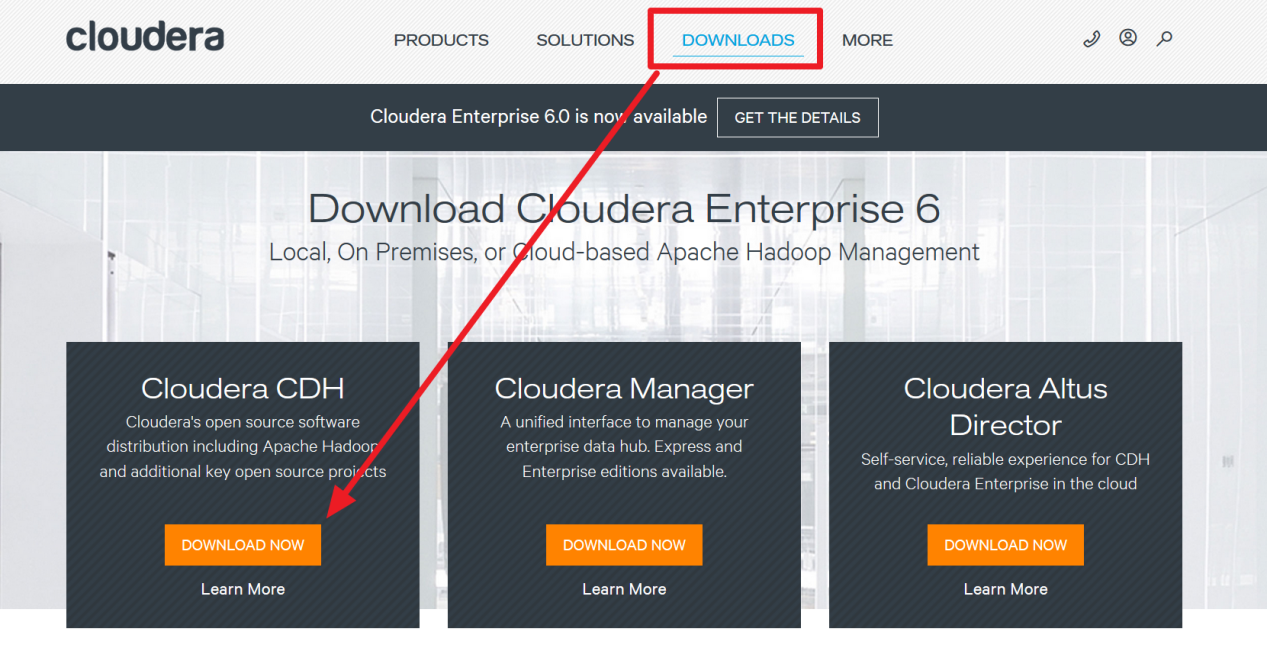
# Flume、kafka、spark

CDH地址：<https://www.cloudera.com/downloads.html>



# 一、Flume简介

　　flume 作为 cloudera 开发的实时日志收集系统，Flume 初始的发行版本目前被统称为 Flume OG（original generation），属于 cloudera。

　　但随着 FLume 功能的扩展，Flume OG 代码工程臃肿、核心组件设计不合理、核心配置不标准等缺点暴露出来，尤其是在 Flume OG 的最后一个发行版本 0.9.4. 中，日志传输不稳定的现象尤为严重，为了解决这些问题，2011 年 10 月 22 号，cloudera 完成了 Flume-728，对 Flume 进行了里程碑式的改动：重构核心组件、核心配置以及代码架构，重构后的版本统称为 Flume NG（next generation）；改动的另一原因是将 Flume 纳入 apache 旗下，cloudera Flume 改名为 Apache Flume。

　　备注：Flume参考资料

　　　　官方网站： http://flume.apache.org/  
　　　　用户文档： http://flume.apache.org/FlumeUserGuide.html  
　　　　开发文档： http://flume.apache.org/FlumeDeveloperGuide.html

# 二、Flume特点

　　flume是一个分布式、可靠、和高可用的海量日志采集、聚合和传输的系统。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(比如文本、HDFS、Hbase等)的能力 。  
　　flume的数据流由事件(Event)贯穿始终。事件是Flume的基本数据单位，它携带日志数据(字节数组形式)并且携带有头信息，这些Event由Agent外部的Source生成，当Source捕获事件后会进行特定的格式化，然后Source会把事件推入(单个或多个)Channel中。你可以把Channel看作是一个缓冲区，它将保存事件直到Sink处理完该事件。

　　Sink负责持久化日志或者把事件推向另一个Source。  
   
　　1）flume的可靠性   
　　　　当节点出现故障时，日志能够被传送到其他节点上而不会丢失。Flume提供了三种级别的可靠性保障，从强到弱依次分别为：end-to-end（收到数据agent首先将event写到磁盘上，当数据传送成功后，再删除；如果数据发送失败，可以重新发送。），Store on failure（这也是scribe采用的策略，当数据接收方crash时，将数据写到本地，待恢复后，继续发送），Besteffort（数据发送到接收方后，不会进行确认）。

　　2）flume的可恢复性还是靠Channel。推荐使用FileChannel，事件持久化在本地文件系统里(性能较差)。

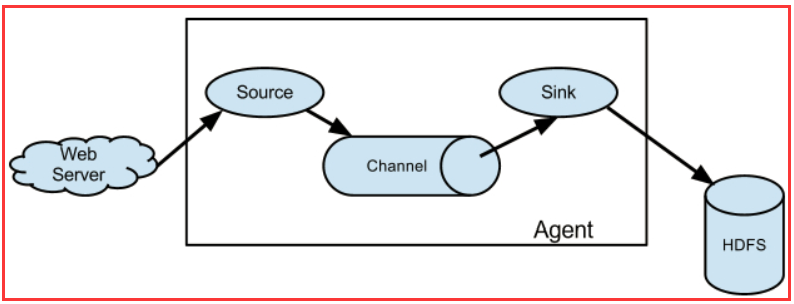
# 三、Flume的一些核心概念

　　Client：Client生产数据，运行在一个独立的线程。

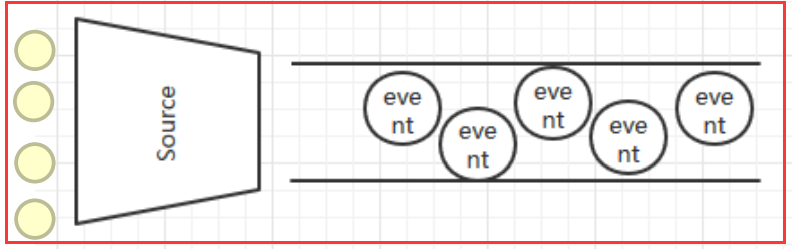
　　Event： 一个数据单元，消息头和消息体组成。（Events可以是日志记录、 avro 对象等。）  
　　Flow： Event从源点到达目的点的迁移的抽象。  
　　Agent： 一个独立的Flume进程，包含组件Source、 Channel、 Sink。（Agent使用JVM 运行Flume。每台机器运行一个agent，但是可以在一个agent中包含多个sources和sinks。）  
　　Source： 数据收集组件。（source从Client收集数据，传递给Channel）  
　　Channel： 中转Event的一个临时存储，保存由Source组件传递过来的Event。（Channel连接 sources 和 sinks ，这个有点像一个队列。）  
　　Sink： 从Channel中读取并移除Event， 将Event传递到FlowPipeline中的下一个Agent（如果有的话）（Sink从Channel收集数据，运行在一个独立线程。）

## 3.1、Agent结构

　　Flume 运行的核心是 Agent。Flume以agent为最小的独立运行单位。一个agent就是一个JVM。它是一个完整的数据收集工具，含有三个核心组件，分别是source、 channel、 sink。通过这些组件， Event 可以从一个地方流向另一个地方，如下图所示。



## 3.2、source

　　Source是数据的收集端，负责将数据捕获后进行特殊的格式化，将数据封装到事件（event） 里，然后将事件推入Channel中。 Flume提供了很多内置的  
　　Source， 支持 Avro， log4j， syslog 和 http post(body为json格式)。可以让应用程序同已有的Source直接打交道，如AvroSource，  
　　SyslogTcpSource。 如果内置的Source无法满足需要， Flume还支持自定义Source。  


　source类型：

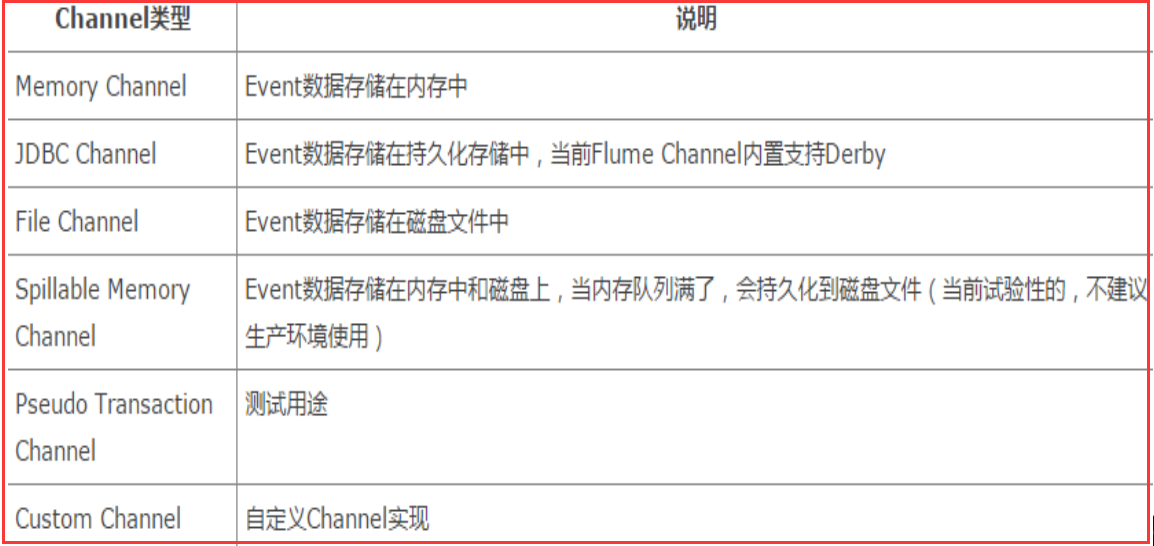


## 3.3、Channel

Channel是连接Source和Sink的组件，大家可以将它看做一个数据的缓冲区（数据队列），它可以将事件暂存到内存中也可以持久化到本地磁盘上， 直到Sink处理完该事件。介绍两个较为常用的Channel， MemoryChannel和FileChannel。

