1 开发手册

1.1内部域名访问

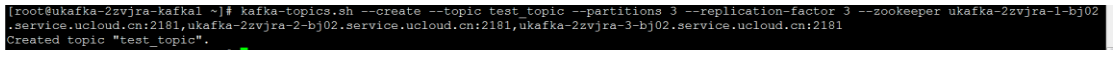
内部域名主要为了解决用户集群需要做迁移或者变更时，可以通过后台变更域名的方法避免 consumer 和 producer 的修改。
在用户申请集群后，为每个 UKafka 节点自动注册一个域名，规则为[集群 ID]-[节点编号]-[可用区编号].service.ucloud.cn
以北京可用区 B 的资源 id 为 ukafka-2zvjra 集群为例，各节点可通过 ukafka-2zvjra-*-bj02.service.ucloud.cn 访问
zookeeper 配置为
ukafka-2zvjra-1-bj02.service.ucloud.cn:2181
ukafka-2zvjra-2-bj02.service.ucloud.cn:2181
ukafka-2zvjra-3-bj02.service.ucloud.cn:2181

内部域名详细介绍请参考：https://docs.ucloud.cn/network/udee/index.html

1.2客户端示例

创建 topic

kafka-topics.sh --create --topic test_topic --partitions 3 --replication-factor 3 --zookeeper ukafka-2zvjra-1-bj02.service.ucloud.cn:2181,ukafka-2zvjra-2-bj02.service.ucloud.cn:2181,ukafka-2zvjra-3-bj02.service.ucloud.cn:2181



发送消息

kafka-console-producer.sh --broker-list ukafka-2zvjra-1-bj02.service.ucloud.cn:9092,ukafka-2zvjra-2-bj02.service.ucloud.cn:9092,ukafka-2zvjra-3-bj02.service.ucloud.cn:9092 --topic test_topic

```
[root@ukafka-2zvjra-kafka1 ~]# kafka-console-producer.sh --broker-list ukafka-2zvjra-1-bj02.service.ucloud.cn:9092,ukafka-2zvjra-2-bj02.service.ucloud.cn:9092,ukafka-2zvjra-3-bj02.service.ucloud.cn:9092 --topic test_topic
[2016-05-11 11:32:27,501] WARN Property topic is not valid (kafka.utils.VerifiableProperties)
hello kafka
```

接收消息

kafka-console-consumer.sh --zookeeper ukafka-2zvjra-1-bj02.service.ucloud.cn:2181,ukafka-2zvjra-2-bj02.service.ucloud.cn:2181,ukafka-2zvjra-3-bj02.service.ucloud.cn:2181 --topic test_topic --from-beginning

```
[root@ukafka-2zvjra-kafka1 ~]# kafka-console-consumer.sh --zookeeper ukafka-2zvjra-1-bj02.service.ucloud.cn:2181,ukafka-2zvjra-2-bj02.service.ucloud.cn:2181,ukafka-2zvjra-3-bj02.service.ucloud.cn:2181 --topic test_topic --from-beginning
f
hello kafka
```

我们可以看到上一条命令发送的内容，我们这里使用的是 from-beginning，所以会显示当前 topic 中的所有内容。

其他客户端请参考：https://cwiki.apache.org/confluence/display/KAFKA/Clients

1.3外网访问 UKafka

路由器配置方法

由于目前没有为 ukafka 节点单独绑定外网 IP，所以目前只能使用路由转发的方式来获取外网访问权限。

为 kafka 集群的 broker 设置一个外网路由规则。

一个配置示例：使用 TCP 协议，源 ip 是绑定给当前 router 的 eip，目标 IP 是 kafka 集群的内网 ip。

路由详细设置方法请参考：https://docs.ucloud.cn/network/unet/common.html#id8

端口映射：

端口	服务
9092	Broker
2181	Zookeeper
9000	Kafka-manager

修改集群配置

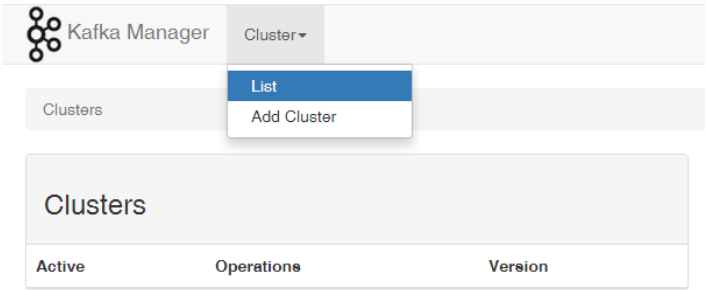
broker 的~/kafka/config/server.properties

