Kafka实现应用日志实时上报统计分析

1、Flume插件

1.1 简介

Apache Flume 是一种分布式的、高可靠的、高可用的日志收集聚合系统，将不同来源海量的日志数据传输到集中的数据存储。

Flume agent 负责把外部事件流（数据流）传输到指定下一跳，agent包括source（数据源）、channel（传输通道）、sink（接收端）。Flume agent可以多跳级联，组成复杂的数据流。 Flume 支持多种类型的source：Avro数据源、Thrift数据源、Kafka数据源、NetCat数据源、Syslog数据源、文件数据源、自定义数据源等，可灵活地与应用系统集成，需要较少的开发代价。 Flume 能够与常见的大数据工具结合，支持多种sink：HDFS、Hive、HBase、Kafka等，将数据传输到这些系统，进行进一步分析处理。

1.2 安装

```
#下载Apache Flume压缩包（需要JDK环境）
wget https://mirror.bit.edu.cn/apache/flume/1.9.0/apache-flume-1.9.0-bin.tar.gz
tar zxvf apache-flume-1.9.0-bin.tar.gz
mv apache-flume-1.9.0-bin flume-1.9.0
```

1.3 配置

```
#拷贝配置文件，从监听端口获取数据，保存到本地文件
cd flume-1.9.0
cp conf/flume-conf.properties.template conf/flume-conf.properties
vim conf/flume-conf.properties
#编辑配置如下
-------------------------------------------------------------------------------
-
# The configuration file needs to define the sources,
# the channels and the sinks.
# Sources, channels and sinks are defined per agent,
# in this case called 'agent'

agent.sources = r1
agent.channels = c1
agent.sinks = s1

# For each one of the sources, the type is defined
agent.sources.r1.type = netcat
agent.sources.r1.bind = 192.168.223.128 #PS:这个地方配成localhost默认会转127.0.0.1
agent.sources.r1.port = 8888

# The channel can be defined as follows.
agent.sources.r1.channels = c1

# Each sink's type must be defined
agent.sinks.s1.type = file_roll
```

```
agent.sinks.s1.sink.directory = /usr/local/flume-1.9.0/logs

#Specify the channel the sink should use
agent.sinks.s1.channel = c1

# Each channel's type is defined.
agent.channels.c1.type = memory

# Other config values specific to each type of channel(sink or source)
# can be defined as well
# In this case, it specifies the capacity of the memory channel
agent.channels.c1.capacity = 100
```

1.4 测试

1.4.1 编写Flume启动/停止/重启脚本

flume的启动，停止，重启脚本比较麻烦，我们这里使用shell脚本编写一个一键启动：

```
vim bin/flume.sh
----------------------------输入如下配置----------------------------------------

#!/bin/bash
#echo "begin start flume..."
#flume的安装根目录（根据自己情况，修#!/bin/bash
#echo "begin start flume..."
#flume的安装根目录（根据自己情况，修改为自己的安装目录）
path=/usr/local/flume-1.9.0
echo "flume home is :$path"
#flume的进程名称，固定值（不用修改）
JAR="flume"
#flume的配置文件名称（根据自己的情况，修改为自己的flume配置文件名称）
Flumeconf="flume-conf.properties"
#定义的soure名称
agentname="agent"
function start(){
        echo "begin start flume process ...."
        #查找flume运行的进程数
        num=$(ps -ef|grep flume-conf|wc -l)
        echo $num
        #判断是否有flume进程运行，如果没有则运行执行启动命令
        if [ $num -lt 1 ] ;then
                $path/bin/flume-ng agent --conf conf -f $path/conf/$Flumeconf --
name $agentname -Dflume.root.logger=INFO,console &
                echo "start success...."
                echo "日志路径: $path/logs/flume.log"
        else
                echo "进程已经存在,启动失败,请检查....."
                exit 0
        fi
}
function stop(){
        echo "begin stop flume process.."
        num=$(ps -ef|grep flume|wc -l)
        echo $num
        #echo "$num...."
        if [ $num -gt 0 ];then
```

```
                #停止flume
                ps -ef|grep flume-conf|awk '{print $2;}'|xargs kill
                echo "进程已经关闭..."
        else
                echo "服务未启动，无须停止..."
        fi
}
function restart(){
        echo "begin stop flume process .."
        #判断程序是否彻底停止
        num=$(ps -ef|grep flume|wc -l)
        #stop完成之后，查找flume的进程数，判断进程数是否为0，如果不为0，则休眠5秒，再次查看，直到进程数为0
        if [ $num -gt 0 ];then
                stop
                echo "flume process stoped,and starting..."
        fi
        #执行start
        start
        echo "started...."
}
#case 命令获取输入的参数，如果参数为start,执行start函数，如果参数为stop执行stop函数，如果参数为restart，执行restart函数
case "$1" in
    "start")
      start $@
      exit 0
    ;;
    "stop")
      stop
      exit 0
     ;;
    "restart")
       restart $@
       exit 0
     ;;
    *)
       echo "用法： $0 {start|stop|restart}"
       exit 1
    ;;
esac
```

命令介绍

参数	作用	举例
–conf 或 -c	指定配置文件夹，包含flume-env.sh和log4j的配置文件	–conf conf
–conf-file 或 -f	配置文件地址	–conf-file conf/flume-conf.properties
–name 或 -n	agent名称	–name agent
-z	zookeeper连接字符串	-z zkhost:2181,zkhost1:2181
-p	zookeeper中的存储路径前缀	-p /flume

启动命令：./bin/flume.sh start

1.4.2 启动客户端测试

```
[root@ydt1 logs]# telnet 127.0.0.1 8888
Trying 127.0.0.1...
Connected to 127.0.0.1.
Escape character is '^]'.
hello flume world
OK

[root@ydt1 logs]# ll
总用量 14220
-rw-r--r--. 1 root root        0 8月  21 17:20 1598001601478-1
-rw-r--r--. 1 root root        0 8月  21 17:24 1598001601478-10
-rw-r--r--. 1 root root        0 8月  21 17:20 1598001601478-2
-rw-r--r--. 1 root root       10 8月  21 17:21 1598001601478-3
-rw-r--r--. 1 root root        0 8月  21 17:21 1598001601478-4
-rw-r--r--. 1 root root        0 8月  21 17:22 1598001601478-5
-rw-r--r--. 1 root root        0 8月  21 17:22 1598001601478-6
-rw-r--r--. 1 root root        0 8月  21 17:23 1598001601478-7
-rw-r--r--. 1 root root        0 8月  21 17:23 1598001601478-8
-rw-r--r--. 1 root root        0 8月  21 17:24 1598001601478-9
-rw-r--r--. 1 root root 14554540 8月  21 16:19 flume.log

[root@ydt1 logs]# pwd
/usr/local/flume-1.9.0/logs
#查看生成的日志信息：
[root@ydt1 ~]# cat /usr/local/flume-1.9.0/logs/1598001601478-10
hello flume world
```

2、Flume集成Kafka

2.1 配置kafka信息

```
# Name the components on this agent
```

```
agent.sources = r1
agent.sinks = k1
agent.channels = c1

# Describe/configure the source
agent.sources.r1.type = netcat
agent.sources.r1.bind = 192.168.223.128
agent.sources.r1.port = 8888

# Describe the sink
agent.sinks.k1.type = org.apache.flume.sink.kafka.KafkaSink
agent.sinks.k1.kafka.bootstrap.servers=192.168.223.128:9092
agent.sinks.k1.kafka.topic=log4j-flume-kafka
agent.sinks.k1.serializer.class=kafka.serializer.StringEncoder
agent.sinks.k1.kafka.producer.acks=1
agent.sinks.k1.custom.encoding=UTF-8


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

# Bind the source and sink to the channel
## 配置传输通道
agent.sources.r1.channels = c1
agent.sinks.k1.channel = c1
```

2.2 启动zookeeper，kafka，flume

```
#启动单台zookeeper
[root@ydt1 zookeeper-3.4.6]# ./bin/zkServer.sh start
JMX enabled by default
Using config: /usr/local/zookeeper-3.4.6/bin/../conf/zoo.cfg
Starting zookeeper ... STARTED

#现在我们只启动单台kafka，所以当前kafka建立过集群，请先删除kafka日志信息，主要是元数据,再启动kafka之前
rm /tmp/kafka-logs/meta.properties -f
[root@ydt1 zookeeper-3.4.6]# cd ../kafka_2.12-2.5.0/
[root@ydt1 kafka_2.12-2.5.0]# ./bin/kafka-server-start.sh config/server.properties

#启动flume脚本
./bin/flume.sh start
```

2.3 测试

```
cd /usr/local/kafka_2.12-2.5.0
#创建kafka topic，有就算了
./bin/kafka-topics.sh --bootstrap-server ydt1:9092 --create --topic log4j-flume-kafka
#启动kafka消费者，消费log4j-flume-kafka数据
./bin/kafka-console-consumer.sh --bootstrap-server ydt1:9092 --topic log4j-flume-kafka --from-beginning
```

```
cd /usr/local/flume-1.9.0
#启动flume服务
./bin/flume.sh start

#连接flume，写入数据信息
[root@ydt1 kafka_2.12-2.5.0]# telnet 192.168.223.128 8888
Trying 192.168.223.128...
Connected to 192.168.223.128.
Escape character is '^]'.
gebilaowang
OK
隔壁老王是一个热心的邻居
OK


#可以看到kafka消费者已经可以收到数据
[root@ydt1 kafka_2.12-2.5.0]# ./bin/kafka-console-consumer.sh --bootstrap-server
ydt1:9092 --topic log4j-flume-kafka --from-beginning
OpenJDK 64-Bit Server VM warning: If the number of processors is expected to
increase from one, then you should configure the number of parallel GC threads
appropriately using -XX:ParallelGCThreads=N
gebilaowang
隔壁老王是一个热心的邻居3、
```