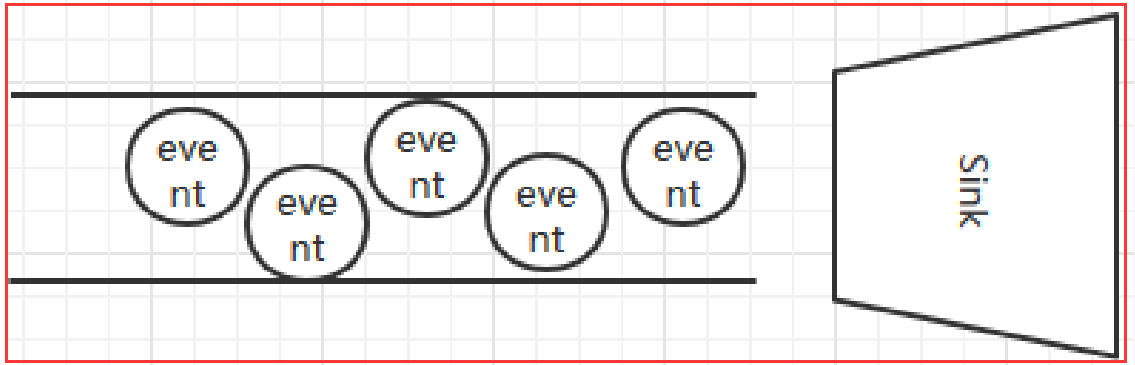
数据只有存储在下一个存储位置（可能是最终的存储位置，如HDFS；也可能是下一个Flume节点的channel），数据才会从当前的channel中删除。这个过程是通过事务来控制的，这样就保证了数据的可靠性。

　　Channel类型：



## 3.4、Sink

　　Sink从Channel中取出事件，然后将数据发到别处，可以向文件系统、数据库、 hadoop存数据， 也可以是其他agent的Source。在日志数据较少时，可  
　　以将数据存储在文件系统中，并且设定一定的时间间隔保存数据。



　　Sink类型：



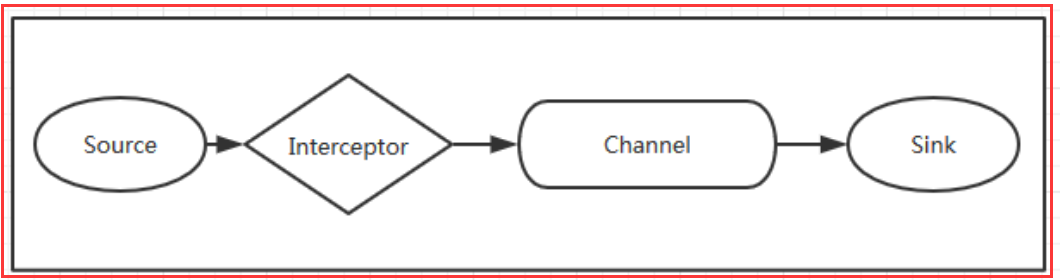
# 四、Flume拦截器、数据流以及可靠性

## 4.1、Flume拦截器

　　当我们需要对数据进行过滤时，除了我们在Source、 Channel和Sink进行代码修改之外， Flume为我们提供了拦截器，拦截器也是chain形式的。

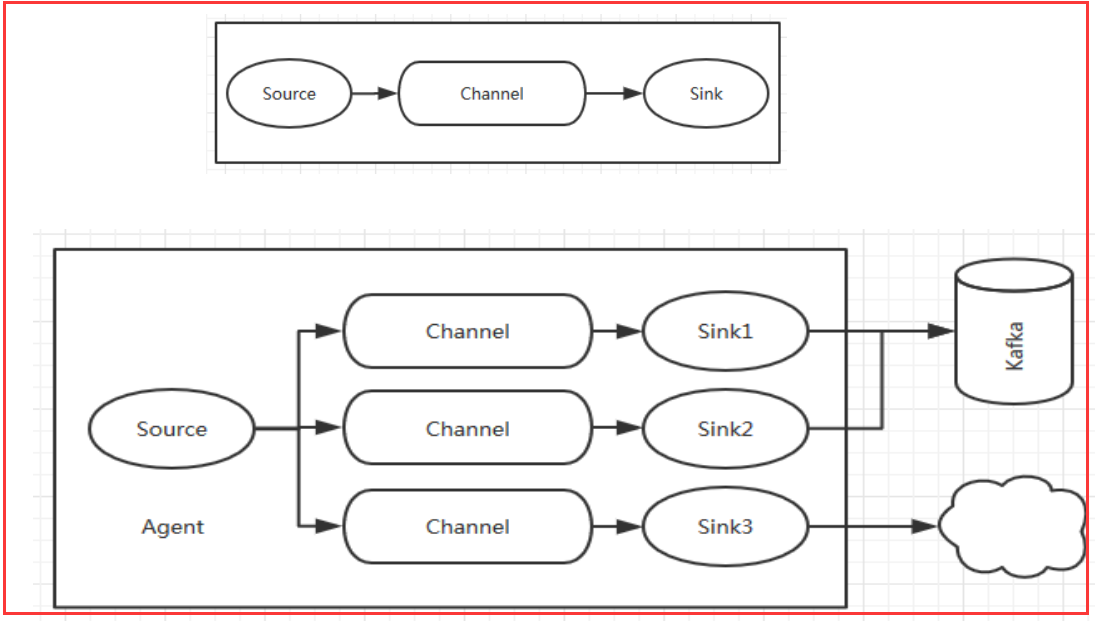
　　拦截器的位置在Source和Channel之间，当我们为Source指定拦截器后，我们在拦截器中会得到event，根据需求我们可以对event进行保留还是

　　抛弃，抛弃的数据不会进入Channel中。



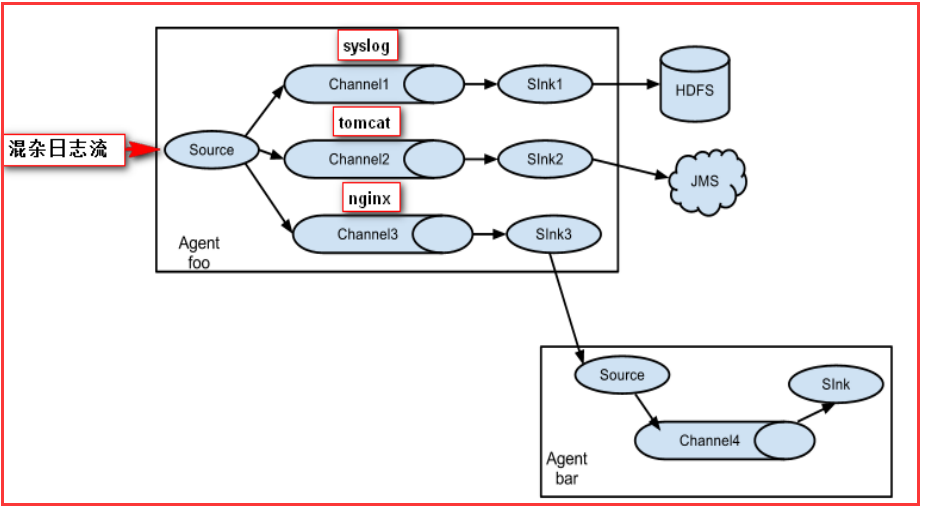
## 4.2、Flume数据流

　　1）Flume 的核心是把数据从数据源收集过来，再送到目的地。为了保证输送一定成功，在送到目的地之前，会先缓存数据，待数据真正到达目的地后，删除自己缓存的数据。  
　　2） Flume 传输的数据的基本单位是 Event，如果是文本文件，通常是一行记录，这也是事务的基本单位。 Event 从 Source，流向 Channel，再到 Sink，

本身为一个 byte 数组，并可携带 headers 信息。 Event 代表着一个数据流的最小完整单元，从外部数据源来，向外部的目的地去。  
  


　　值得注意的是，Flume提供了大量内置的Source、Channel和Sink类型。不同类型的Source,Channel和Sink可以自由组合。组合方式基于用户设置的配置文件，非常灵活。

　　比如：Channel可以把事件暂存在内存里，也可以持久化到本地硬盘上。Sink可以把日志写入HDFS, HBase，甚至是另外一个Source等等。Flume支持用户建立多级流，也就是说，多个agent可以协同工作，并且支持Fan-in、Fan-out、Contextual Routing、Backup Routes，这也正是Flume强大之处。如下图所示：



## 4.3、Flume可靠性

　　Flume 使用事务性的方式保证传送Event整个过程的可靠性。 Sink 必须在Event 被存入 Channel 后，或者，已经被传达到下一站agent里，又或者，

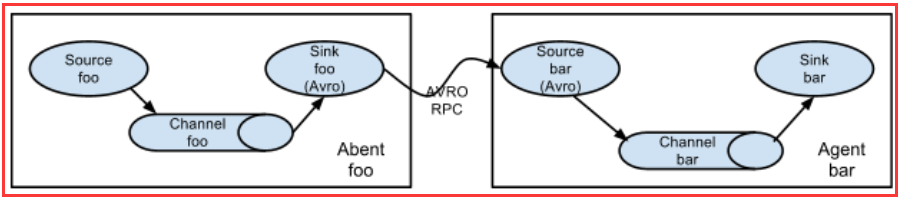
　　已经被存入外部数据目的地之后，才能把 Event 从 Channel 中 remove 掉。这样数据流里的 event 无论是在一个 agent 里还是多个 agent 之间流转，

　　都能保证可靠，因为以上的事务保证了 event 会被成功存储起来。比如 Flume支持在本地保存一份文件 channel 作为备份，而memory channel 将event存在内存 queue 里，速度快，但丢失的话无法恢复。

# 五、Flume使用场景

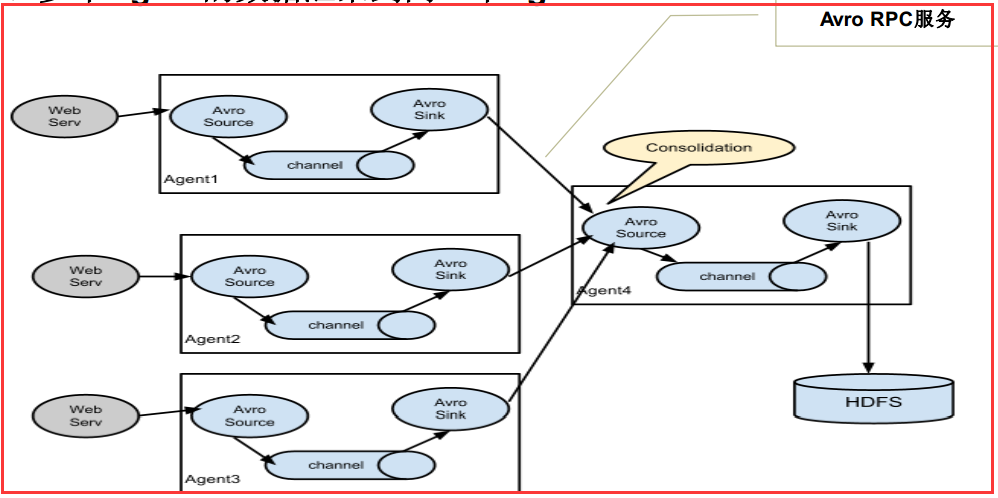
　　Flume在英文中的意思是水道， 但Flume更像可以随意组装的消防水管，下面根据官方文档，展示几种Flow。

## 5.1、多个agent顺序连接



　　可以将多个Agent顺序连接起来，将最初的数据源经过收集，存储到最终的存储系统中。这是最简单的情况，一般情况下，应该控制这种顺序连接的Agent 的数量，因为数据流经的路径变长了，如果不考虑failover的话，出现故障将影响整个Flow上的Agent收集服务。