advertised.host.name=上面配置的外网 ip

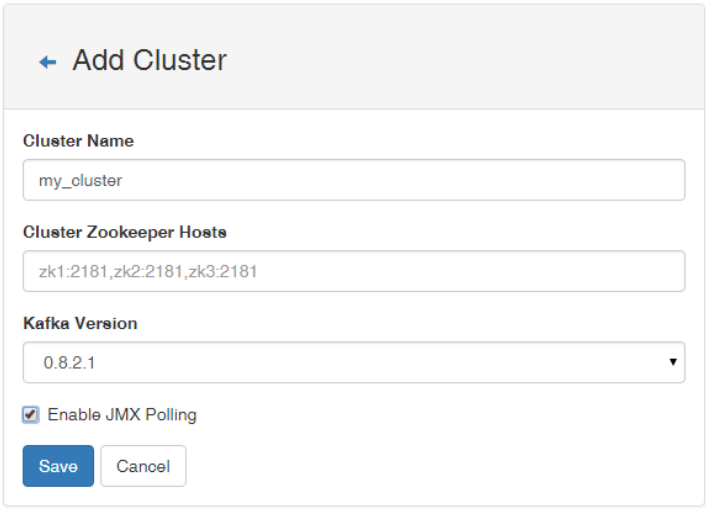
这个值是 broker 回给 consumer 和 producer 的访问地址，默认是 host.name 配置的地址。

1.4Kafka-manager 设置

通过 http://EIP:9000 可以看到下面的内容



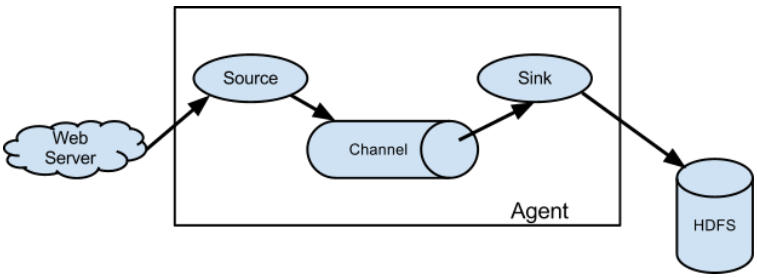
添加 zookeeper 地址（域名+port），选择 kafka 版本（可选 0.8.2.1），并勾选 jmx 监控。



Kafka-manager 更多用法和指引请参考：https://github.com/yahoo/kafka-manager

1.5使用 Flume+UKafka

Flume 简介



FlumeEvent

Flume event 是 flume 自定义的传输数据的格式。

Agent

启动的一个 flume 进程.

Source

Flume Agent 的 Source 用于接收外部数据源发送过来的数据(例如上例中的 Web Server)，注意的是，外部数据源发送的数据必须满足 Agent Source 定义的数据源的格式。比如对于 Avro Source，那么这个 Source 仅接收 Avro 格式的数据，外部数据源可以通过 Avro Client 的方式给 Agent 的 Source 发送数据；Avro Source 也可以接收其它 Agent 的 Avro Sink 发送的 Avro Event 数据。

Channel

Agent Channel 是 Agent Source 接收到数据的一个缓冲，数据在被消费前(写入到 Sink)对读取到的数据进行缓冲。一个 source 可以写到多个 channel。

Sink

从 channel 的缓冲中取出数据，存储到最终流向的地方。

下载安装

http://apache.fayea.com/flume/1.6.0/apache-flume-1.6.0-bin.tar.gz

$ tar -xf apache-flume-1.6.0-bin.tar.gz

修改配置文件

apache-flume-1.6.0-bin/conf/flume-env.sh

```
export JAVA_OPTS="-Xms100m -Xmx1024m -Dcom.sun.management.jmxremote
"
```