3、Flume生产日志收集

3.1 日志收集配置

实际应用中，flume常和应用程序部署在同一机器上，应用程序将日志写入文件中，flume再以监听命令的方式（tail命令打开文件）对该文件进行监听，再把其传入到Kafka集群中。flume的配置为：

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

#指定源类型为Linux 命令(单个文件)
#agent.sources.r1.type = exec
#agent.sources.r1.command = tail -f /usr/local/redis-4.0.6/log/redis.log

#多个文件
agent.sources.r1.type = TAILDIR
agent.sources.r1.filegroups = f1
agent.sources.r1.filegroups.f1 = /usr/local/redis-4.0.6/log/.*

#指定事件不包括头信息
#agent.sources.r1.fileHeader = false


# Describe the sink
agent.sinks.k1.type = org.apache.flume.sink.kafka.KafkaSink
agent.sinks.k1.kafka.bootstrap.servers=192.168.223.128:9092
agent.sinks.k1.kafka.topic=log4j-flume-kafka
```

```
agent.sinks.k1.serializer.class=kafka.serializer.StringEncoder
agent.sinks.k1.kafka.producer.acks=1
agent.sinks.k1.custom.encoding=UTF-8


# Use a channel which buffers events in memory
#直接使用内存做数据的临时缓存，虽然快，但是安全性不高，不过我们这里只是记录日志，就算了，如果是那种重要的实时统计，还是需要使用文件数据临时缓存的形式
agent.channels.c1.type = memory
agent.channels.c1.capacity = 1000
agent.channels.c1.transactionCapacity = 100
#agent.channels.c1.type = file
#agent.channels.c1.checkpointDir = /usr/local/flume-1.9.0/log/checkpoint
#agent.channels.c1.dataDirs = /usr/local/flume-1.9.0/log/data

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

这样，Kafka和flume集群故障时，都不会影响到应用程序的正常运行。flume成了Kafka的一个producer，因为flume是一个轻服务应用，可在每台应用服务器上都部署一个。

3.2 测试

```
#重新启动flume
./bin/flume-ng agent -n agent -c conf -f conf/flume-conf.properties -Dflume.root.logger=INFO,console
```

再次打开kafka消费者客户端，可以看到redis.log文件中所有信息！

```
[root@ydt1 kafka_2.12-2.5.0]# ./bin/kafka-console-consumer.sh --bootstrap-server ydt1:9092 --topic log4j-flume-kafka --from-beginning
OpenJDK 64-Bit Server VM warning: If the number of processors is expected to increase from one, then you should configure the number of p
lel GC threads appropriately using -XX:ParallelGCThreads=N
gebilaowang
隔壁老王是一个热心的邻居
haha
ddddd
17482:S 21 Aug 22:56:10.560 * Connecting to MASTER 192.168.223.129:6379
17482:S 21 Aug 22:56:10.560 * MASTER <-> SLAVE sync started
17482:S 21 Aug 22:56:12.955 # Error condition on socket for SYNC: No route to host
17482:S 21 Aug 22:56:13.581 * Connecting to MASTER 192.168.223.129:6379
17482:S 21 Aug 22:56:13.581 * MASTER <-> SLAVE sync started
17482:S 21 Aug 22:56:14.588 - DB 0: 5 keys (0 volatile) in 8 slots HT.
17482:S 21 Aug 22:56:14.588 - 0 clients connected (0 slaves), 835783 bytes in use
17482:S 21 Aug 22:56:15.960 # Error condition on socket for SYNC: No route to host
17482:S 21 Aug 22:56:16.634 * Connecting to MASTER 192.168.223.129:6379
17482:S 21 Aug 22:56:16.634 * MASTER <-> SLAVE sync started
17482:S 21 Aug 22:56:18.966 # Error condition on socket for SYNC: No route to host
17482:S 21 Aug 22:56:19.905 - DB 0: 5 keys (0 volatile) in 8 slots HT.
17482:S 21 Aug 22:56:19.905 - 0 clients connected (0 slaves), 835783 bytes in use
17482:S 21 Aug 22:56:19.905 * Connecting to MASTER 192.168.223.129:6379
17482:S 21 Aug 22:56:19.905 * MASTER <-> SLAVE sync started
17482:S 21 Aug 22:56:21.973 # Error condition on socket for SYNC: No route to host
17482:S 21 Aug 22:56:22.935 * Connecting to MASTER 192.168.223.129:6379
17482:S 21 Aug 22:56:22.935 * MASTER <-> SLAVE sync started
17482:S 21 Aug 22:56:24.949 - DB 0: 5 keys (0 volatile) in 8 slots HT.
17482:S 21 Aug 22:56:24.949 - 0 clients connected (0 slaves), 835783 bytes in use
17482:S 21 Aug 22:56:24.978 # Error condition on socket for SYNC: No route to host
17482:S 21 Aug 22:56:25.957 * Connecting to MASTER 192.168.223.129:6379
17482:S 21 Aug 22:56:25.957 * MASTER <-> SLAVE sync started
17482:S 21 Aug 22:56:27.984 # Error condition on socket for SYNC: No route to host
```

4、Fink安装和简单实用

4.1 概述

Apache Flink是一个框架和分布式处理引擎，用于对无界和有界数据流进行有状态计算。Flink设计为在所有常见的集群环境中运行，以内存速度和任何规模执行计算。

官方下载地址：https://flink.apache.org/downloads.html

4.2 安装配置

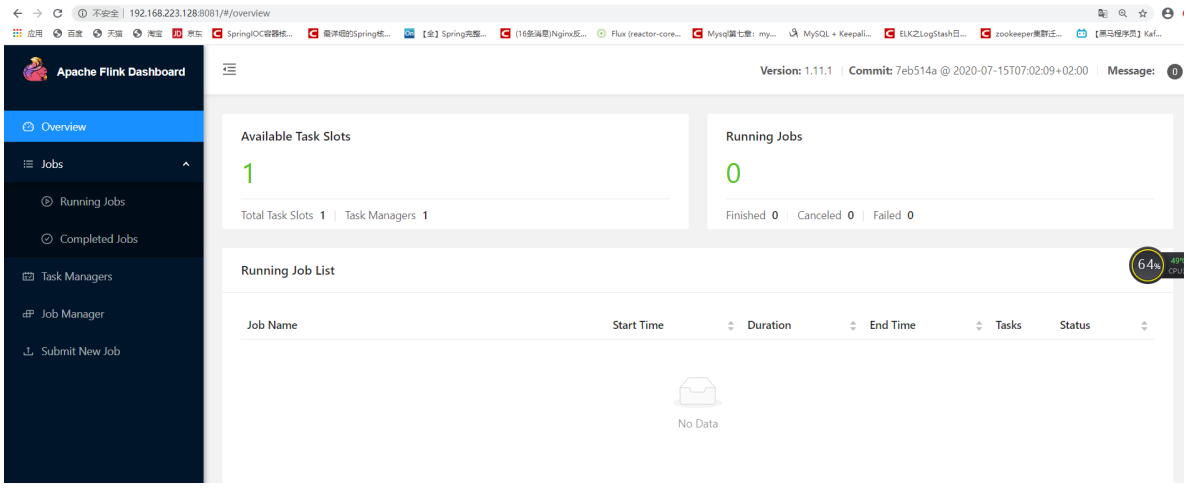
```
#下载安装
wget https://mirror.bit.edu.cn/apache/flink/flink-1.11.1/flink-1.11.1-bin-scala_2.11.tgz
tar -xvf flink-1.11.1-bin-scala_2.11.tgz

#配置
cd flink-1.11.1/
vim conf/flink-conf.yaml
#修改该配置为主机名或者ip：
jobmanager.rpc.address: ydt1
```

4.3 启动测试

```
[root@ydt1 flink-1.11.1]# ./bin/start-cluster.sh
Starting cluster.
Starting standalonesession daemon on host ydt1.
Starting taskexecutor daemon on host ydt1.
```

关闭防火墙或者开启8081端口：service firewalld stop

访问IP地址为：http://192.168.223.128:8081/



4.4 体验Flink

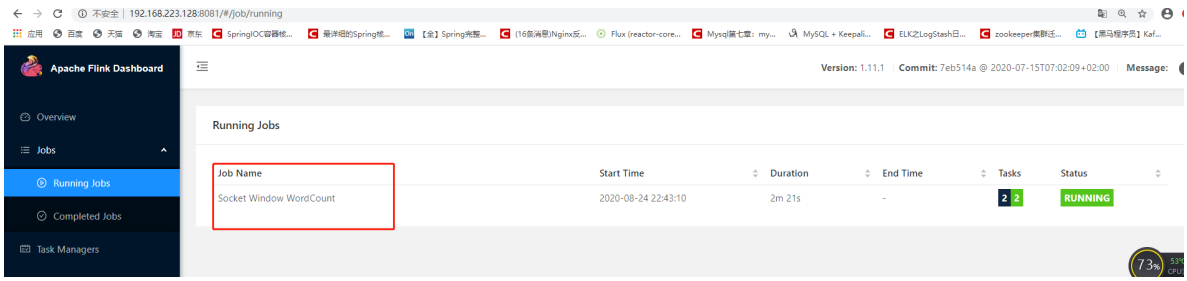
在机器上执行以下命令连接本机9000端口：

```
nc -l 9000
```

