[String]=new SimpleStringSchema()//字符串反序列化机制
var flinkKafkaConsumer:FlinkKafkaConsumer[String]=new FlinkKafkaConsumer[String]("topica",deserializationSchema,properties)

//从kafka消费数据
val log: DataStream[String] = environment.addSource(flinkKafkaConsumer)
```

业务系统集成Flume

1. 安装Flume
2. Flume在业务系统中的应用一
    - 把日志信息写入到一个磁盘文件
    - flume监控磁盘文件，读取数据写入kafka

需要一个logback.xml更改对应的日志显示方式

```
<?xml version="1.0" encoding="UTF-8"?>
<configuration>

    <!--控制台日志， 控制台输出 -->
    <appender name="STDOUT" class="ch.qos.logback.core.ConsoleAppender">
        <encoder class="ch.qos.logback.classic.encoder.PatternLayoutEncoder">
            <!--格式化输出：%d表示日期，%thread表示线程名，%-5level：级别从左显示5个字符宽度,%msg：日志消息，%n是换行符-->

            <pattern>%-5level %d{yyyy-MM-dd HH:mm:ss} %msg%n</pattern>
```

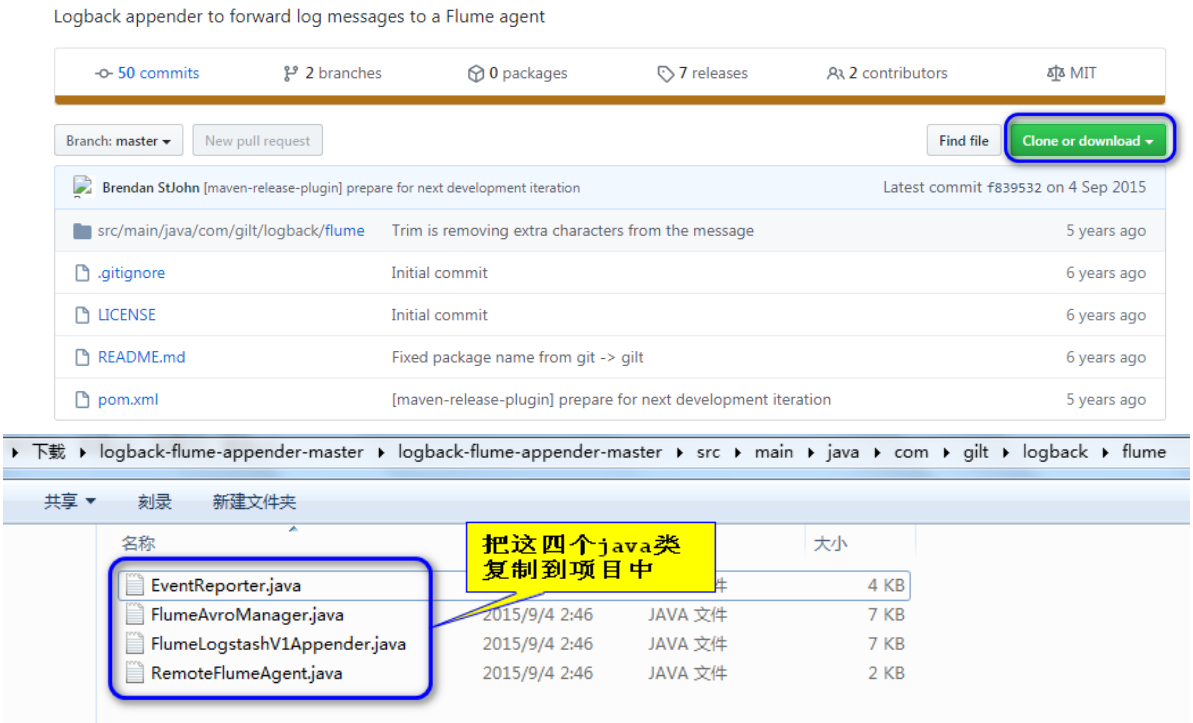
```
        </encoder>
    </appender>

    <!-- 日志输出级别 -->
    <root level="info">
        <appender-ref ref="STDOUT" />
    </root>
</configuration>
```

3. Flume在业务系统中的应用二：直接在springboot项目中集成Flume
    - 添加依赖