抓取本地日志上传到 UKafka

配置文件：flume-conf.properties.sink.kafka

```
# 自定义一个 source channel 和 sink
agent.sources = seqGenSrc
agent.channels = memoryChannel
agent.sinks = kafkaSink
# source 的来源通过 unix 命令方式获取
# 参考 https://flume.apache.org/FlumeUserGuide.html#exec-source
agent.sources.seqGenSrc.type = exec
agent.sources.seqGenSrc.command = tail -f /tmp/access.log

# 为 source 绑定一个 channel.
agent.sources.seqGenSrc.channels = memoryChannel

# sink source 的内容到 kafka
# 详细内容请参考 https://flume.apache.org/FlumeUserGuide.html#kafka-sink
agent.sinks.kafkaSink.type = org.apache.flume.sink.kafka.KafkaSink
agent.sinks.kafkaSink.topic = flume_kafka_sink
# kafka brokers 地址
agent.sinks.kafkaSink.brokerList = ukafka-ytqpg4-kafka1:9092,ukafka-ytqpg4-kafka
2:9092,ukafka-ytqpg4-kafka3:9092
agent.sinks.kafkaSink.batchSize = 20
agent.sinks.kafkaSink.partition.key=region
agent.sinks.kafkaSink.partitioner.class=org.apache.flume.plugins.SinglePartition

# 为 sink 绑定一个 channel.
agent.sinks.kafkaSink.channel = memoryChannel

# channel 的配置，将 Source 接收到数据的一个缓冲到内存中。
# 详细说明请参考 https://flume.apache.org/FlumeUserGuide.html#memory-channel
```

```
agent.channels.memoryChannel.type = memory
agent.channels.memoryChannel.capacity = 10000
agent.channels.memoryChannel.transactionCapacity = 1500
```

启动命令

./bin/flume-ng agent -n agent -c conf -f conf/flume-conf.properties.sink.kafka

查看结果

在 kafka 集群上执行 kafka-console-consumer.sh --zookeeper ukafka-ytqpg4-kafka1:2181 --topic flume_kafka_sink
可以看到上传的日志内容

同步 UKafka 数据到 hdfs

准备 jar 包

由于我们 hadoop 集群使用的 jar 包是这个，hadoop-hdfs-2.6.0-cdh5.4.9.jar，所以需要从集群上面拷贝 jar 包到 apache-flume-1.6.0-bin/lib 目录下

配置文件: flume-conf.properties

```
agent.sources = seqGenSrc
agent.channels = memoryChannel
agent.sinks = hdfsSink
#  source 的来源通过 kafka 获取
# 请参考 https://flume.apache.org/FlumeUserGuide.html#kafka-source
agent.sources.seqGenSrc.type = org.apache.flume.source.kafka.KafkaSource
# kafka zookeeper 地址
agent.sources.seqGenSrc.zookeeperConnect = ukafka-ytqpg4-kafka1:2181,ukafka-yt
```

```
qpg4-kafka2:2181,ukafka-ytqpg4-kafka3:2181
agent.sources.seqGenSrc.topic = flume_kafka_sink
agent.sources.seqGenSrc.groupId = flume
agent.sources.seqGenSrc.interceptors = i1
agent.sources.seqGenSrc.interceptors.i1.type = timestamp
agent.sources.seqGenSrc.kafka.consumer.timeout.ms = 100
# 为 soure 绑定 channel
agent.sources.seqGenSrc.channels = memoryChannel

# sink 到 hdfs
agent.sinks.hdfsSink.type = hdfs
# sink 到 hdfs 的地址
agent.sinks.hdfsSink.hdfs.path = hdfs://uhadoop-wslk1c-master1:8020/kafka/%{topi
c}/%y-%m-%d
agent.sinks.hdfsSink.hdfs.rollInterval = 0
agent.sinks.hdfsSink.hdfs.rollSize = 134217728
agent.sinks.hdfsSink.hdfs.rollCount = 0
agent.sinks.hdfsSink.hdfs.rollInterval = 0
agent.sinks.hdfsSink.hdfs.minBlockReplicas = 1
agent.sinks.hdfsSink.hdfs.writeFormat = Text
agent.sinks.hdfsSink.hdfs.fileType = DataStream
agent.sinks.hdfsSink.hdfs.batchSize = 1000
agent.sinks.hdfsSink.hdfs.threadsPoolSize= 100
# 指定从哪个 channel sink 数据
agent.sinks.hdfsSink.channel = memoryChannel

# channel 的配置，将 Source 接收到数据的一个缓冲到内存中。
# 详细说明请参考 https://flume.apache.org/FlumeUserGuide.html#memory-channel
agent.channels.memoryChannel.type = memory
agent.channels.memoryChannel.capacity = 10000
agent.channels.memoryChannel.transactionCapacity = 1500
```