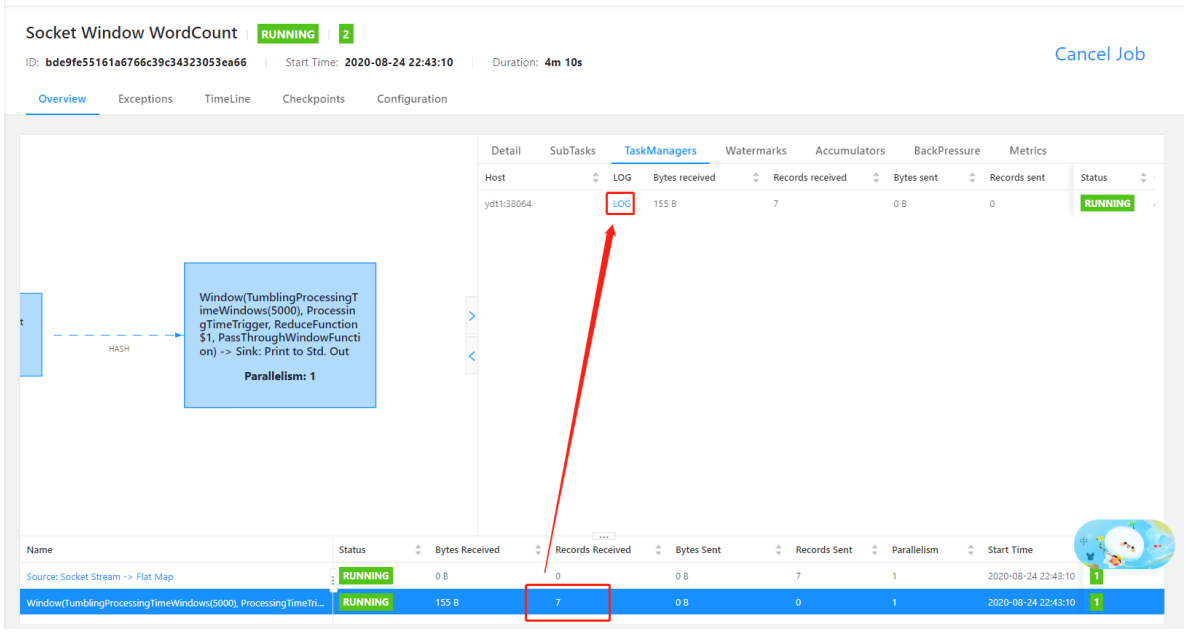
执行以下命令，即可监听本机9000端口，等待该端口的数据：

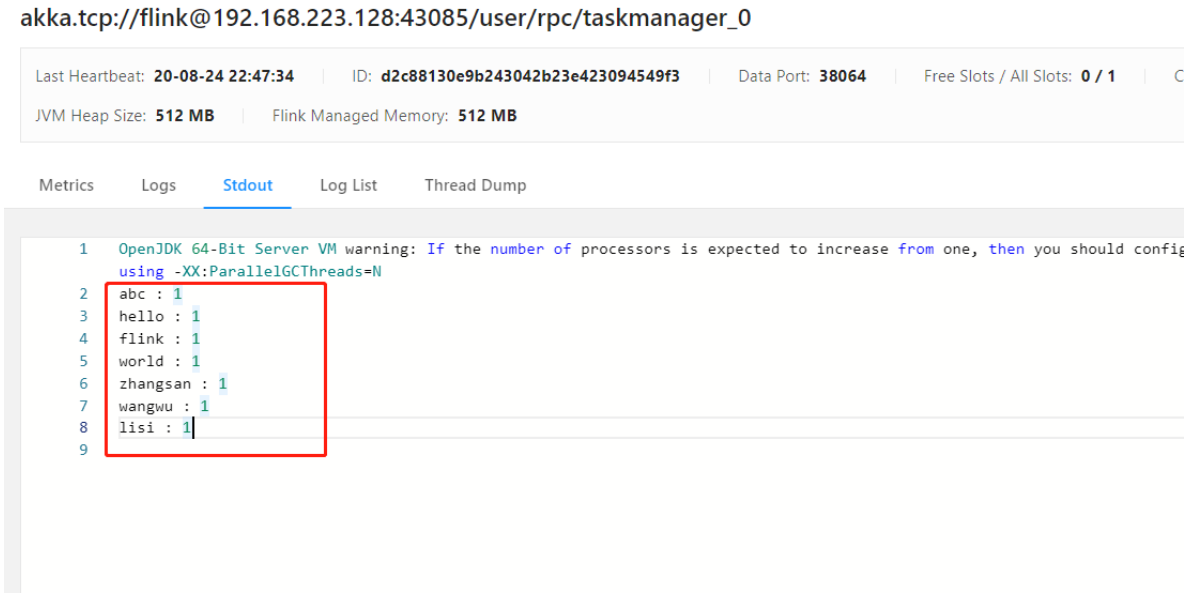
```
./bin/flink run examples/streaming/SocketWindowWordCount.jar --port 9000
```

此时再去刷新web页面，可见如下信息，新增了一个Job，这个job只是计数的样例



点进去可以看到接收的数据条数，想看详情继续点进去看：





5、Flink集成Kafka

Flink 提供了特殊的Kafka Connectors来从Kafka topic中读取数据或者将数据写入到Kafkatopic中，Flink的Kafka Consumer与Flink的检查点机制相结合，提供exactly-once处理语义。为了做到这一点，Flink并不完全依赖于Kafka的consumer组的offset跟踪，而是在自己的内部去跟踪和检查。

上一章节，我们知道flink主要是通过执行jar包任务来调度日志消息，所以我们需要定义一个maven项目来获取Kafka主题消息

5.1 引入pom依赖

```
<dependencies>

        <!-- https://mvnrepository.com/artifact/org.apache.flink/flink-java -->
        <dependency>
            <groupId>org.apache.flink</groupId>
            <artifactId>flink-java</artifactId>
            <version>1.11.1</version>
        </dependency>

        <!-- https://mvnrepository.com/artifact/org.apache.flink/flink-clients -
->
        <dependency>
            <groupId>org.apache.flink</groupId>
            <artifactId>flink-clients_2.11</artifactId>
            <version>1.11.1</version>
        </dependency>

        <!-- https://mvnrepository.com/artifact/org.apache.flink/flink-
streaming-java -->
        <dependency>
            <groupId>org.apache.flink</groupId>
            <artifactId>flink-streaming-java_2.11</artifactId>
            <version>1.11.1</version>
            <!-- <scope>provided</scope>-->
        </dependency>

        <!-- https://mvnrepository.com/artifact/org.apache.flink/flink-
connector-kafka-0.11 -->
        <dependency>
            <groupId>org.apache.flink</groupId>
            <artifactId>flink-connector-kafka_2.11</artifactId>
            <version>1.11.1</version>
        </dependency>


        <!-- https://mvnrepository.com/artifact/org.slf4j/slf4j-log4j12 -->
        <dependency>
            <groupId>org.slf4j</groupId>
            <artifactId>slf4j-log4j12</artifactId>
            <version>1.7.25</version>
        </dependency>

        <!-- https://mvnrepository.com/artifact/log4j/log4j -->
        <dependency>
            <groupId>log4j</groupId>
            <artifactId>log4j</artifactId>
            <version>1.2.17</version>
        </dependency>

    </dependencies>

    <build>
        <plugins>
            <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-compiler-plugin</artifactId>
                <version>3.2</version>
                <configuration>
```

```
                    <source>1.8</source>
                    <target>1.8</target>
                </configuration>
            </plugin>


            <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-dependency-plugin</artifactId>
                <executions>
                    <execution>
                        <id>copy-dependencies</id>
                        <phase>test</phase>
                        <goals>
                            <goal>copy-dependencies</goal>
                        </goals>
                        <configuration>
                            <outputDirectory>
                                target/classes/lib
                            </outputDirectory>
                        </configuration>
                    </execution>
                </executions>
            </plugin>
            <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-jar-plugin</artifactId>
                <configuration>
                    <archive>
                        <manifest>
                            <addClasspath>true</addClasspath>
                            <mainClass>
                                com.ydt.flinkkafka.FlinkKafka
                            </mainClass>
                            <classpathPrefix>lib/</classpathPrefix>
                        </manifest>
                        <manifestEntries>
                            <Class-Path>.</Class-Path>
                        </manifestEntries>
                    </archive>
                </configuration>
            </plugin>
        </plugins>
    </build>
```