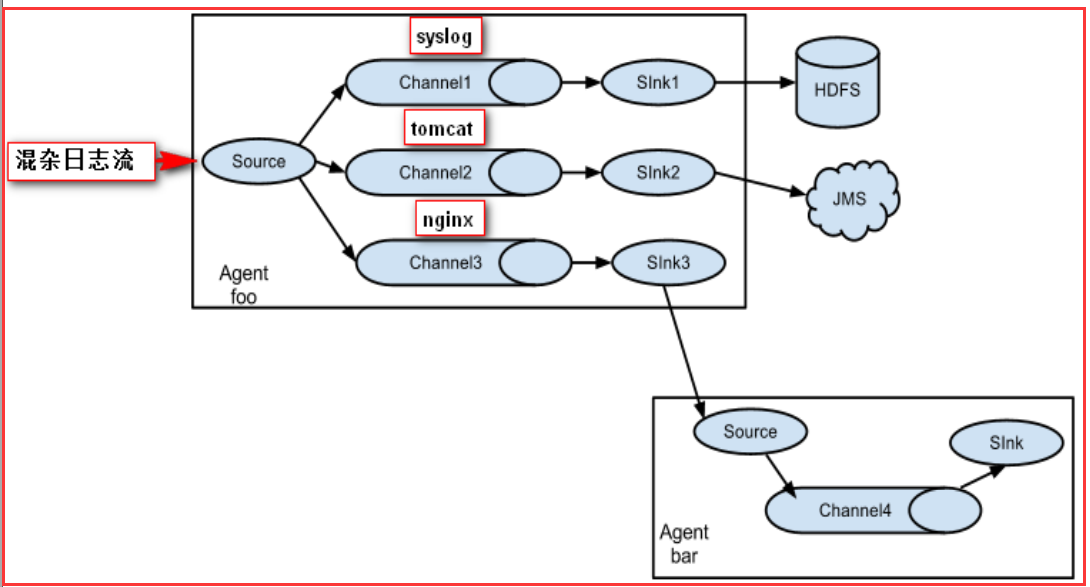
## 5.2、多个Agent的数据汇聚到同一个Agent



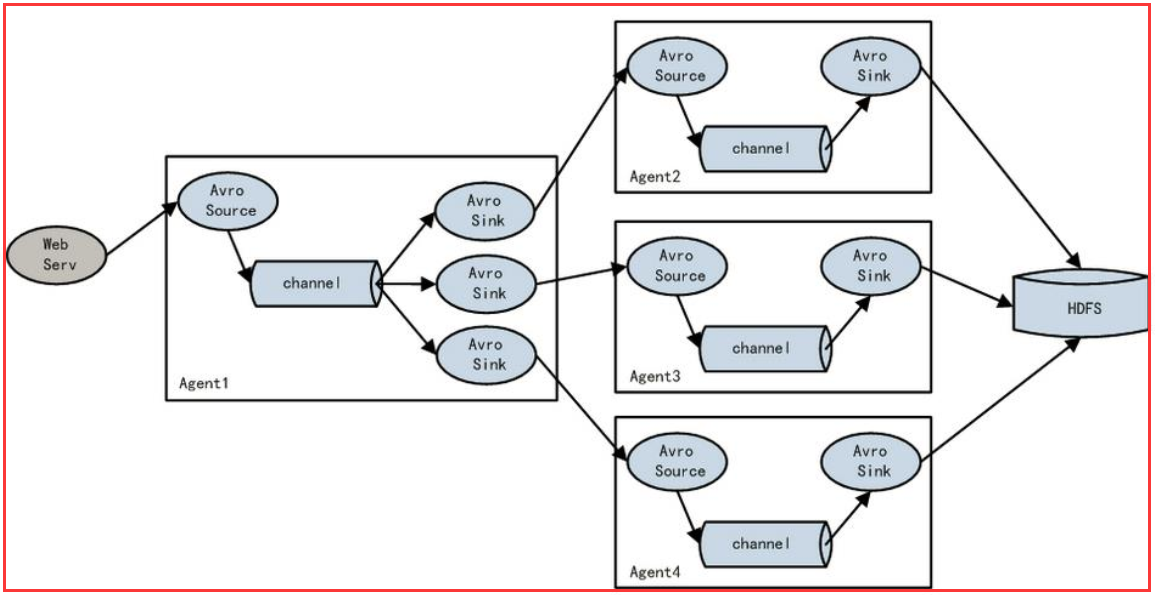
　　这种情况应用的场景比较多，比如要收集Web网站的用户行为日志， Web网站为了可用性使用的负载集群模式，每个节点都产生用户行为日志，可以为  
　　每 个节点都配置一个Agent来单独收集日志数据，然后多个Agent将数据最终汇聚到一个用来存储数据存储系统，如HDFS上。

## 5.3、多级流

　　Flume还支持多级流，什么多级流？结合在云开发中的应用来举个例子，当syslog， java， nginx、 tomcat等混合在一起的日志流开始流入一个agent  
　　后，可以agent中将混杂的日志流分开，然后给每种日志建立一个自己的传输通道。



5.4、load balance功能



　　上图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 。

# 六、Flume核心组件

　　Flume主要由3个重要的组件构成：  
　　1）Source： 完成对日志数据的收集，分成transtion 和 event 打入到channel之中  
　　　Flume提供了各种source的实现，包括Avro Source、 Exce Source、 Spooling  
　　　　Directory Source、 NetCat Source、 Syslog Source、 Syslog TCP Source、  
　　　　Syslog UDP Source、 HTTP Source、 HDFS Source， etc。  
　　2）Channel： Flume Channel主要提供一个队列的功能，对source提供中的数据进行简单的缓存。  
Flume对于Channel，提供了Memory Channel、JDBC Chanel、 File Channel，etc

　　3）Sink： Flume Sink取出Channel中的数据，进行相应的存储文件系统，数据库，或者提交到远程服务器。  
　　　　包括HDFS sink、 Logger sink、 Avro sink、 File Roll sink、 Null sink、 HBasesink， etc。

## 6.1、Source

Spool Source 如何使用？  
　　在实际使用的过程中，可以结合log4j使用，使用log4j的时候，将log4j的文件分割机制设为1分钟一次，将文件拷贝到spool的监控目录。

log4j有一个TimeRolling的插件，可以把log4j分割的文件到spool目录。基本实现了实时的监控。 Flume在传完文件之后，将会修 改文件的后缀，变为.COMPLETED（后缀也可以在配置文件中灵活指定）  
  
　　Exec Source 和Spool Source 比较  
　　1） ExecSource可以实现对日志的实时收集，但是存在Flume不运行或者指令执行出错时，将无法收集到日志数据，无法保证日志数据的完整性。  
　　2） SpoolSource虽然无法实现实时的收集数据，但是可以使用以分钟的方式分割文件，趋近于实时。  
　　3）总结：如果应用无法实现以分钟切割日志文件的话，可以两种 收集方式结合使用。

## 6.2、Channel

　　1）MemoryChannel可以实现高速的吞吐， 但是无法保证数据完整性  
　　2）MemoryRecoverChannel在官方文档的建议上已经建义使用FileChannel来替换。  
　　　　FileChannel保证数据的完整性与一致性。在具体配置不现的FileChannel时，建议FileChannel设置的目录和程序日志文件保存的目录设成不同的磁盘，以便提高效率。

## 6.3、Sink

　　Flume Sink在设置存储数据时，可以向文件系统中，数据库中， hadoop中储数据，在日志数据较少时，可以将数据存储在文件系中，并且设定一定的时间间隔保存数据。在日志数据较多时，可以将相应的日志数据存储到Hadoop中，便于日后进行相应的数据分析。

## 版本选择（Flume 1.6 +）

## 环境部署

1. java运行环境，要安装jdk1.7以上版本
2. 系统内存要足够用运行agent组件source,channel和sink。
3. 磁盘空间要足够
4. 目录权限，agent要有足够读写权限

查看: echo $JAVA\_HOME

查看: echo $PATH

# [yum安装jdk环境变量配置](https://www.cnblogs.com/benjamin77/p/8460030.html)

#安装之前先查看一下有无系统自带jdk

rpm -qa |grep java

rpm -qa |grep jdk

rpm -qa |grep gcj

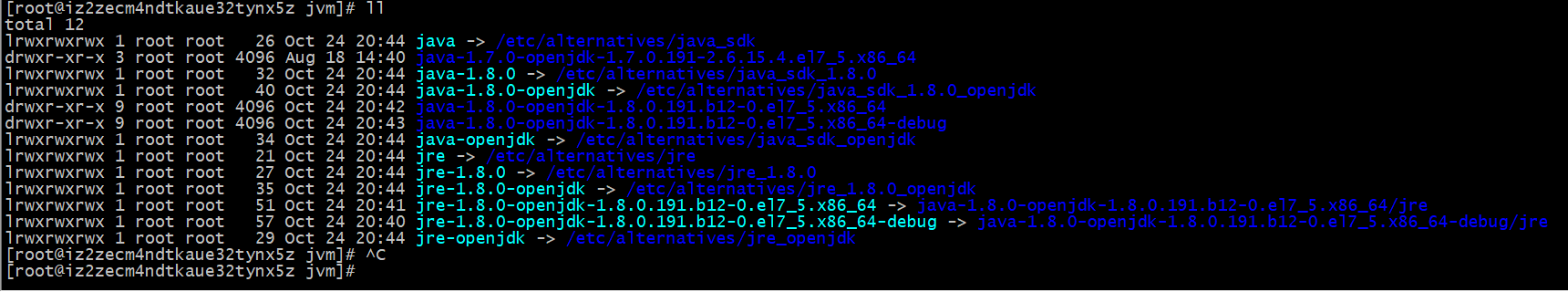
#如果有就使用批量卸载命令

rpm -qa | grep java | xargs rpm -e --nodeps

直接yum安装1.8.0版本openjdk

[root@localhost ~]# yum install java-1.8.0-openjdk\* -y

默认jre jdk 安装路径是/usr/lib/jvm 下面



用cd 命令进入到 jvm下唯一的一个目录中

/usr/lib/jvm/java-1.8.0-openjdk-1.8.0.191.b12-0.el7\_5.x86\_64

vim ~/.bashrc

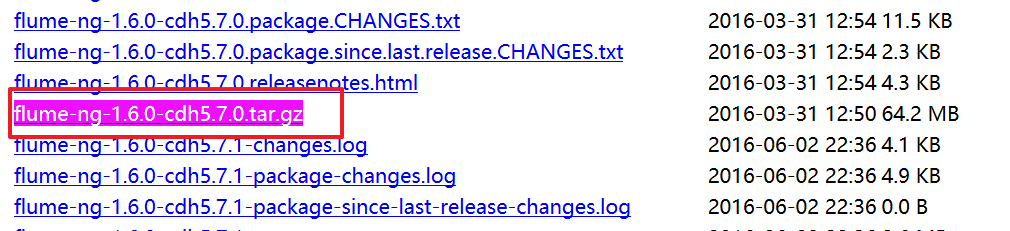
#在文件底部加入下面一句

export JAVA\_HOME=/usr/lib/jvm/java-1.8.0-openjdk-1.8.0.191.b12-0.el7\_5.x86\_64

source /etc/profile

### 安装Flume：

wget <http://archive.cloudera.com/cdh5/cdh/5/flume-ng-1.6.0-cdh5.7.0.tar.gz>



cd $FLUME\_HOME/conf

cp flume-env.sh.template flume-env.sh

修改配置：

vim flume-env.sh

添加：export JAVA\_HOME=/usr/lib/jvm/java-1.8.0-openjdk-1.8.0.191.b12-0.el7\_5.x86\_64