启动命令

./bin/flume-ng agent -n agent -c conf -f conf/flume-conf.properties

执行结果

可以在 hdfs 上看到上传的文件

[hadoop@uhadoop-wslk1c-master1 root]$ hdfs dfs -ls -R /kafka

drwxrwxrwt - root supergroup 0 2016-03-12 18:48 /kafka/flume_kafka_sink

drwxrwxrwt - root supergroup 0 2016-03-12 18:48 /kafka/flume_kafka_sink/16-03-12

-rw-r--r-- 3 root supergroup 6 2016-03-12 18:48 /kafka/flume_kafka_sink/16-03-12/FlumeData.1457779695244.tmp

1.6使用 spark 消费 UKafka 消息

Java 示例:

```
package org.apache.spark.examples.streaming;

import java.util.Map;
import java.util.HashMap;
import java.util.regex.Pattern;



import scala.Tuple2;

import com.google.common.collect.Lists;
import org.apache.spark.SparkConf;
import org.apache.spark.api.java.function.FlatMapFunction;
```

```
import  org.apache.spark.api.java.function.Function;
import  org.apache.spark.api.java.function.Function2;
import  org.apache.spark.api.java.function.PairFunction;

import  org.apache.spark.streaming.Duration;
import  org.apache.spark.streaming.api.java.JavaDStream;
import  org.apache.spark.streaming.api.java.JavaPairDStream;
import  org.apache.spark.streaming.api.java.JavaPairReceiverInputDStream;
import  org.apache.spark.streaming.api.java.JavaStreamingContext;
import  org.apache.spark.streaming.kafka.KafkaUtils;


public final class JavaKafkaWordCount {
  private static final Pattern SPACE = Pattern.compile(" ");

  private JavaKafkaWordCount() {
  }

  public static void main(String[] args) {
    if (args.length < 4) {
      System.err.println("Usage: JavaKafkaWordCount <zkQuorum> <grou
p> <topics> <numThreads>");
      System.exit(1);
    }


    SparkConf sparkConf = new SparkConf().setAppName("JavaKafkaWord
Count");
    // Create the context with a 1 second batch size
    JavaStreamingContext jssc = new JavaStreamingContext(sparkConf, new
 Duration(2000));
```

```
        int numThreads = Integer.parseInt(args[3]);
        Map<String, Integer> topicMap = new HashMap<String, Integer>();
        String[] topics = args[2].split(",");
        for (String topic: topics) {
          topicMap.put(topic, numThreads);
        }

        JavaPairReceiverInputDStream<String, String> messages =
                KafkaUtils.createStream(jssc, args[0], args[1], topicMap);

        JavaDStream<String> lines = messages.map(new Function<Tuple2<Strin
g, String>, String>() {
          @Override
          public String call(Tuple2<String, String> tuple2) {
            return tuple2._2();
          }
        });

        JavaDStream<String> words = lines.flatMap(new FlatMapFunction<Strin
g, String>() {
          @Override
          public Iterable<String> call(String x) {
            return Lists.newArrayList(SPACE.split(x));
          }
        });

        JavaPairDStream<String, Integer> wordCounts = words.mapToPair(
          new PairFunction<String, String, Integer>() {
            @Override
            public Tuple2<String, Integer> call(String s) {
              return new Tuple2<String, Integer>(s, 1);
            }
```

```
            }).reduceByKey(new Function2<Integer, Integer, Integer>() {
                @Override
                public Integer call(Integer i1, Integer i2) {
                    return i1 + i2;
                }
            });

            wordCounts.print();

            // wordCounts.saveAsHadoopFiles("jwc", "sufix");
            jssc.start();
            jssc.awaitTermination();
        }
}
```