5.2 创建一个Flink任务执行类

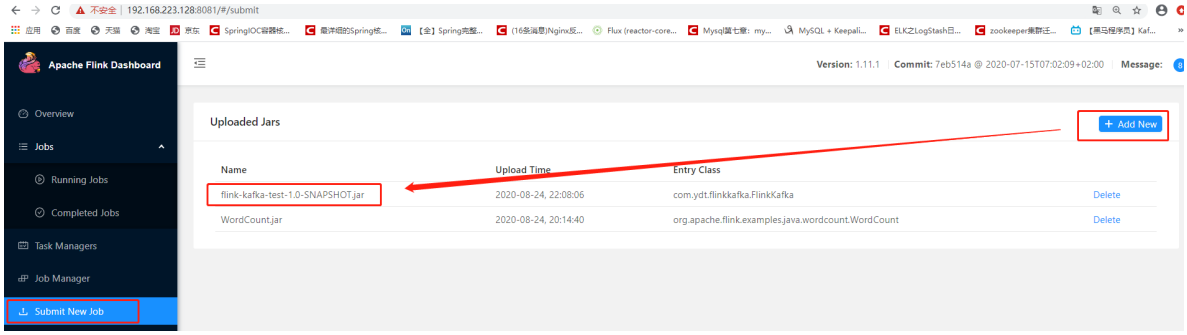
创建一个Flink任务执行类，将Kafka数据转为转为flink的dataStream类型

```
package com.ydt.flinkkafka;

import org.apache.flink.api.common.serialization.SimpleStringSchema;
import org.apache.flink.streaming.api.TimeCharacteristic;
import org.apache.flink.streaming.api.datastream.DataStreamSink;
import org.apache.flink.streaming.api.environment.StreamExecutionEnvironment;
import org.apache.flink.streaming.connectors.kafka.FlinkKafkaConsumer;
```

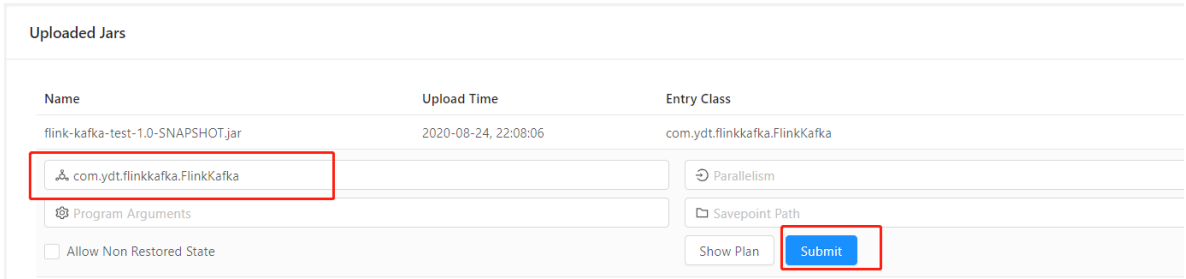
```
import java.util.Properties;

public class FlinkKafka {

    public static void main(String[] args) throws Exception {
        try {
            // 获取上下文环境StreamExecutionEnvironment对象
            final StreamExecutionEnvironment env =
StreamExecutionEnvironment.getExecutionEnvironment();
            env.enableCheckpointing(5000); // 要设置启动检查点
            env.setStreamTimeCharacteristic(TimeCharacteristic.EventTime);//设置
事件触发时写入流

            // 配置kafka的ip和端口，以及消费者组
            Properties properties = new Properties();
            properties.setProperty("bootstrap.servers", "ydt1:9092");
            properties.setProperty("group.id", "flume-kafka");
            //将消费者数据对象加入到上下文环境StreamExecutionEnvironment对象中，并生成
DataStream对象；
            DataStreamSink<String> dataStream =env.
                    addSource(new FlinkKafkaConsumer<>("log4j-flume-kafka", new
SimpleStringSchema(), properties))
                    .print();
            //设置job名称
            env.execute("consumer from kafka data");
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

5.3 打包上传启动

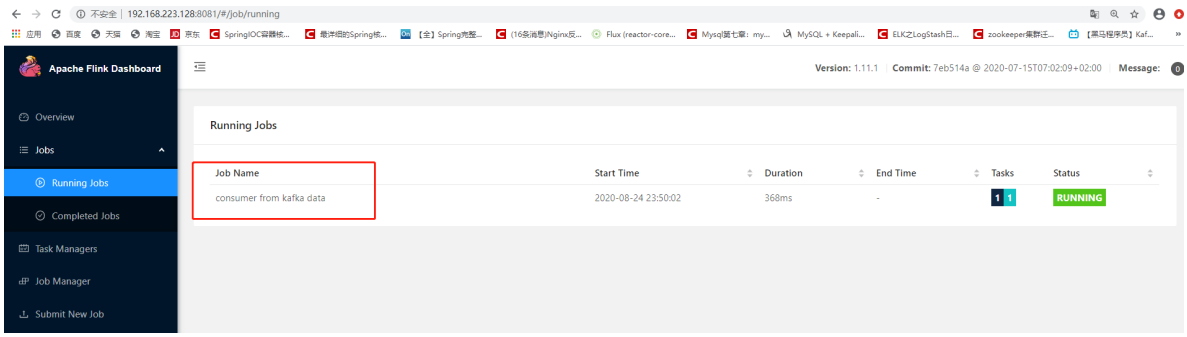
将以上项目打成jar包包上传到flink，并且启动



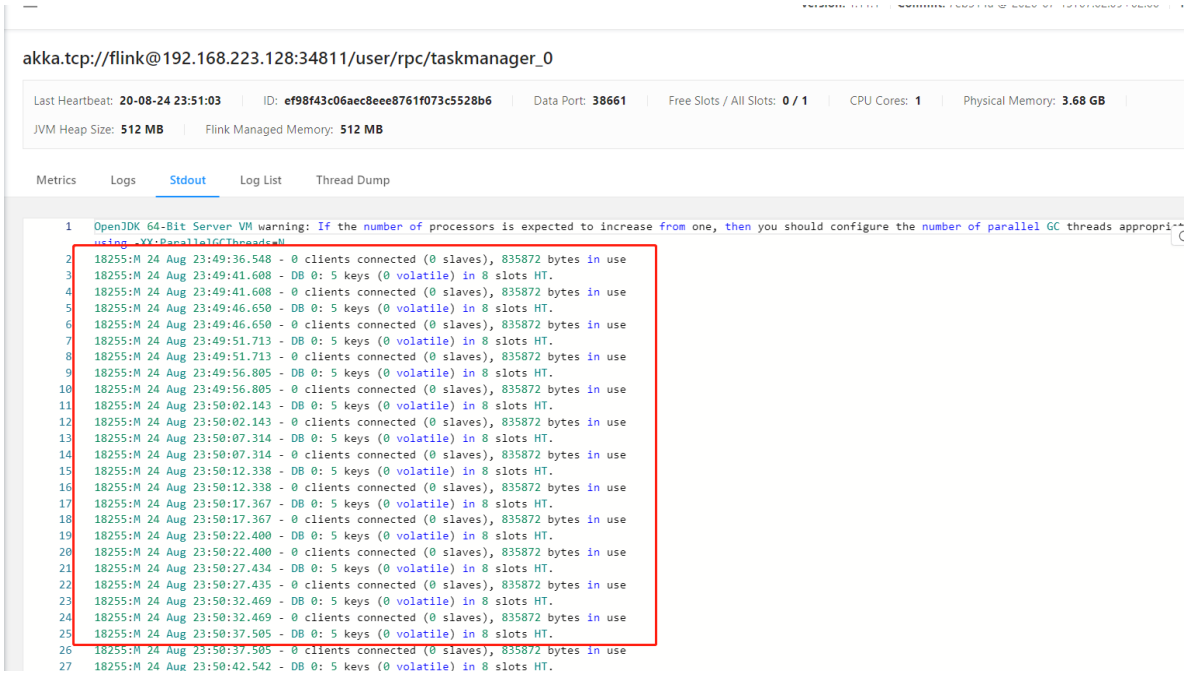
执行项目中的任务类：com.ydt.flinkkafka.FlinkKafka



可以看到刚刚上传的job包已经在运行了：



点进去可以看到日志信息：



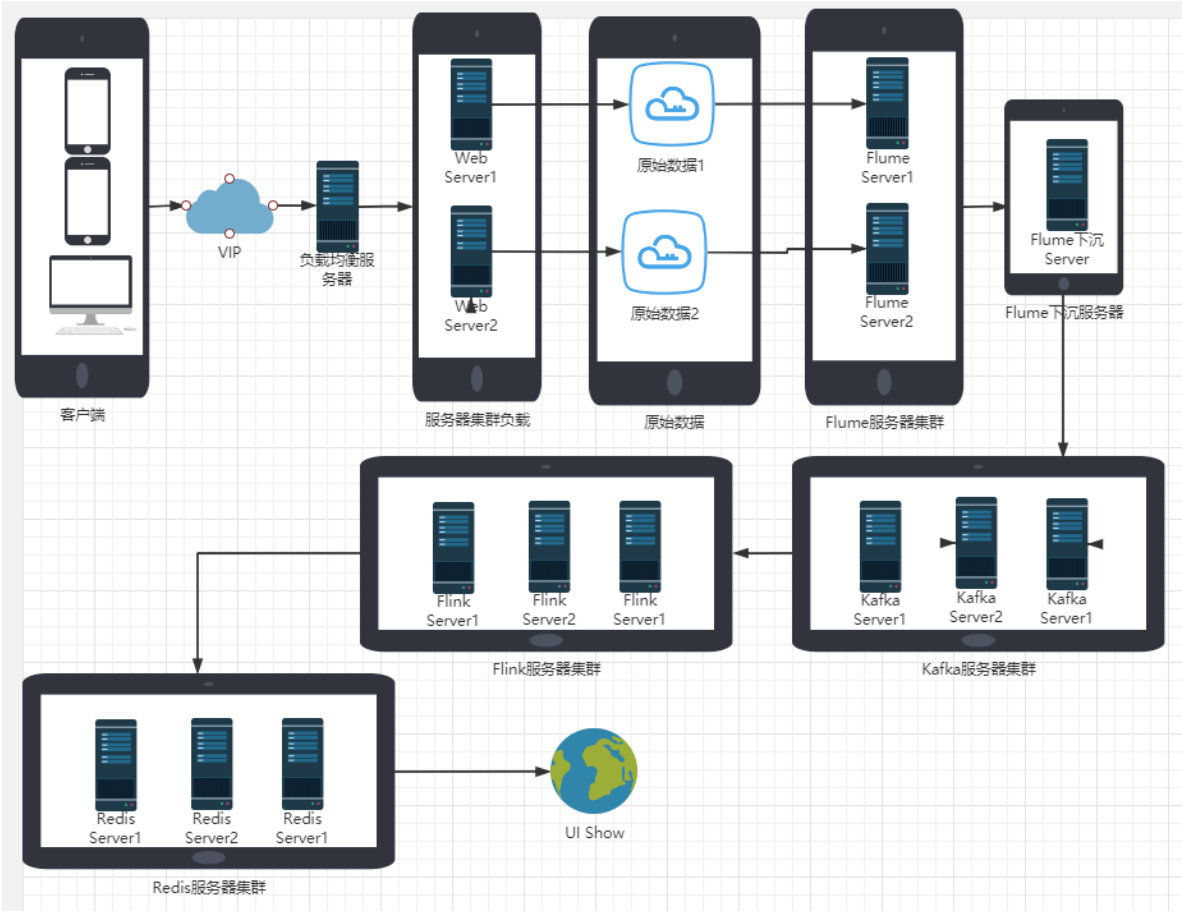
5、搭建一个完整的实时日志统计平台

在互联网应用中，不管是哪一种处理方式，其基本的数据来源都是日志数据，例如对于web应用来说，则可能是用户的访问日志、用户的点击日志等。

如果对于数据的分析结果在时间上有比较严格的要求，则可以采用在线处理的方式来对数据进行分析，如使用Flink进行处理。比较贴切的一个例子是天猫双十一的成交额，在其展板上，我们看到交易额是实时动态进行更新的，对于这种情况，则需要采用在线处理。下面要介绍的是实时数据处理方式，即基于Flink的在线处理，在下面给出的完整案例中，我们将会完成下面的几项工作：

- 1.如何一步步构建我们的实时处理系统（Flume+Kafka+Flink+Redis）
- 2.实时处理网站的用户访问日志，并统计出该网站的PV(访问量)、UV(独立访客)，IP
- 3.将实时分析出的PV、UV动态地展示在我们的前端页面上

5.1 架构图



即从上面的架构中我们可以看出，其由下面的几部分构成：

- Flume集群
- Kafka集群
- Flink集群

从构建实时处理系统的角度出发，我们需要做的是，如何让数据在各个不同的集群系统之间打通（从上面的图示中也能很好地说明这一点），即需要做各个系统之前的整合，包括Flume与Kafka的整合，Kafka与Flink的整合。当然，各个环境是否使用集群，依个人的实际需要而定，在我们的环境中，Flume、Kafka、Flink都使用集群。

5.2 服务器矩阵图

Web服务节点：两台提供应用服务，一台提供负载均衡服务

Flume:两台采集数据

Zookeeper：三台服务搭建集群

Kafka:三台服务搭建集群

Flink:三台服务搭建集群

Redis：三台服务搭建集群

Keepalived：负责将负载均衡服务器漂移VIP（这里就不备份了，你高兴可以继续搞一个负载均衡服务来漂）

VIP：虚拟IP

	192.168.223.128	192.168.223.129	192.168.223.130	192.168.223.131
Flume		1	1	
Zookeeper	1	1	1	
Kafka	1	1	1	
Flink	1（主）	1	1	
Redis	1	1	1	
VIP				1
Web服务节点	1（负载均衡）	1	1	
Keppalived	1（VIP漂移）			

5.2 Web服务和负载均衡节点

我们的目标是是是统计访客数据，这里以Nginx作为Web服务器（你高兴可以用tomcat）,主要是统计访问日志access.log进行分析

因为我们重点不在nginx的使用，所以仅仅只是访问一个静态图片而已，你高兴你可以弄的更复杂!

想获取用户真实IP，我们需要安装realip模块，否则得到的是代理服务器IP