检测：flume-ng version

Flume 1.6.0-cdh5.7.0

Source code repository: https://git-wip-us.apache.org/repos/asf/flume.git

Revision: 8f5f5143ae30802fe79f9ab96f893e6c54a105d1

Compiled by jenkins on Wed Mar 23 11:38:48 PDT 2016

From source with checksum 50b533f0ffc32db9246405ac4431872e

[root@iz2zecm4ndtkaue32tynx5z bin]#

#### （1）从端口采集访问日志(核心是配置agent配置文件)

编写example.conf （从固定端口采集数据）

|  |
| --- |
| # example.conf: A single-node Flume configuration  # Name the components on this agent **#配置agent**  a1.sources = r1  a1.sinks = k1  a1.channels = c1  # Describe/configure the source **#配置source**  a1.sources.r1.type = netcat  a1.sources.r1.bind = localhost  a1.sources.r1.port = 44444  # Describe the sink **#配置sink**  a1.sinks.k1.type = logger  # Use a channel which buffers events in memory **#配置event**  a1.channels.c1.type = memory  #a1.channels.c1.capacity = 1000  #a1.channels.c1.transactionCapacity = 100  # Bind the source and sink to the channel **#配置channel**  a1.sources.r1.channels = c1  a1.sinks.k1.channel = c1 |

参数说明：

|  |
| --- |
| **r1 #设置source名字**  **k1 #设置sink名字，可以有多个，用逗号隔开**  **c1 #设置channel名字，可以有多个，用逗号隔开**  **a1.sources = r1 #设置source名字**  **a1.sinks = k1 #设置sink名字，可以有多个，用逗号隔开**  **a1.channels = c1 #设置channel名字，可以有多个，用逗号隔开**  **a1.sources.r1.type = netcat #设置source监控的数据类型**  **a1.sources.r1.bind = hadoop001 #设置source监控的地址**  **a1.sources.r1.port = 44444 #设置source的监控端口**  **a1.channels.c1.type = memory #设置channel的类型**  **a1.sources.r1.channels = c1 #设置source与channel的连接**  **a1.sinks.k1.channel = c1 #设置sink与channel的连接** |

将example.conf 放到 **$FLUME\_HOME/conf/**

运行Flume，在${FLUME\_HOME}/bin 下运行

|  |
| --- |
| **./flume-ng agent \**  **-n a1 \**  **-c $FLUME\_HOME/conf \**  **-f $FLUME\_HOME/conf/example.conf \**  **-Dflume.root.logger=INFO,console** |

参数说明：

**#agent 运行一个agent**

**# -n a1 ：指定agent的名字**

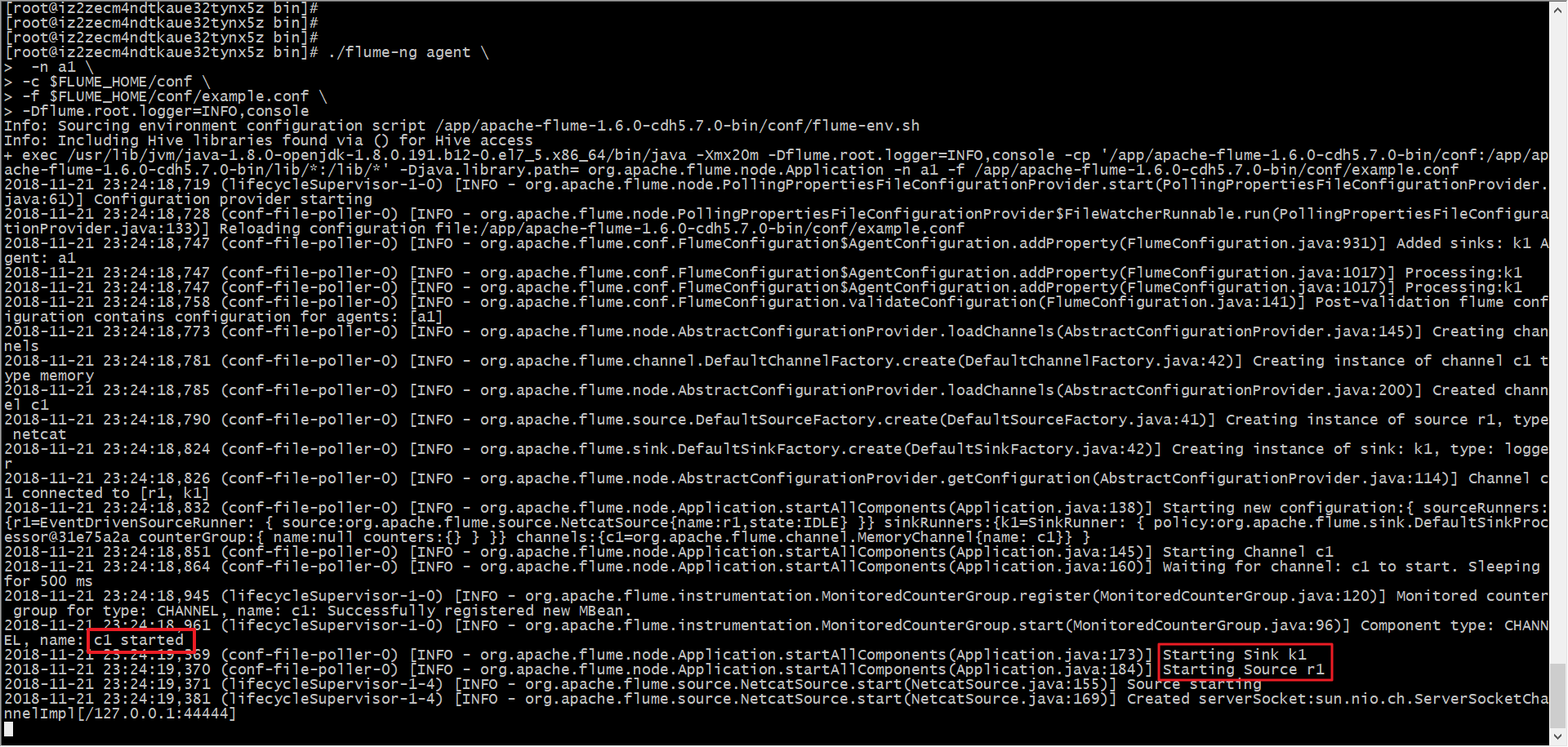
**# -c $FLUME\_HOME/conf 指定配置目录**

**# -f $FLUME\_HOME/conf/example.conf 执行agent具体的的配置文件**

**# Dflume.root.logger=INFO,console：接收数据打印到控制台**

**# example.conf：配置文件。需要自己根据flume的语法创建**

执行并观看状态:出现红框的内容表示启动成功！！！

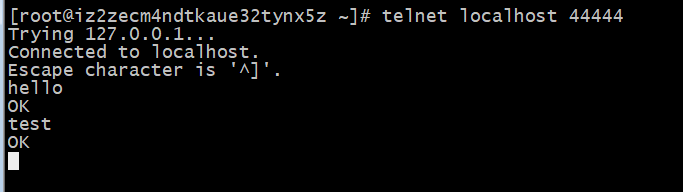


测试在另外一个窗口通过命令测试：

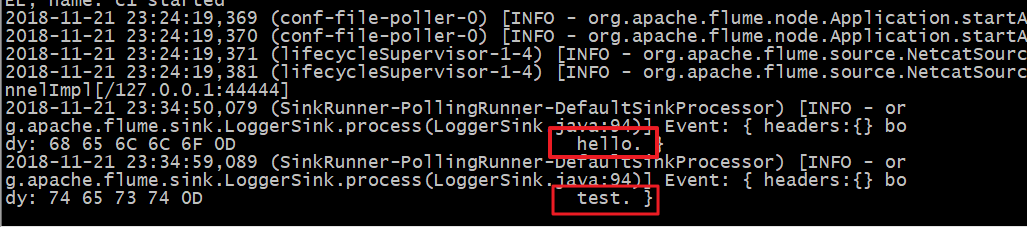
telnet localhost 44444

输入hello

输入test



在flume端可以采集到内容：



Event: { headers:{} body: 73 70 72 69 6E 67 0D spring. }

#### **（2）从日志文件采集数据**

Agent选型：exec source + memory channel +logger sink

vim exec-mem-logger.conf

|  |
| --- |
| # example.conf: A single-node Flume configuration  # Name the components on this agent #配置agent  a1.sources = r1  a1.sinks = k1  a1.channels = c1  # Describe/configure the source #配置source  **a1.sources.r1.type = exec**  **a1.sources.r1.shell = /bin/bash -c**  **a1.sources.r1.command = tail -F /data/data.log**  # Describe the sink #配置sink  a1.sinks.k1.type = logger  # Use a channel which buffers events in memory #配置event  a1.channels.c1.type = memory  #a1.channels.c1.capacity = 1000  #a1.channels.c1.transactionCapacity = 100  # Bind the source and sink to the channel #配置channel  a1.sources.r1.channels = c1  a1.sinks.k1.channel = c1 |

运行：

|  |
| --- |
| **./flume-ng agent \**  **-n a1 \**  **-c $FLUME\_HOME/conf \**  **-f $FLUME\_HOME/conf/exec-mem-logger.conf \**  **-Dflume.root.logger=INFO,console** |