下载依赖包：

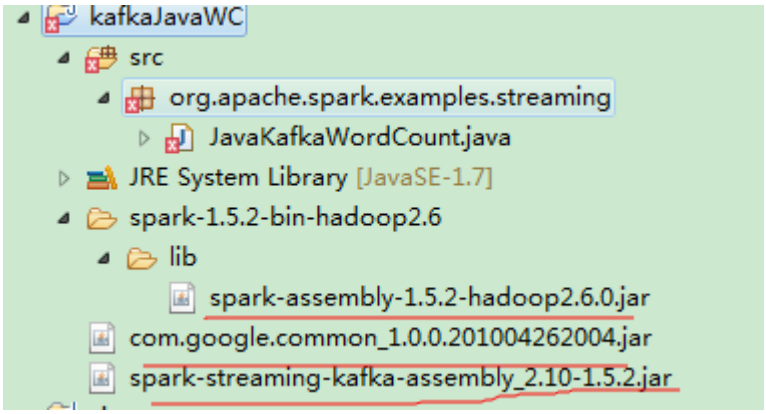
http://www.java2s.com/Code/JarDownload/com.google/com.google.common_1.0.0.201004262004.jar.zip

http://central.maven.org/maven2/org/apache/spark/spark-streaming-kafka-assembly_2.10/1.5.2/spark-streaming-kafka-assembly_2.10-1.5.2.jar