```
#129,130节点需要安装realip模块
cd /usr/local/nginx-1.12.2
./configure --prefix=/usr/local/nginx --with-http_stub_status_module --with-
http_ssl_module --with-http_realip_module
make && make install
```

129，130节点 Nginx服务器nginx.conf配置：

```
server {
      listen 80;
      location / {
          root html;
          set_real_ip_from 192.168.223.128; #指接受从哪个信任前代理处获得真实用户ip
          real_ip_header X-Real-IP; #存储X-Real-IP变量名称
      }
    }
#访问日志格式
log_format  main  '$remote_addr~$time_local~$request~$status~$http_user_agent';
#开启访问日志记录
access_log  logs/access.log  main;
#弄张图片到nginx/html目录下，启动nginx，访问http://192.168.223.129/mv.jpg，可以看到美女就行
```

128节点Nginx服务器nginx.conf负载均衡配置：

```
upstream test {
   server 192.168.223.129:80; #内部服务器1
   server 192.168.223.130:80; #内部服务器2
}
server {
   listen 80;
   location / {
      proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;#记录来访IP
      proxy_pass http://test;
   }
}
#访问负载均衡代理节点http://192.168.223.128/mv.jpg可以看到两台机器上的美女即可
```

5.3 配置keepalived VIP漂移

编辑keepalived的配置如下：

```
vim /etc/keepalived/keepalived.conf

-------------------------------------------------------------------------
! Configuration File for keepalived
global_defs {
    #不与其他节点重名即可
   router_id flume-kafka-flink-redis
}

vrrp_instance kafka {
    state MASTER
    interface eth0      #指定虚拟ip的网卡接口
    mcast_src_ip 192.168.223.128
    virtual_router_id 51    #路由器标识，MASTER和BACKUP必须是一致的
    priority 100            #定义优先级，数字越大，优先级越高，在同一个vrrp_instance下，
MASTER的优先级必须大于BACKUP的优先级。这样MASTER故障恢复后，就可以将VIP资源再次抢回来
    authentication {
        auth_type PASS
        auth_pass 1111
    }
    virtual_ipaddress {
        192.168.223.132
    }
}
#访问VIP地址：http://192.168.223.131/mv.jpg可以看到两台机器上的美女即可
```

5.4 搭建Flume数据采集

Flume安装略！

129,130节点Flume配置文件flume-conf.properties如下：

```
#################################################################
##
##主要作用是监听文件中的新增数据，采集到数据之后，输出到avro
##    注意：Flume agent的运行，主要就是配置source channel sink
##  下面的a1就是agent的代号，source叫r1 channel叫c1 sink叫k1
#################################################################
a1.sources = r1
```

```
a1.sinks = k1
a1.channels = c1

#对于source的配置描述 监听文件中的新增数据 exec
a1.sources.r1.type = exec
a1.sources.r1.command  = tail -f /usr/local/nginx/logs/access.log

#对于sink的配置描述 使用avro日志做数据的消费
a1.sinks.k1.type = avro
#这个地方配置你的hostname或者ip
a1.sinks.k1.hostname = ydt2 or ydt3
a1.sinks.k1.port = 44444

#对于channel的配置描述 使用文件做数据的临时缓存 这种的安全性要高
a1.channels.c1.type = file
a1.channels.c1.checkpointDir = /usr/local/flume-1.9.0/checkpoint
a1.channels.c1.dataDirs = /usr/local/flume-1.9.0/data

#通过channel c1将source r1和sink k1关联起来
a1.sources.r1.channels = c1
a1.sinks.k1.channel = c1
```