新窗口测试，执行命令往data.log文件写入内容：

echo hello >> /data/data.log

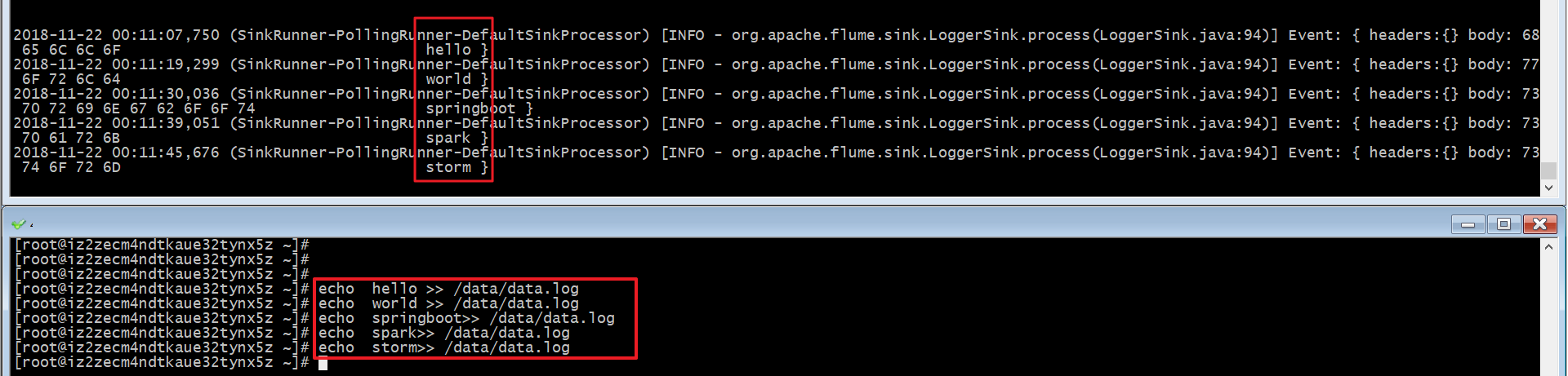
echo world >> /data/data.log

echo springboot>> /data/data.log

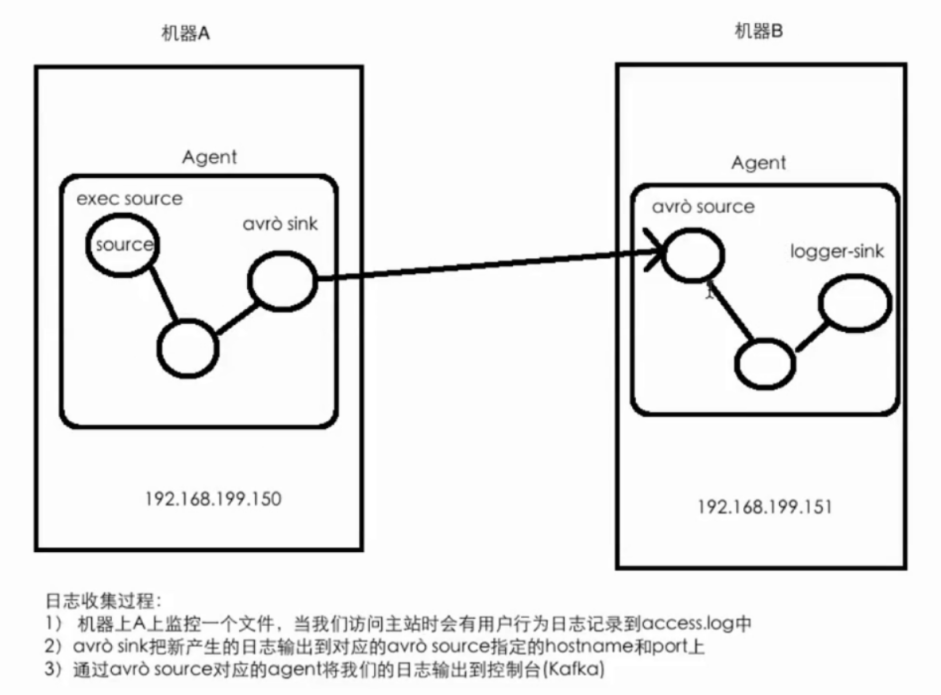
echo spark>> /data/data.log

echo storm>> /data/data.log

测试效果：



#### **（3）agent 到agent的链式的数据传输（重点）**



案例测试在单台机器上模拟伪分布式集群效果（生成环境将主机名称换成真实的ip和port）

机器A配置文件

vim exec-mem-avro.conf

|  |
| --- |
| exec-mem-avro.sources = exec-source  exec-mem-avro.sinks =avro-sink  exec-mem-avro.channels = mem-channel  exec-mem-avro**.sources.**exec-source**.type = exec**  exec-mem-avro**.sources.**exec-source**.shell = /bin/bash -c**  exec-mem-avro**.sources.**exec-source**.command = tail -F /data/data.log**  exec-mem-avro.sinks.avro-sink.type =avro  exec-mem-avro.sinks.avro-sink.hostname =localhost  exec-mem-avro.sinks.avro-sink.port =44444  exec-mem-avro.channels.mem-channel.type = memory  exec-mem-avro.sources.exec-source.channels =mem-channel  exec-mem-avro.sinks.avro-sink.channel = mem-channel |

机器B配置文件

vim avro-mem-logger.conf

|  |
| --- |
| avro-mem-logger.sources = avro-source  avro-mem-logger.sinks =logger-sink  avro-mem-logger.channels = mem-channel  avro-mem-logger**.sources.**avro-source**.type = avro**  avro-mem-logger**.sources.**avro-source**.bind=**localhost  avro-mem-logger**.sources.**avro-source**.port=** 44444  avro-mem-logger.sinks.logger-sink.type =logger  avro-mem-logger.channels.mem-channel.type = memory  avro-mem-logger.sources.avro-source.channels =mem-channel  avro-mem-logger.sinks.logger-sink.channel = mem-channel |

启动顺序：

先启动B机器，avro-mem-logger.conf

|  |
| --- |
| **./flume-ng agent \**  **-n** avro-mem-logger **\**  **-c $FLUME\_HOME/conf \**  **-f $FLUME\_HOME/conf/**avro-mem-logger.conf **\**  **-Dflume.root.logger=INFO,console** |

启动成功会有如下字样：



在启动A机器，exec-mem-avro.conf

|  |
| --- |
| **./flume-ng agent \**  **-n** exec-mem-avro **\**  **-c $FLUME\_HOME/conf \**  **-f $FLUME\_HOME/conf/**exec-mem-avro**.conf \**  **-Dflume.root.logger=INFO,console** |

启动成功会有如下字样：

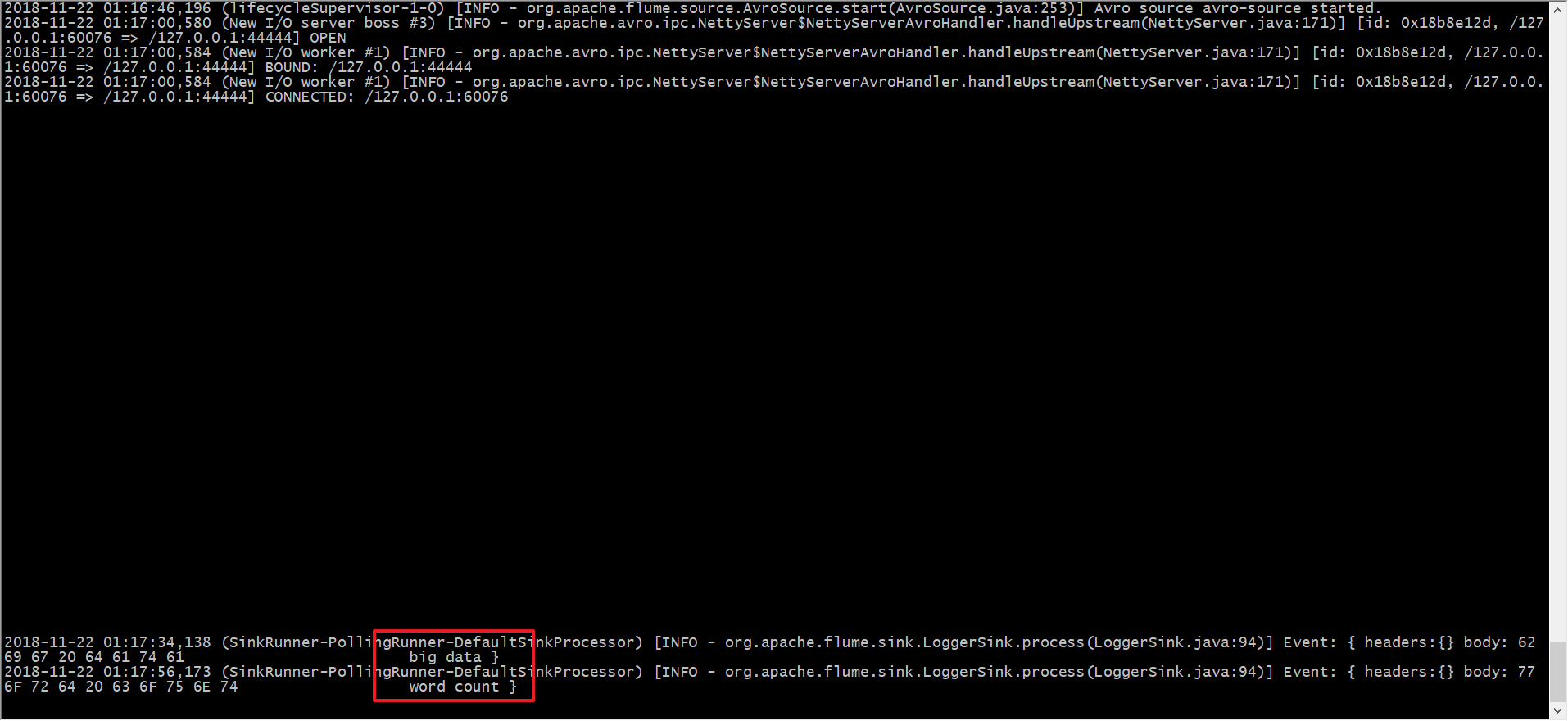


新窗口测试，执行命令往data.log文件写入内容：

echo big data >> /data/data.log

echo word count >> /data/data.log

测试效果：



总结：

Flume 完成的功能就是将分散的日志数据聚合到一起，统一发送到另外的位置进行处理或者存储。

这里只是简单模拟了一下，将流程跑通，真实开发环境中应该将flume 的 sink对接到kafka消息队列中。

### Kafka 部署及使用

参考资料：

<https://www.w3cschool.cn/apache_kafka/>

单节点单broker 部署及使用

单节点多borker 部署及使用

多节点多borker 部署及使用

1、安装ZK ：

wget <http://archive.cloudera.com/cdh5/cdh/5/zookeeper-3.4.5-cdh5.7.0.tar.gz>



2、配置环境变量：

export ZK=/app/zookeeper-3.4.5-cdh5.7.0

export PATH=$PATH:$ZK/bin

source /etc/profile

3、修改zoo.conf配置文件

cd $ZK/conf

cp zoo\_sample.cfg zoo.cfg

vim zoo.cfg

修改内容：

# tmp 目录每次系统重启都会清空，真实环境请更换路径

dataDir=/app/zk

4、为配置创建目录：

mkdir -p /app/zk

1. 启动zk

cd $ZK/bin

./zkServer.sh start

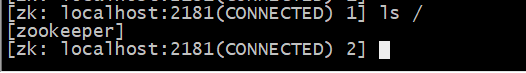
6、执行jps查看进程：

[root@iz2zecm4ndtkaue32tynx5z bin]# jps

22696 QuorumPeerMain

7、连接测试

zkCli.sh



安装kafka

wget <https://archive.apache.org/dist/kafka/0.9.0.0/kafka_2.11-0.9.0.0.tgz>

export KAFKA\_HOME=/app/kafka\_2.11-0.9.0.0

export PATH=$PATH:$KAFKA\_HOME/bin

source /etc/profile

vim $KAFKA\_HOME/config/server.properties

|  |
| --- |
| broker.id=0  listeners=PLAINTEXT://:9092  host.name=localhost  log.dirs=/app/kafka/kafka-logs  zookeeper.connect=localhost:2181 |

mkdir -p /app/kafka/kafka-logs

查看kafka版本：

find / -name \\*kafka\_\\* | head -1 | grep -o '\kafka[^\n]\*'

启动kafka：

bin/kafka-server-start.sh config/server.properties

打开两个窗口，一个作为生产者A，一个作为消费者B，

发送消息之前先有个约定：

创建topic（主题）

bin/kafka-topics.sh --create --zookeeper localhost:2181 --replication-factor 1 --partitions 1 --topic test