https://spark.apache.org/downloads.html

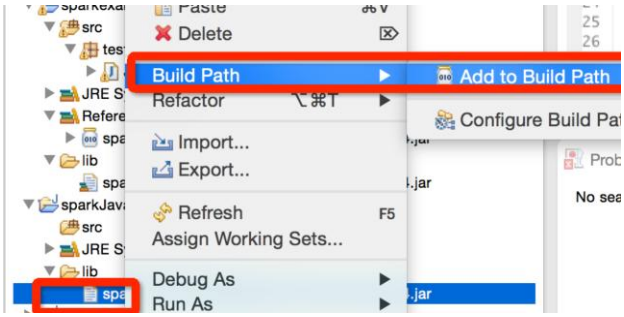
使用 **eclipse** 创建一个新的工程，并将添加上面的代码到工程。

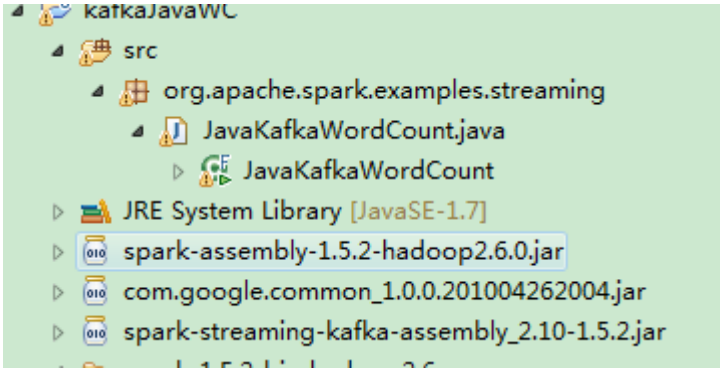
导入下面 **jar** 包



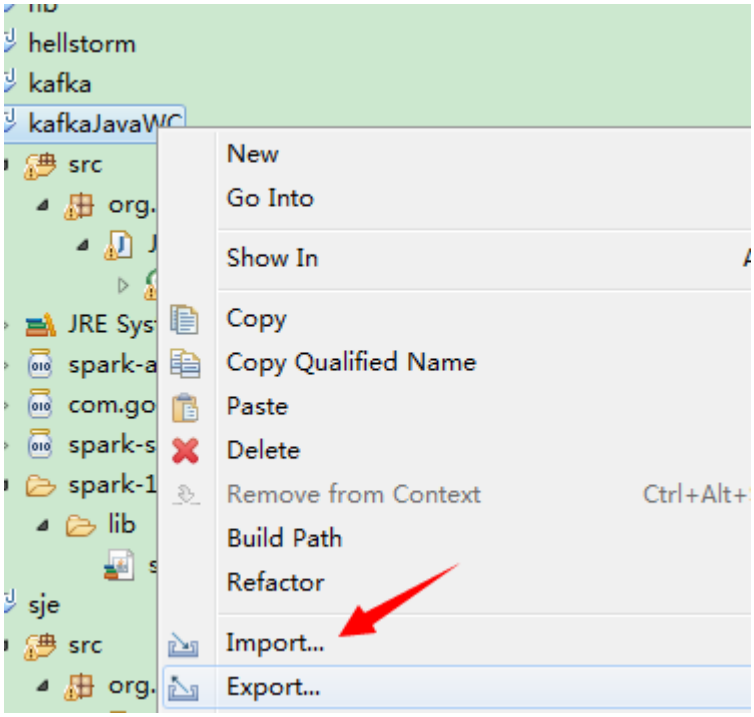
其中 spark-assembly-1.5.2-hadoop2.6.0.jar 在上面下载的 spark-1.5.2-bin-hadoop2.6.tgz 的 lib 目录下

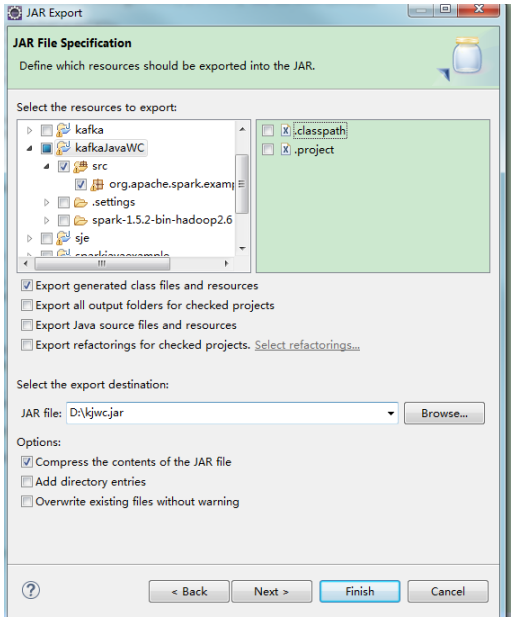
添加到 **jar** 包到 **build** 目录





导出 jar 包


hellstorm
kafka
kafkaJavaWC
src
org.
JRE Sys
spark-a
com.go
spark-s
spark-1
lib
sje
src
org.
New
Go Into
Show In
Copy
Copy Qualified Name
Paste
Delete
Remove from Context
Ctrl+Alt+
Build Path
Refactor
Import...
Export...


JAR Export
JAR File Specification
Define which resources should be exported into the JAR.
Select the resources to export:
kafka
kafkaJavaWC
src
org.apache.spark.exam
.settings
spark-1.5.2-bin-hadoop2.6
sje
.classpath
.project
Export generated class files and resources
Export all output folders for checked projects
Export Java source files and resources
Export refactorings for checked projects. Select refactorings...
Select the export destination:
JAR file: D:\kjwc.jar
Browse...
Options:
Compress the contents of the JAR file
Add directory entries
Overwrite existing files without warning
< Back
Next >
Finish
Cancel

启动命令

将上一步导出的 jar 包拷贝到 spark 集群，为了减少拷贝文件大小，所以在打包 kjwc.jar 的时候没有带 spark-streaming-kafka-assembly_2.10-1.5.2.jar 这个包，所以需要在执行时手动指定。

spark-submit --master yarn --jars spark-streaming-kafka-assembly_2.10-1.5.2.jar --class org.apache.spark.examples.streaming.JavaKafkaWordCount kjwc.jar ukafka-ytqpg4-kafka1:2181 test-consumer-group test_topic 1 2

启动程序后在使用上面介绍的发送消息方法向 test_topic 这个 topic 发送消息，可以在在输出终端可以看到类似这样的字段。

```
-------------------------------------------
Time: 1457593316000 ms
-------------------------------------------
(one,1)
(onee,1)
```