```
#启动Flume Agent，对日志文件进行监听
./bin/flume-ng agent --conf conf -n a1 -f conf/flume-conf.properties >/dev/null
2>&1 &
```

5.5 部署Kafka集群

Kafka集群部署（略，参照之前课程）,注意需要先启动zookeeper集群，再启动kafka集群（如果kafka集群结构有变化，需要删除元数据）

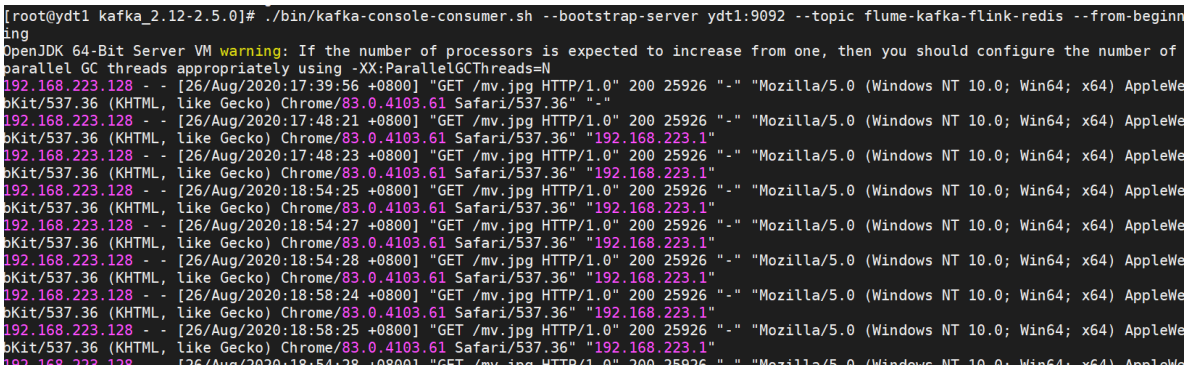
```
./bin/flume.sh restart
```

启动成功：

启动Flume下沉节点：

```
#创建kafka主题
[root@ydt1 kafka_2.12-2.5.0]# ./bin/kafka-topics.sh --bootstrap-server ydt1:9092
--create --topic flume-kafka-flink-redis
OpenJDK 64-Bit Server VM warning: If the number of processors is expected to
increase from one, then you should configure the number of parallel GC threads
appropriately using -XX:ParallelGCThreads=N
Created topic flume-kafka-flink-redis.

#启动kafka消费者，消费flume-kafka-flink-redis数据
./bin/kafka-console-consumer.sh --bootstrap-server ydt1:9092 --topic flume-
kafka-flink-redis --from-beginning
```

可以看到原始数据采集到了kafka集群：



5.6 Flink集群搭建

安装（略，参照之前课程）

Flink可在所有类UNIX环境中运行，例如Linux，Mac OS X和Cygwin（适用于Windows），并期望集群由一个主节点和一个或多个工作节点组成。在开始设置系统之前，请确保在每个节点上安装了以下软件：

- Java 1.8.x或更高版本，
- ssh（必须运行sshd才能使用管理远程组件的Flink脚本）

  如果您的群集不满足这些软件要求，则需要安装/升级它。

  在所有群集节点上使用无密码SSH和相同的目录结构将允许您使用我们的脚本来控制所有内容。
- 无密码SSH配置

```
#1. 过程为对每个节点，生成密钥对，然后将生成的所有公钥都追加 authorized_keys 文件中，再
将authorized_keys文件放到每个节点 ~/.ssh/ 下

#2. 在每个节点上生成密钥对，一路回车，生成密钥对：id_rsa 和 d_rsa.pub，默认存储在
/home/jiecxy/.ssh 下：
# 例：在master节点上
# 生成密钥对
[root@ydt1 ~]$ ssh-keygen -t rsa -P ''
# 将 id_rsa.pub 追加到授权的key中
[root@ydt1 ~]$ cat ~/.ssh/id_rsa.pub >> ~/.ssh/authorized_keys



#3. 每个节点修改配置文件 /etc/ssh/sshd_config
# 例：在master节点上
```

```
[root@ydt1 ~]$ sudo vi /etc/ssh/sshd_config
#修改内容如下：
RSAAuthentication yes # 启用 RSA 认证
PubkeyAuthentication yes # 启用公钥私钥配对认证方式
AuthorizedKeysFile .ssh/authorized_keys # 公钥文件路径

#4. 使每个 authorized_keys 包含所有公钥
# 在master节点上
# 复制 authorized_keys 到 worker1，会提示输入worker1的密码，下面的 jiecxy 为用户名，三台节点均有相同的用户名 jiecxy
[root@ydt1 ~]$ scp ~/.ssh/authorized_keys jiecxy@worker1:~/

#然后登陆 worker1，追加master的公钥到worker1的authorized_keys，如下：

# 在 worker1 上
[root@ydt2 ~]$ chmod 700 ~/.ssh
[root@ydt2 ~]$ cat ~/authorized_keys >> ~/.ssh/authorized_keys
[root@ydt2 ~]$ rm ~/authorized_keys
此时authorized_keys包含master和worker1的公钥，将该authorized_keys复制到worker2，追加worker2的公钥

# 在 worker1 上远程拷贝到worker2
[root@ydt2 ~]$ scp ~/.ssh/authorized_keys jiecxy@worker2:~/

# 在 worker2 上
[root@ydt3 ~]$ cat ~/authorized_keys >> ~/.ssh/authorized_keys
[root@ydt3 ~]$ rm ~/authorized_keys

#注：若果还有节点，一次类推，一直到最后一个节点，最后节点上的authorized_keys就拥有所有节点的公钥，然后再把该authorized_keys传到其他所有节点对应位置
# 在 worker2 上
[root@ydt3 ~]$ scp ~/.ssh/authorized_keys
jiecxy@worker1:~/.ssh/authorized_keys
[root@ydt3 ~]$ scp ~/.ssh/authorized_keys
jiecxy@master:~/.ssh/authorized_keys

#另外，将所有节点的authorized_keys改一下权限 !!!非常重要!!!
$ chmod 600 ~/.ssh/authorized_keys

#测试
#需要先重启sshd服务，不行就重启服务器试试
service sshd restart
#在命令行直接输入 ssh worker1 看是否不需要输入密码就能登陆
# 在master节点上
[root@ydt1 flink-1.11.1]# ssh ydt2
Last login: Wed Aug 26 22:21:24 2020 from ydt1
[root@ydt2 ~]#
```

我们只使用**Standalone 模式集群**

5.6.1 主节点128上修改

```
/usr/local/flink-1.11.1
#master配置
vim conf/masters  #输入
--------------------------------------------------------------------------
```

```
ydt1:8081

#slaves配置
vim conf/workers  #输入
-----------------------------------------------------------------------
ydt2
ydt3
#wq保存退出

#flink-conf.yaml配置
vim conf/flink-conf.yaml #输入
-----------------------------------------------------------------------
#每个TaskManager提供的任务槽数。每个插槽运行一个并行管道。
taskmanager.numberOfTaskSlots: 2
jobmanager.rpc.address: ydt1
```

5.6.2 拷贝安装包

scp将安装包复制到129,130节点

```
scp -r flink-1.11.1/ ydt2:/usr/local/ #弄到129
scp -r flink-1.11.1/ ydt2:/usr/local/ #弄到130
```

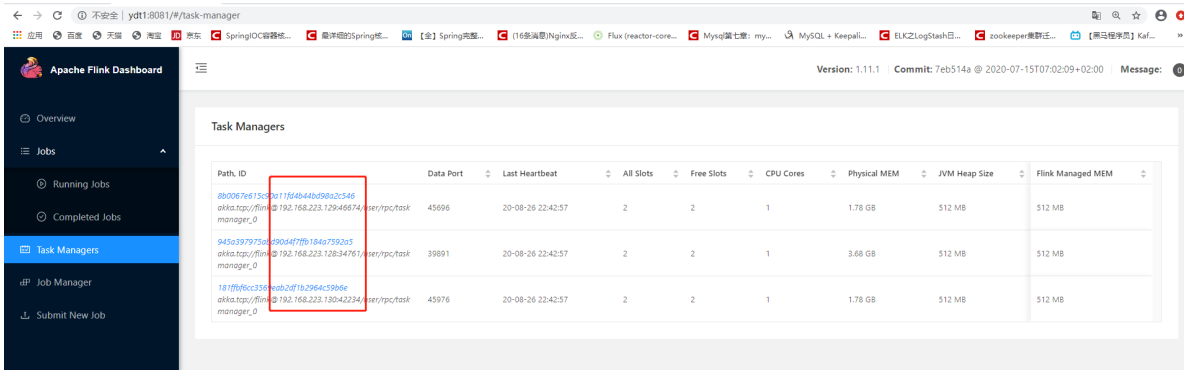
5.6.3 配置环境变量

配置所有节点Flink的环境变量

```
#vim /etc/profile

export FLINK_HOME=/usr/local/flink-1.11.1
export PATH=$PATH:$FLINK_HOME/bin

#立马生效
source /etc/profile
```

5.6.4 启动Flink集群

```
#master节点启动
[root@ydt1 flink-1.11.1]# ./bin/start-cluster.sh
Starting cluster.
Starting standalonesession daemon on host ydt1.
Starting taskexecutor daemon on host ydt2.
Starting taskexecutor daemon on host ydt3.
```



5.7 Kafka+Flink整合

该章节主要是将Kafka中的byte[]数据转换为对象，所以不能使用之前Flink提供的简单类型的schema了，需要自己定义一个转换类:

5.7.1 转换的实体对象