查看所有topic：bin/kafka-topics.sh --list --zookeeper localhost:2181

生产者：bin/kafka-console-producer.sh --broker-list localhost:9092 --topic test

消费者：bin/kafka-console-consumer.sh --zookeeper localhost:2181 --topic test --from-beginning

查看所有topic详细信息： bin/kafka-topics.sh --describe --zookeeper localhost:2181

查看指定topic详细信息：bin/kafka-topics.sh --describe --zookeeper localhost:2181 --topic test

#### 单节点多broker配置：

cp config/server.properties config/server-1.properties

cp config/server.properties config/server-2.properties

cp config/server.properties config/server-3.properties

server-1.properties

broker.id=1

     listeners=<PLAINTEXT://:9093>

     log.dirs=/tmp/kafka-logs-1

server-2.properties

broker.id=2

     listeners=[PLAINTEXT://:9094](PLAINTEXT://:9093)

     log.dirs=/tmp/kafka-logs-2

server-3.properties

broker.id=3

     listeners=[PLAINTEXT://:9095](PLAINTEXT://:9093)

     log.dirs=/tmp/kafka-logs-3

启动：

bin/kafka-server-start.sh -daemon config/server-1.properties &

bin/kafka-server-start.sh -daemon config/server-2.properties &

bin/kafka-server-start.sh -daemon config/server-3.properties &

创建topic：

bin/kafka-topics.sh --create --zookeeper localhost:2181 --replication-factor 3 --partitions 1 --topic my-replicated-topic

查看所有topic详细信息： bin/kafka-topics.sh --describe --zookeeper localhost:2181

查看指定topic详细信息: bin/kafka-topics.sh --describe --zookeeper localhost:2181 --topic my-replicated-topic

### Scala 操作kafka API

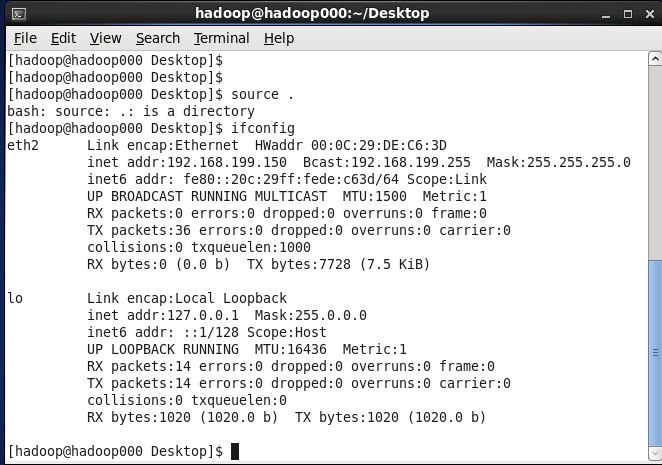
虚拟机用户名： hadoop/hadoop

##### CentOS 7更改网卡名eth0和配置网卡

sudo vi /etc/sysconfig/network-scripts/ifcfg-eth0 为：

    （提示：按a键进入insert模式，按esc后，按“：wq”保存退出）

|  |
| --- |
| DEVICE="eth0"  BOOTPROTO="static"  ONBOOT="yes"  TYPE="Ethernet"  IPADDR=192.168.199.150  GATEWAY=192.168.199.1  DNS1=8.8.8.8  DNS2=192.168.1.1  NETMASK=255.255.255.0 |



##### CentOS中如何将网卡eth2更改为eth0

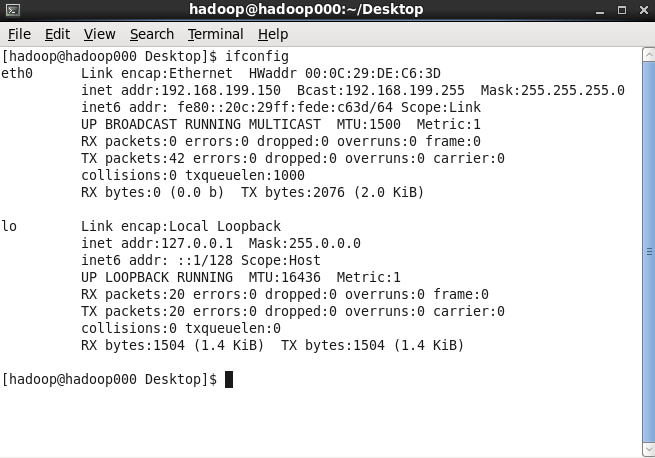
在Linux中，还有一个配置文件：/etc/udev/rules.d/70-persistent-net.rules，打开这个配置文件后，会发现里面列出了所有Linux找到的网卡，包括历史的。因此，解决无法加载网卡的问题就有两个解决办法：

1.修改interfaces配置文件，使其使用最新的eth网卡

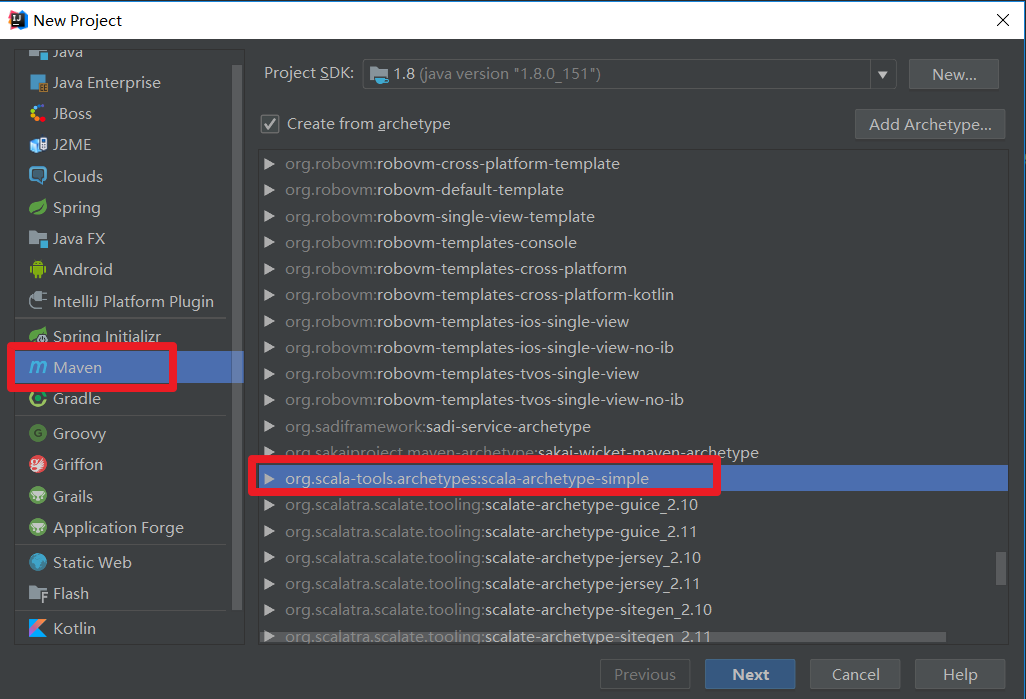
2.删除.rules文件，让linux重新将网卡认为是eth0(需要重启linux系统)

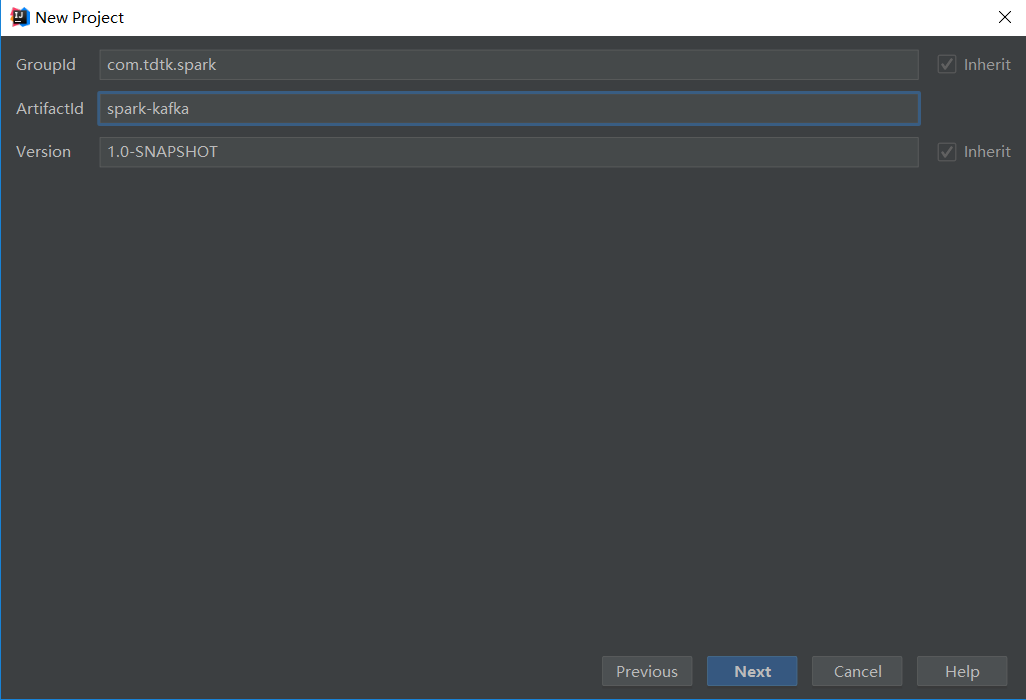
sudo rm -rf /etc/udev/rules.d/70-persistent-net.rules