Scala 示例

```
package org.apache.spark.examples.streaming

import java.util.HashMap
import org.apache.kafka.clients.producer.{ProducerConfig, KafkaProducer, ProducerRecord}
import org.apache.spark.streaming._
import org.apache.spark.streaming.kafka._
import org.apache.spark.SparkConf

object KafkaWordCount {
  def main(args: Array[String]) {
    if (args.length < 4) {
      System.err.println("Usage: KafkaWordCount <zkQuorum> <group> <topic
```

```
s> <numThreads>")
        System.exit(1)
      }

      StreamingExamples.setStreamingLogLevels()

      val Array(zkQuorum, group, topics, numThreads) = args
      val sparkConf = new SparkConf().setAppName("KafkaWordCount")
      val ssc = new StreamingContext(sparkConf, Seconds(2))
      ssc.checkpoint("checkpoint")

      val topicMap = topics.split(",").map((_, numThreads.toInt)).toMap
      val lines = KafkaUtils.createStream(ssc, zkQuorum, group, topicMap).map
(_._2)
      val words = lines.flatMap(_.split(" "))
      val wordCounts = words.map(x => (x, 1L))
        .reduceByKeyAndWindow(_ + _, _ - _, Minutes(10), Seconds(2), 2)
      wordCounts.print()

      ssc.start()
      ssc.awaitTermination()
   }
}

// Produces some random words between 1 and 100.
object KafkaWordCountProducer {

   def main(args: Array[String]) {
      if (args.length < 4) {
        System.err.println("Usage: KafkaWordCountProducer <metadataBrokerList>
 <topic> " +
          "<messagesPerSec> <wordsPerMessage>")
```

```
        System.exit(1)
      }

      val Array(brokers, topic, messagesPerSec, wordsPerMessage) = args

      // Zookeeper connection properties
      val props = new HashMap[String, Object]()
      props.put(ProducerConfig.BOOTSTRAP_SERVERS_CONFIG, brokers)
      props.put(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG,
        "org.apache.kafka.common.serialization.StringSerializer")
      props.put(ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG,
        "org.apache.kafka.common.serialization.StringSerializer")

      val producer = new KafkaProducer[String, String](props)

      // Send some messages
      while(true) {
        (1 to messagesPerSec.toInt).foreach { messageNum =>
          val str = (1 to wordsPerMessage.toInt).map(x => scala.util.Random.nex
tInt(10).toString)
            .mkString(" ")

          val message = new ProducerRecord[String, String](topic, null, str)
          producer.send(message)
        }
        Thread.sleep(1000)
      }
    }
}
```

发送消息

spark-submit --master yarn --deploy-mode client --class org.apache.spark.examples.streaming.KafkaWordCountProducer /home/hadoop/spark/lib/spark-examples-1.5.2-hadoop2.6.0-cdh5.4.4.jar ukafka-ytqpg4-kafka1:9092 test_topic 1 2

其中与 kafka 相关的参数是 ukafka-ytqpg4-kafka1:9092 表示 producer 的地址和端口，test_topic 表示 topic，1 表示每秒发 1 条消息，2 表示每条消息中有 2 个单词

在 test_topic 的消息接收端，可以看到有持续的记录输出。

消费消息

spark-submit --master yarn --deploy-mode client --class org.apache.spark.examples.streaming.KafkaWordCount /home/hadoop/spark/lib/spark-examples-1.5.2-hadoop2.6.0-cdh5.4.4.jar ukafka-ytqpg4-kafka1:2181 test-consumer-group test_topic 1

ukafka-ytqpg4-kafka1:2181 表示 zookeeper 的监听地址，test-consumer-group 表示当前消费的程序的一个编号，test_topic 表示 topic，1 表示线程数。

会在输出的 teminal 中显示类似下面的内容

```
-------------------------------------------
Time: 1457593574000 ms
-------------------------------------------
(one,2)
(three,1)
(to,1)
(fo,1)
```