```
package com.ydt.flinkkafka;

import java.io.Serializable;

public class MyAccess implements Serializable {
    private String ip;

    private String browser;

    private String date;

    private String status;

    public MyAccess(String ip, String browser, String date, String status) {
        this.ip = ip;
        this.browser = browser;
        this.date = date;
        this.status = status;
    }

    public String getIp() {
        return ip;
    }

    public void setIp(String ip) {
        this.ip = ip;
    }

    public String getBrowser() {
        return browser;
    }

    public void setBrowser(String browser) {
        this.browser = browser;
    }

    public String getDate() {
        return date;
    }
```

```
    public void setDate(String date) {
        this.date = date;
    }

    public String getStatus() {
        return status;
    }

    public void setStatus(String status) {
        this.status = status;
    }

    @Override
    public String toString() {
        return "MyAccess{" +
                "ip='" + ip + '\'' +
                ", browser='" + browser + '\'' +
                ", date='" + date + '\'' +
                ", status='" + status + '\'' +
                '}';
    }
}
```

5.7.2 自定义转换类

```
package com.ydt.flinkkafka;

import org.apache.flink.api.common.serialization.DeserializationSchema;
import org.apache.flink.api.common.typeinfo.TypeInformation;
import org.apache.flink.api.java.typeutils.TypeExtractor;

import java.io.IOException;
import java.nio.ByteBuffer;
import java.nio.ByteOrder;
import java.nio.CharBuffer;
import java.nio.charset.Charset;
import java.nio.charset.CharsetDecoder;

public class ConsumerDeserializationSchema implements
DeserializationSchema<MyAccess> {

    private Class<MyAccess> clazz;

    public ConsumerDeserializationSchema(Class<MyAccess> clazz) {
        this.clazz = clazz;
    }


    @Override
    public MyAccess deserialize(byte[] message) throws IOException {
        ByteBuffer buffer =
ByteBuffer.wrap(message).order(ByteOrder.LITTLE_ENDIAN);

        String mess = byteBuffertoString(buffer);
        //封装为POJO类
```

```
        String[] split = mess.split("~");
        MyAccess myAccess = new MyAccess(split[0],split[4],split[1],split[3]);
        return myAccess;
    }

    public static String byteBuffertoString(ByteBuffer buffer) {
        Charset charset = null;
        CharsetDecoder decoder = null;
        CharBuffer charBuffer = null;
        try {
            charset = Charset.forName("UTF-8");
            decoder = charset.newDecoder();
            // charBuffer = decoder.decode(buffer);//用这个的话，只能输出来一次结果，
第二次显示为空
            charBuffer = decoder.decode(buffer.asReadOnlyBuffer());
            return charBuffer.toString();
        } catch (Exception ex) {
            ex.printStackTrace();
            return "";
        }
    }


    @Override
    public boolean isEndOfStream(MyAccess myAccess) {
        return false;
    }

    @Override
    public TypeInformation<MyAccess> getProducedType() {
        return TypeExtractor.getForClass(clazz);
    }
}
```

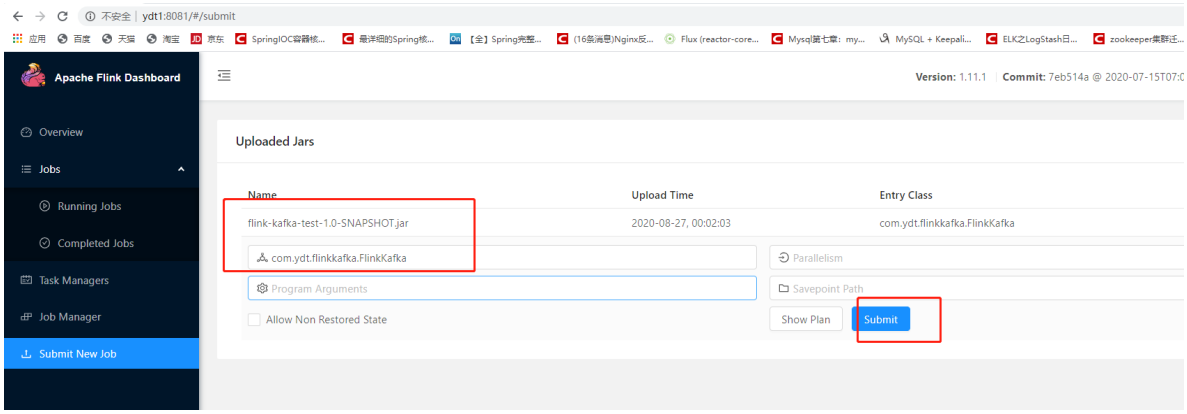
5.7.3 消费者处理类

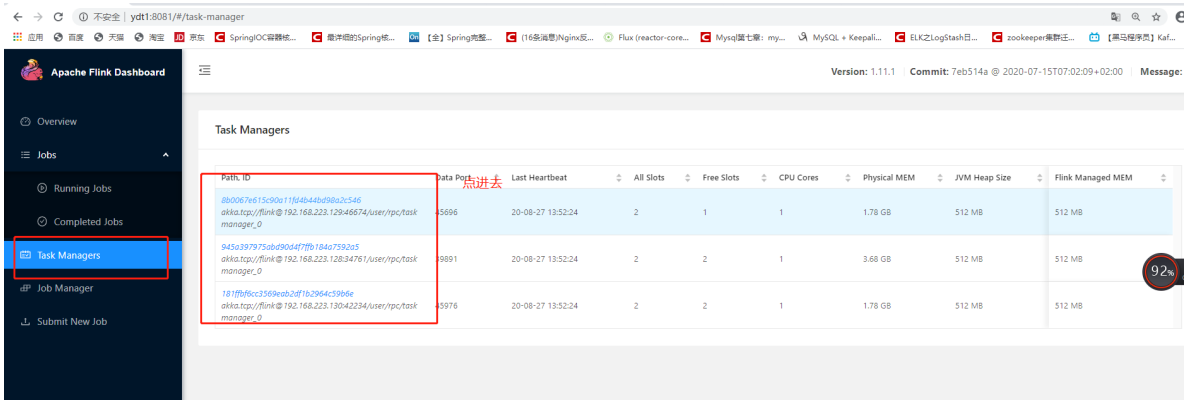
```
package com.ydt.flinkkafka;

import org.apache.flink.api.common.eventtime.WatermarkGenerator;
import org.apache.flink.api.common.eventtime.WatermarkGeneratorSupplier;
import org.apache.flink.api.common.eventtime.WatermarkStrategy;
import org.apache.flink.api.common.serialization.SimpleStringSchema;
import org.apache.flink.metrics.MetricGroup;
import org.apache.flink.streaming.api.TimeCharacteristic;
import org.apache.flink.streaming.api.datastream.DataStreamSink;
import org.apache.flink.streaming.api.datastream.DataStreamSource;
import org.apache.flink.streaming.api.environment.StreamExecutionEnvironment;
import org.apache.flink.streaming.api.functions.source.SourceFunction;
import org.apache.flink.streaming.api.watermark.Watermark;
import org.apache.flink.streaming.connectors.kafka.FlinkKafkaConsumer;
import org.apache.kafka.clients.consumer.ConsumerRecords;
import org.apache.kafka.clients.consumer.KafkaConsumer;

import java.text.ParseException;
import java.text.SimpleDateFormat;
import java.util.Collections;
```

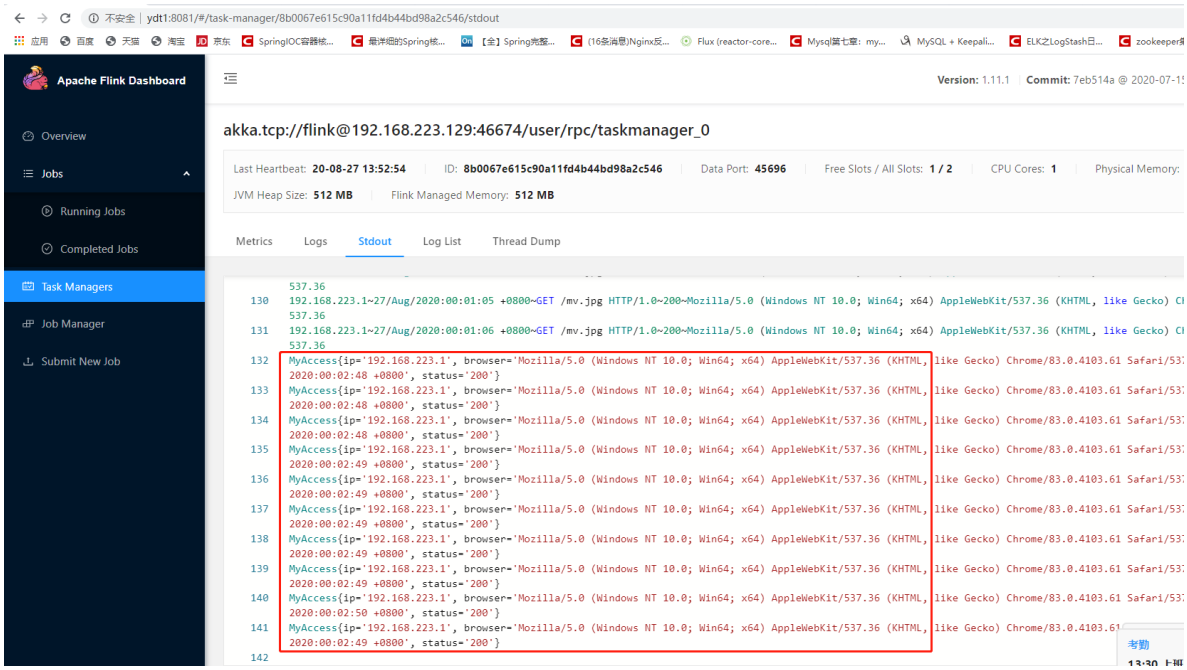
```
import java.util.Date;
import java.util.Properties;

public class FlinkKafka {

    public static void main(String[] args) throws Exception {
        try {
            // 获取上下文环境StreamExecutionEnvironment对象
            final StreamExecutionEnvironment env =
StreamExecutionEnvironment.getExecutionEnvironment();
            env.enableCheckpointing(5000); // 要设置启动检查点
            env.setStreamTimeCharacteristic(TimeCharacteristic.EventTime);//设置
事件触发时写入流