删除后执行reboot重启系统。

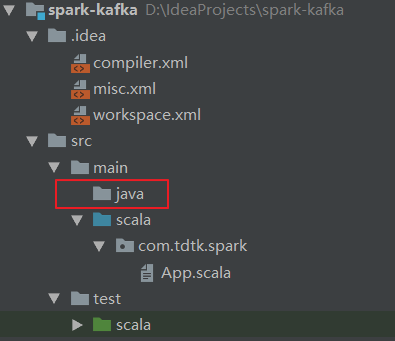


#### 创建工程





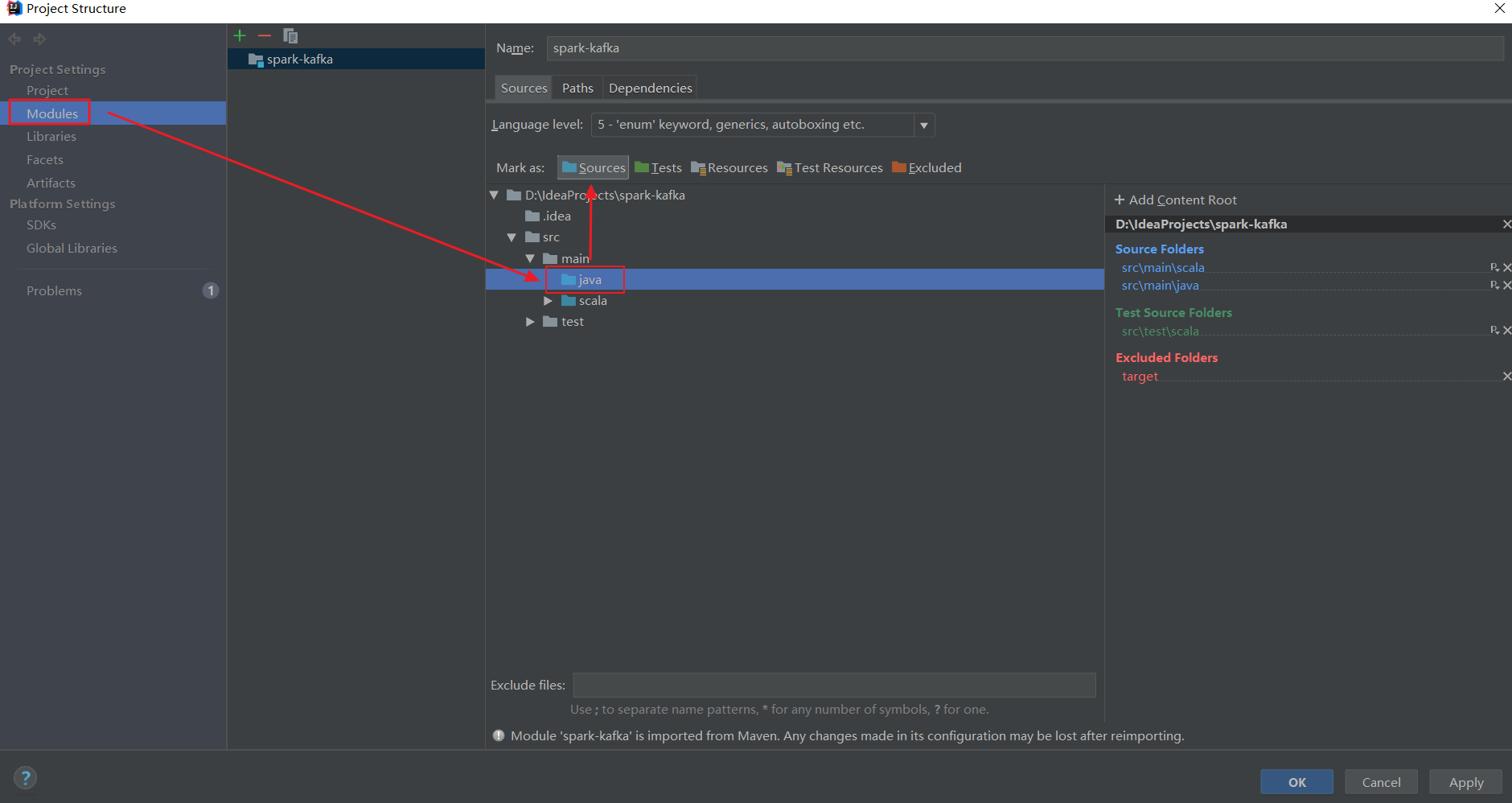
创建java目录：



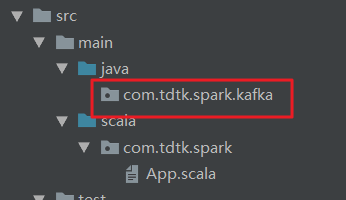
点击右上角图标，打开project structure窗口



将目录设置为source



创建包名：com.tdtk.spark.kafka



在修改本地开发环境所在主机的hosts文件

|  |
| --- |
| 192.168.199.150 kafka.server #kafka IP  192.168.199.150 zk.server #zookeeper IP |

**POM配置：**

|  |
| --- |
| <dependency>  <groupId>org.apache.kafka</groupId>  <artifactId>kafka\_2.11</artifactId>  <version>0.9.0.0</version> </dependency> |

**Kafka常用配置文件**

|  |
| --- |
| package com.tdtk.spark.kafka; */\*\*  \* Kafka常用配置文件  \*/* public class KafkaProperties {   public static final String *ZK* = "zk.server:2181";   public static final String *TOPIC* = "test\_topic";   public static final String *BROKER\_LIST* = "kafka.server:9092";   public static final String *GROUP\_ID* = "test\_group\_01";  } |

生产者：

|  |
| --- |
| package com.tdtk.spark.kafka;  import org.apache.kafka.clients.producer.KafkaProducer; import org.apache.kafka.clients.producer.Producer; import org.apache.kafka.clients.producer.ProducerRecord;  import java.util.Properties;   */\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  \*\*\*\*\*\** ***@ClassName*** *: ProducerDemo.java 生产者  \*\*\*\*\*\** ***@author*** *: zl ^ ^  \*\*\*\*\*\** ***@date*** *: 2018 11 23 02:30  \*\*\*\*\*\** ***@version*** *: v1.0.x  \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/* public class ProducerDemo {   public static void main(String[] args){  Properties properties = new Properties();  properties.put("bootstrap.servers", KafkaProperties.*BROKER\_LIST*);//"localhost:9092"  properties.put("acks", "all");  properties.put("retries", 0);  properties.put("batch.size", 16384);  properties.put("linger.ms", 1);  properties.put("buffer.memory", 33554432);  properties.put("key.serializer", "org.apache.kafka.common.serialization.StringSerializer");  properties.put("value.serializer", "org.apache.kafka.common.serialization.StringSerializer");  Producer<String, String> producer = null;  try {  producer = new KafkaProducer<String, String>(properties);  for (int i = 0; i < 10000; i++) {  String msg = "Message " + i;  producer.send(new ProducerRecord<String, String>(KafkaProperties.*TOPIC*, msg));  System.*out*.println("Sent:" + msg );  }  } catch (Exception e) {  e.printStackTrace();   } finally {  producer.close();  }   } } |

消费者：（接收消息由于用的阿里云服务器，会有延迟，不要着急，等一会儿就会接收到消息了）

|  |
| --- |
| package com.tdtk.spark.kafka;  import org.apache.kafka.clients.consumer.Consumer; import org.apache.kafka.clients.consumer.ConsumerRecord; import org.apache.kafka.clients.consumer.ConsumerRecords; import org.apache.kafka.clients.consumer.KafkaConsumer;  import java.io.IOException; import java.util.Arrays; import java.util.Collection; import java.util.List; import java.util.Properties;  */\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  \*\*\*\*\*\** ***@ClassName*** *: ConsumerDemo.java 消费者  \*\*\*\*\*\** ***@author*** *: zl ^ ^  \*\*\*\*\*\** ***@date*** *: 2018 11 23 02:30  \*\*\*\*\*\** ***@version*** *: v1.0.x  \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/* public class ConsumerDemo {  public static void main(String[] args) throws IOException {  Properties props = new Properties();  // 老版本  //props.put("zookeeper.connect", "localhost:2181");  //0.9 以后新版本  props.put("bootstrap.servers", KafkaProperties.*BROKER\_LIST*);//"localhost:9092"   props.put("group.id",KafkaProperties.*TOPIC*);  // 自动确认设置  //0.9 以后新版本  props.put("enable.auto.commit", "true");  // props.put("enable.auto.commit", "false");  props.put("auto.commit.interval.ms", "1000");  props.put("session.timeout.ms", "30000"); //30000和不写都正常，改成40000就启动报错   //老版本 // props.put("zookeeper.session.timeout.ms", "30000"); // props.put("zookeeper.sync.time.ms", "200"); // props.put("auto.commit.interval.ms", "1000");   props.put("key.deserializer", "org.apache.kafka.common.serialization.StringDeserializer");  props.put("value.deserializer", "org.apache.kafka.common.serialization.StringDeserializer");  Consumer<String, String> consumer = new KafkaConsumer<String, String>(props);  Collection<String> topics = Arrays.*asList*(KafkaProperties.*TOPIC*);  //消费者订阅topic  consumer.subscribe((List<String>) topics);  ConsumerRecords<String, String> consumerRecords = null;  while (true) {  //接下来就要从topic中拉取数据  consumerRecords = consumer.poll(1000);  //遍历每一条记录  for (ConsumerRecord<String, String> consumerRecord : consumerRecords) {  long offset = consumerRecord.offset();  int partition = consumerRecord.partition();  Object key = consumerRecord.key();  Object value = consumerRecord.value();  System.*out*.println(offset + " " + partition + " " + key + " " + value);  }  }  }  } |

#### 异常：（最简单的办法是换zk和kafka的新版本）

log4j:WARN No appenders could be found for logger (kafka.utils.VerifiableProperties).

log4j:WARN Please initialize the log4j system properly.

**Exception in thread "Thread-0" kafka.common.FailedToSendMessageException: Failed to send messages after 3 tries.**

at kafka.producer.async.DefaultEventHandler.handle(DefaultEventHandler.scala:91)

at kafka.producer.Producer.send(Producer.scala:77)

at kafka.javaapi.producer.Producer.send(Producer.scala:33)

at com.tdtk.spark.kafka.KafkaProducer.run(KafkaProducer.java:39)

Process finished with exit code 0

host.name与advertised.host.name的设置

host.name：按配置文件说明，是Kafka绑定的interface。其实这个说明有点误导，下面会见到。

advertised.host.name：是注册到zookeeper，client要访问的broker地址。（可能producer也是拿这个值，没有验证）

如果advertised.host.name没有设，会用host.name的值注册到zookeeper，如果host.name也没有设，则会使用JVM拿到的本机hostname注册到zk。

这里有两个坑要注意：

如果advertised.host.name没有设，host.name不能设为0.0.0.0，否则client通过zk拿到的broker地址就是0.0.0.0。  
如果指定要bind到所有interface，host.name不设就可以。

如果host.name和advertised.host.name都不设，client通过zk拿到的就是JVM返回的本机hostname，如果这个hostname是client无法访问到的，client就会连不上broker。  
所以如果要bind到所有interface，client又能访问，解决的办法是host.name不设或设置0.0.0.0，advertised.host.name一定要设置为一个client可以访问的地址，如直接设IP地址。  
如果不需要bind到所有interface，也可以只在host.name设置IP地址。

简单的检查broker是否可以被client访问到的办法，就是在zookeeper中看broker信息，上面显示的hostname是否是client可以访问到的地址。  
**外网无法访问：**

**在zkCli中执行ls /brokers/topics**

**配置如下：**

0.9版本及以前，修改如下配置：

host.name=0.0.0.0

listeners=PLAINTEXT://0.0.0.0:9092

advertised.host.name=外网IP

advertised.port=9092

新版本0.10.x broker配置弃用了advertised.host.name和 advertised.port这两个配置项，使用advertised.listeners替换advertised.host.name和 advertised.port这两个配置项