```
<dependency>
    <groupId>org.apache.flume</groupId>
    <artifactId>flume-ng-sdk</artifactId>
    <version>1.9.0</version>
</dependency>
```

    - 到github中克隆logback-flume-appender: `https://github.com/gilt/logback-flume-appender`



    - 在logback.xml中添加flume相关的appender

```
<appender name="flume" class="com.baizhi.easyui.flume.FlumeLogstashV1Appender">
    <flumeAgents>
        192.168.77.151:44444,

        192.168.77.151:44444,
```

```
            192.168.77.151:44444
        </flumeAgents>
        <flumeProperties>
            connect-timeout=4000;
            request-timeout=8000
        </flumeProperties>
        <batchSize>1</batchSize>
        <reportingWindow>1000</reportingWindow>
        <reporterMaxThreadPoolSize>150</reporterMaxThreadPoolSize>
        <reporterMaxQueueSize>102400</reporterMaxQueueSize>
        <additionalAvroHeaders>
            myHeader=myValue
        </additionalAvroHeaders>
        <application>smapleapp</application>
        <layout class="ch.qos.logback.classic.PatternLayout">
            <pattern>%p %c#%M %d{yyyy-MM-dd HH:mm:ss} %m%n</pattern>
        </layout>
    </appender>

    <!-- 日志所在的类 -->
    <logger name="com.baizhi.easyui.flume" level="info" additivity="false">
        <!-- 日志打印到控制台 -->
        <appender-ref ref="stdout" />
        <!-- 日志输出到Flume-->
        <appender-ref ref="flume" />
    </logger>
```

- flume的conf目录下创建配置文件avrokafka.properties

```
a1.sources = s1
a1.sinks = sk1
a1.channels = c1

# 组件配置
a1.sources.s1.type = avro
a1.sources.s1.bind = hadoop10
a1.sources.s1.port = 44444

a1.sinks.sk1.type = org.apache.flume.sink.kafka.KafkaSink
a1.sinks.sk1.kafka.bootstrap.servers =hadoop10:9092
a1.sinks.sk1.kafka.topic = topica
a1.sinks.sk1.kafka.flumeBatchSize = 20
a1.sinks.sk1.kafka.producer.acks = 1
a1.sinks.sk1.kafka.producer.linger.ms = 1
a1.sinks.sk1.kafka.producer.compression = snappy


a1.channels.c1.type = memory
a1.channels.c1.capacity = 1000
a1.channels.c1.transactionCapacity = 100
```

```
# 链接组件
a1.sources.s1.channels = c1
a1.sinks.sk1.channel = c1
```

- 启动flume

```
bin/flume-ng agent --conf conf/ --conf-file job/avrokafka.properties --name a1
```

```
* 写测试日志代码--只是为了测试logback.xml基础的配置

  ~~~java
  import org.slf4j.Logger;
  import org.slf4j.LoggerFactory;
  import org.springframework.stereotype.Component;

  @Component
  public class MyLog {
      private static final Logger LOGGER= LoggerFactory.getLogger(MyLog.class);

      public void test(){
          System.out.println("myLog");
          LOGGER.debug("this is flume");
      }
  }
```

4. 系统联调
   - 在业务系统中以日志的方式把云计算需要的数据封装好
   - 启动系统

报告写入到HDFS

把HDFS作为flink的sink

1. 添加`flink-connector-filesystem`依赖

```
<dependency>
    <groupId>org.apache.flink</groupId>
    <artifactId>flink-connector-filesystem_2.11</artifactId>
    <version>1.10.0</version>
</dependency>
```

本地执行，如果使用HDFS，需要添加hadoop的客户端

```
<!--要使用hdfs，还需要在依赖中添加一个hadoo的客户端-->
        <dependency>
            <groupId>org.apache.hadoop</groupId>
            <artifactId>hadoop-client</artifactId>
            <version>2.9.2</version>
        </dependency>
```

2. 写代码

```
var streamFileSink = StreamingFileSink.forRowFormat(new
Path("hdfs://flink.baizhiedu.com:8020/flink-result"),
      new SimpleStringEncoder[EvaluateReport]())
      //按照指定格式生成写入路径;如果没有这个，系统flink会按照其内置的路径yyyy-MM-dd--HH
      .withBucketAssigner(new DateTimeBucketAssigner[EvaluateReport]("yyyy-MM-dd"))
      .build();

result.addSink(streamFileSink);
```