            // 配置kafka的ip和端口，以及消费者组
            Properties properties = new Properties();
            properties.setProperty("bootstrap.servers", "ydt1:9092");
            properties.setProperty("group.id", "kafka-flink");
            SimpleDateFormat dateFormat = new SimpleDateFormat("yyyy-MM-dd");//
注意月份是MM
            FlinkKafkaConsumer<MyAccess> consumer
                    = new FlinkKafkaConsumer<MyAccess>("flume-kafka-flink-redis"
                    , new ConsumerDeserializationSchema(MyAccess.class),
properties);
            //将消费者数据对象加入到上下文环境StreamExecutionEnvironment对象中，并生成
DataStream对象；
            env.addSource(consumer).print();
            //设置job名称
            env.execute("consumer from kafka data");
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

5.7.4 打包上传到Flink

打包上窜到Flink,并启动任务







5.8 Flink+Redis整合

在很多大数据场景下，要求数据形成数据流的形式进行计算和存储。上面介绍了Flink消费Kafka数据实现对象转换，该章节需要完成的是将实时计算的结果写到redis。当kafka从其他端获取数据立刻到Flink计算，Flink计算完后结果写到Redis，整个过程就像流水一样形成了数据流的处理

5.8.1 增加POM依赖

```
<dependency>
            <groupId>org.apache.flink</groupId>
            <artifactId>flink-connector-redis_2.10</artifactId>
            <version>1.1.5</version>
        </dependency>
```

5.8.2 修改消费者处理类

增加flink sink代码，将数据实时刷入redis

```
package com.ydt.flinkkafka;

import org.apache.flink.api.common.eventtime.WatermarkGenerator;
import org.apache.flink.api.common.eventtime.WatermarkGeneratorSupplier;
import org.apache.flink.api.common.eventtime.WatermarkStrategy;
import org.apache.flink.api.common.functions.FlatMapFunction;
import org.apache.flink.api.common.serialization.SimpleStringSchema;
```

```
import org.apache.flink.api.java.tuple.Tuple2;
import org.apache.flink.metrics.MetricGroup;
import org.apache.flink.streaming.api.TimeCharacteristic;
import org.apache.flink.streaming.api.datastream.DataStream;
import org.apache.flink.streaming.api.datastream.DataStreamSink;
import org.apache.flink.streaming.api.datastream.DataStreamSource;
import org.apache.flink.streaming.api.environment.StreamExecutionEnvironment;
import org.apache.flink.streaming.api.functions.source.SourceFunction;
import org.apache.flink.streaming.api.watermark.Watermark;
import org.apache.flink.streaming.connectors.kafka.FlinkKafkaConsumer;
import org.apache.flink.streaming.connectors.redis.RedisSink;
import
org.apache.flink.streaming.connectors.redis.common.config.FlinkJedisPoolConfig;
import org.apache.flink.streaming.connectors.redis.common.mapper.RedisCommand;
import
org.apache.flink.streaming.connectors.redis.common.mapper.RedisCommandDescriptio
n;
import org.apache.flink.streaming.connectors.redis.common.mapper.RedisMapper;
import org.apache.flink.util.Collector;
import org.apache.kafka.clients.consumer.ConsumerRecords;
import org.apache.kafka.clients.consumer.KafkaConsumer;
import redis.clients.jedis.Jedis;

import java.text.DateFormat;
import java.text.ParseException;
import java.text.SimpleDateFormat;
import java.util.Collections;
import java.util.Date;
import java.util.Properties;

public class FlinkKafka {

    private static Jedis jedis = new Jedis("ydt1",6379);

    private static DateFormat format = new SimpleDateFormat("yyyy-MM-dd");

    public static void main(String[] args) throws Exception {
        try {
            // 获取上下文环境StreamExecutionEnvironment对象
            final StreamExecutionEnvironment env =
StreamExecutionEnvironment.getExecutionEnvironment();
            // 为了打印到控制台的结果不乱序，我们配置全局的并发为1，这里改变并发对结果正确性
没有影响
            env.setParallelism(1);
            /*ProcessingTime：事件被处理的时间。也就是由机器的系统时间来决定。(默认)
            EventTime：事件发生的时间。一般就是数据本身携带的时间。*/
            env.setStreamTimeCharacteristic(TimeCharacteristic.EventTime);
            // 配置kafka的ip和端口，以及消费者组
            Properties properties = new Properties();
            properties.setProperty("bootstrap.servers", "ydt1:9092");
            properties.setProperty("group.id", "kafka-flink");
            SimpleDateFormat dateFormat = new SimpleDateFormat("yyyy-MM-dd");//
注意月份是MM
            FlinkKafkaConsumer<MyAccess> consumer
                    = new FlinkKafkaConsumer<MyAccess>("flume-kafka-flink-redis"
                    , new ConsumerDeserializationSchema(MyAccess.class),
properties);
```

```
            //将消费者数据对象加入到上下文环境StreamExecutionEnvironment对象中，并生成
DataStream对象；
            DataStreamSource<MyAccess> streamSource = env.addSource(consumer);
            streamSource.print();
            //实例化Flink和Redis关联类FlinkJedisPoolConfig，设置Redis端口
            FlinkJedisPoolConfig conf = new FlinkJedisPoolConfig.Builder()
                    .setHost("192.168.223.128")
                    .setPort(6379)
                    .build();
            //实例化RedisSink，并通过flink的addSink的方式将flink转换的结果插入到redis
            //ip
            streamSource.addSink(new RedisSink(conf,new
RedisExampleMapperIp()));
            //浏览器
            streamSource.addSink(new RedisSink(conf,new
RedisExampleMapperBrowser()));
            //日期
            streamSource.addSink(new RedisSink(conf,new
RedisExampleMapperDate()));
            //设置job名称
            env.execute("consumer from kafka data");
        } catch (Exception e) {
            e.printStackTrace();
        }
    }

    //指定Redis key并将flink数据类型映射到Redis数据类型
    public static final class RedisExampleMapperIp implements
RedisMapper<MyAccess>{


        @Override
        public RedisCommandDescription getCommandDescription() {
            return new
RedisCommandDescription(RedisCommand.HSET,"my_access_ip");
        }

        @Override
        public String getKeyFromData(MyAccess myAccess) {
            return myAccess.getIp();
        }

        @Override
        public String getValueFromData(MyAccess myAccess) {
            String value = jedis.hget("my_access_ip", myAccess.getIp());
            if(value == null){
                return 1+"";
            }
            return String.valueOf(Integer.valueOf(value)+1);
        }
    }

    //指定Redis key并将flink数据类型映射到Redis数据类型
    public static final class RedisExampleMapperBrowser implements
RedisMapper<MyAccess>{


        @Override
```

```
        public RedisCommandDescription getCommandDescription() {
            return new
RedisCommandDescription(RedisCommand.HSET,"my_access_browser");
        }

        @Override
        public String getKeyFromData(MyAccess myAccess) {
            return getBrowserType(myAccess.getBrowser());
        }

        @Override
        public String getValueFromData(MyAccess myAccess) {
            String browserType = getBrowserType(myAccess.getBrowser());
            String value = jedis.hget("my_access_browser", browserType);
            if(value == null){
                return 1+"";
            }
            return String.valueOf(Integer.valueOf(value)+1);
        }
    }

    public static String getBrowserType(String browser){
        if(browser.indexOf("AppleWebKit") != -1){
            return "Google";
        }else if(browser.indexOf("QQBrowser") != -1){
            return "QQ";
        }else if(browser.indexOf("Trident") != -1){
            return "IE";
        }else if(browser.indexOf("Firefox") != -1){
            return "Firefox";
        }
        return "other";
    }

    //指定Redis key并将flink数据类型映射到Redis数据类型
    public static final class RedisExampleMapperDate implements
RedisMapper<MyAccess>{


        @Override
        public RedisCommandDescription getCommandDescription() {
            return new
RedisCommandDescription(RedisCommand.HSET,"my_access_date");
        }

        @Override
        public String getKeyFromData(MyAccess myAccess) {
            String format = FlinkKafka.format.format(new Date());
            return format;
        }

        @Override
        public String getValueFromData(MyAccess myAccess) {
            String format = FlinkKafka.format.format(new Date());
            String value = jedis.hget("my_access_date", format);
            if(value == null){
                return 1+"";
            }
```

```
                return String.valueOf(Integer.valueOf(value)+1);
            }
        }
    }
```

5.8.3 打包上传测试

切换浏览器统计结果如下：