**新版配置：**advertised.listeners=PLAINTEXT://59.64.11.22:9092

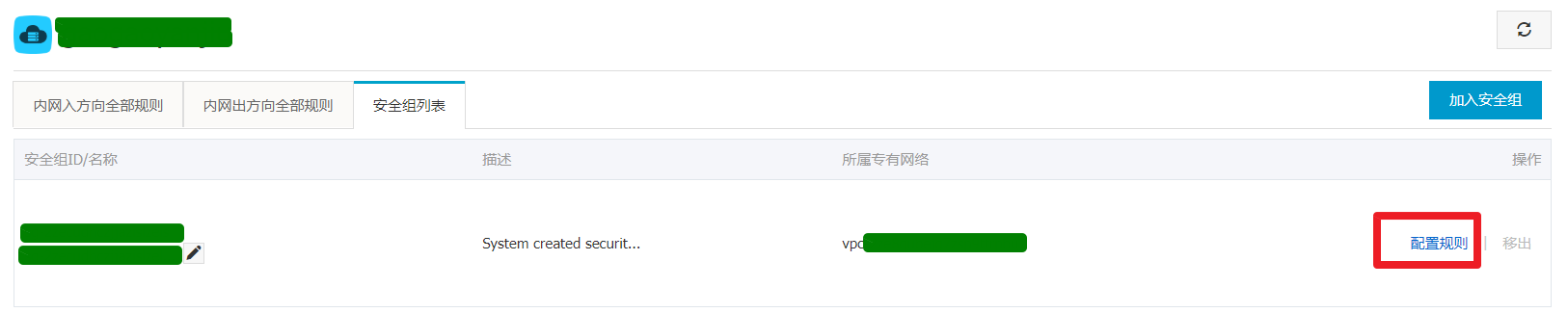


# 2.阿里云ECS外网不能连接访问ZooKeeper解决

1.进入阿里云ECS实例控制台



1. 选择更多-安全组配置   
   
2. 选择配置规则

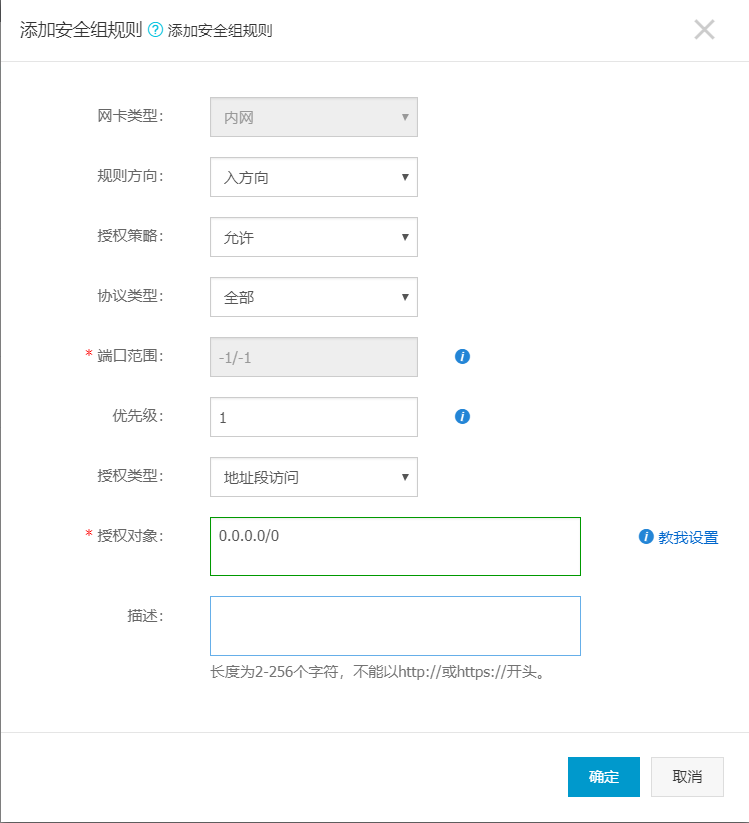


4.选择添加安全组规则



5.添加安全组规则

可根据需要选择协议类型及设置对应端口范围，为了省事，我选择协议类型为全部。点击确定，安全组规则设置完成。



异常2：can't rebalance after 4 retries

4次重试后无法再平衡

这个问题就是连接不上主机

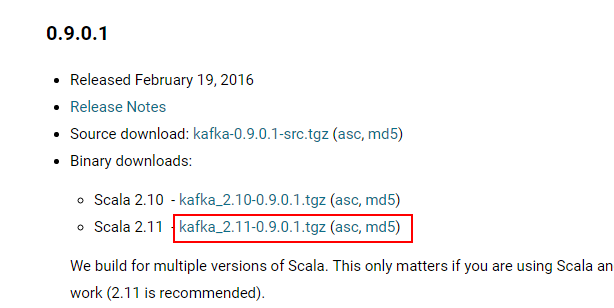
### **[Kafka集群搭建 (2.11-0.9.0.1)](https://www.cnblogs.com/freeweb/p/7380492.html)**

　　之前写过kafka\_2.9.2-0.8.2.2版本的安装，kafka在新的0.9版本以上改动比较大，配置和api都有很大更新，并且broker对应的partition支持多线程生产和消费，所以性能比之前好得多，比如老版本的kafka单机每秒可以推送100条数据，但是新版的可以每秒推送达到上千条数据，多节点的性能提升非常大，下面是具体的安装过程

　　访问Apache Kafka官网下载安装包，地址：http://kafka.apache.org/

　　点击download按钮，进入版本选择，这里选择0.9.0.1版本的基于Scala 2.11的kafka\_2.11-0.9.0.1.tgz安装包

　　注意不要安装0.9.0.0的版本，这个版本存在问题，并且已经在0.9.0.1中得到修复



　　安装kafka集群之前，确保zookeeper服务已经正常运行，这里3台zookeeper准备工作都已完成，三台主机分别为：linux1,linux2,linux3，接下来在linux1主节点上执行释放并做软链：

tar -xvzf kafka\_2.11-0.9.0.1.tgz

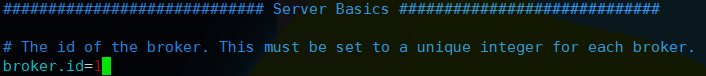
mv kafka\_2.11-0.9.0.1 /bigdata/

cd /bigdata/ln -s kafka\_2.11-0.9.0.1 kafka

cd kafka

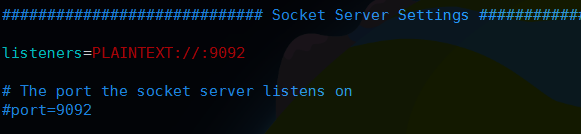
　　接下来执行 vim config/server.properties 编辑配置文件

　　修改broker.id=1，默认是0



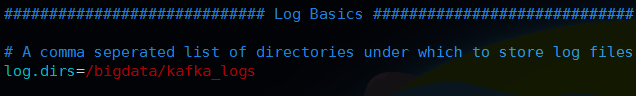
　　这个值是集群中唯一的一个整数，每台机器各不相同，这里linux1设置为1其他机器后来再更改

　　然后往下找到listeners这个配置项一般配置PLAINTEXT://ip:9092，如果配置0.0.0.0则绑定全部网卡，如果默认像下面这样，kafka会绑定默认的所有网卡ip，一般在机器中hosts，hostname都要正确配置，这里默认即可；然后下面的port默认不用配置，所以这块配置和老版本不同

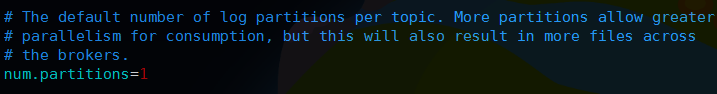


　　注意另外有一些advertised开头的参数，这些参数可以提供客户端访问的配置，比如advertised.listeners会覆盖listeners配置效果和listeners配置完全一样，但是客户端访问时会通过配置的参数访问zookeeper，另外还有advertised.host.name参数，这个也是指定客户端获取元数据的zookeeper主机，默认情况下如果这些参数不配置，那么客户端访问时hosts必须配置kafka集群的地址映射，否则直接会找不到对应的主机，但是advertised.host.name配置之后，客户端可以直接使用ip地址就可以生产或消费kafka，kafka会自动返回advertised.host.name的值供客户端使用，注意这里advertised.host.name一定要设置为ip地址，而不是主机名因为kafka会原样返回；这些就是advertised参数的区别和用途，生产环境中根据需要进行配置即可

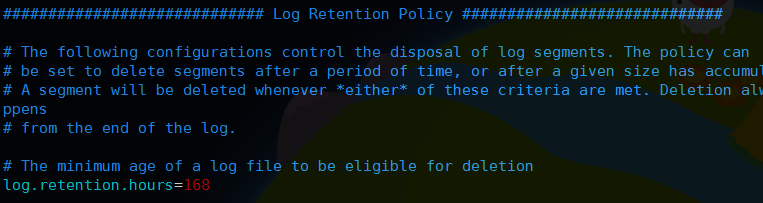
　　然后配置kafka日志目录，注意目录要提前建好



　　然后下面num.partitions是默认单个broker上的partitions数量，默认是1个，如果想提高单机的并发性能，这里可以配置多个

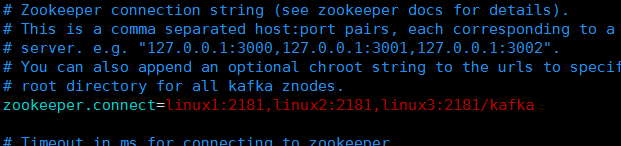


　　然后是kafka日志的保留时间，单位小时，默认是168小时，也就是7天

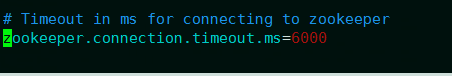


　　注意之前有个log.cleaner.enable表示是否清理日志，这个配置在新版本已经废弃了，也就是日志必须是定时清理的，仅仅通过上面的保留时间参数来控制

　　然后设置协调的zookeeper集群列表，然后指定了Kafka在zookeeper上创建的znode为/kafka，



　　最后一项配置，默认即可



　　这个表示连接zookeeper服务器的超时时间，以上设置都完毕，保存配置并退出，然后将kafka目录发送至其他主机

scp -r kafka\_2.11-0.9.0.1 linux2:/bigdata/scp -r kafka\_2.11-0.9.0.1 linux3:/bigdata/

　　这样就发送到了linux2和linux3这两台主机，然后依次修改linux2和linux3中config/server.properties配置文件中broker.id分别为2和3并保存

　　最后对三个节点都要创建日志目录： mkdir /bigdata/kafka\_logs 并且根据需要创建软链接，完成之后kafka集群就安装完毕了，

　　然后启动所有主机的kafka服务，分别进入kafka目录，执行下面命令启动服务：

bin/kafka-server-start.sh -daemon config/server.properties

　　新版本的kafka无需使用nohup挂起，直接使用-daemon参数就可以运行在后台，启动后通过jps查看有Kafka进程就启动成功，对于创建topic，生产，消费操作和之前基本都是一样的，停止同样执行bin/kafka-server-stop.sh即可

## 在0.9版本消费者新API中，有3种ack策略

### 1.自动提交

就是上面示例中的，程序只要获取到消息就会提交offset，不管你的方法有没有执行完。同时消费速度是最快的

props.put("enable.auto.commit", "true");

props.put("auto.commit.interval.ms", "1000");

### 2.手动提交

手动提交就是在你的方法执行完后手动调用consumer.commitSync();手动同步ack。每处理完1条消息，commitSync 1次  
速度慢，测试手动commit耗时在0.1秒左右

props.put("enable.auto.commit", "false");

// 执行完方法后调用consumer.commitSync();

### 3.手动异步提交

手动异步提交跟手动提交方式一样，只是加了异步，速度会有所提升

props.put("enable.auto.commit", "false");

// 执行完方法后调用consumer.commitAsync();

参考资料：

<http://www.cnblogs.com/dennyzhangdd/p/7759876.html>

Intellij IDEA 将工程转换成maven工程

项目上右键 Add Framework Support，选择maven