Python

```
#/usr/bin/python
import sys


from pyspark import SparkContext
```

```
    from pyspark.streaming import StreamingContext
    from pyspark.streaming.kafka import KafkaUtils

    if __name__ == "__main__":
        if len(sys.argv) != 4:
            print("%d",len(sys.argv))
            ## print("Usage: kafka_wordcount.py <zk> <topic>", file=sys.stder
r)
            exit(-1)

        sc = SparkContext(appName="PythonStreamingKafkaWordCount")
        ssc = StreamingContext(sc, 10)

        zkQuorum, topic, file = sys.argv[1:]
        kvs = KafkaUtils.createStream(ssc, zkQuorum, "spark-streaming-consum
er", {topic: 1})
        lines = kvs.map(lambda x: x[1])
        counts = lines.flatMap(lambda line: line.split(" ")) \
            .map(lambda word: (word, 1)) \
            .reduceByKey(lambda a, b: a+b)
        counts.pprint()
        lines.pprint()
        lines.saveAsTextFiles(file)
        ssc.start()
        ssc.awaitTermination()
```

执行代码方法：

需要配置外网 IP，才能自动下载依赖。

spark-submit --packages org.apache.spark:spark-streaming-kafka_2.10:1.5.2 --master yarn --deploy-mode client wordcount.py ukafka-ytqpg4-kafka1:2181 test_topic wc

手动下载依赖包：

wget http://central.maven.org/maven2/org/apache/spark/spark-streaming-kafka-assembly_2.10/1.5.2/spark-streaming-kafka-assembly_2.10-1.5.2.jar

spark-submit --jars spark-streaming-kafka-assembly_2.10-1.5.2.jar --master yarn --deploy-mode client wordcount.py ukafka-ytqpg4-kafka1:2181 test_topic wc

在另一个终端向指定的 topic 发送消息。

```
-------------------------------------------
Time: 2016-03-10 15:07:53
-------------------------------------------
(u'', 4)
(u'helo', 1)
(u'eon', 1)
(u'three', 2)
(u'one', 7)
(u'to', 4)
(u'\tthree', 1)
(u'threee', 1)
(u'two', 1)
(u'fo', 2)
```

1.7 Storm 消费 kafka 消息

安装 Storm 依赖库

下载并解压 Storm 发布版本

http://apache.fayea.com/storm/apache-storm-0.9.2-incubating/apache-storm-0.9.2-incubating.zip

这个版本带有 kafka 相关操作的完整的完整的依赖包。

修改 storm.yaml 配置文件

Storm 发行版本解压目录下有一个 conf/storm.yaml 文件，用于配置 Storm。默认配置在可以查看 https://github.com/nathanmarz/storm/blob/master/conf/defaults.yaml。conf/storm.yaml 中的配置选项将覆盖 defaults.yaml 中的默认配置。以下配置选项是必须在 conf/storm.yaml 中进行配置的：

1) storm.zookeeper.servers: Storm 集群使用的 Zookeeper 集群地址，其格式如下：

```
storm.zookeeper.servers:
  - "111.222.333.444"
  - "555.666.777.888"
```

如果 Zookeeper 集群使用的不是默认端口，那么还需要 storm.zookeeper.port 选项。

2) storm.local.dir: Nimbus 和 Supervisor 进程用于存储少量状态，如 jars、confs 等的本地磁盘目录，需要提前创建该目录并给以足够的访问权限。然后在 storm.yaml 中配置该目录，如：

```
storm.local.dir: "/data/storm
```

3) nimbus.host: Storm 集群 Nimbus 机器地址，各个 Supervisor 工作节点需要知道哪个机器是 Nimbus，以便下载 Topologies 的 jars、confs 等文件，如：

```
nimbus.host: "111.222.333.444"
```

4) supervisor.slots.ports: 对于每个 Supervisor 工作节点，需要配置该工作节点可

以运行的 worker 数量。每个 worker 占用一个单独的端口用于接收消息，该配置选项即用于定义哪些端口是可被 worker 使用的。默认情况下，每个节点上可运行 4 个 workers，分别在 6700、6701、6702 和 6703 端口，如：

```
supervisor.slots.ports:
    - 6700
    - 6701
    - 6702
    - 6703
```

配置文件不要顶格写。

启动 Storm 各个后台进程。

最后一步，启动 Storm 的所有后台进程。和 Zookeeper 一样，Storm 也是快速失败（fail-fast)的系统，这样 Storm 才能在任意时刻被停止，并且当进程重启后被正确地恢复执行。这也是为什么 Storm 不在进程内保存状态的原因，即使 Nimbus 或 Supervisors 被重启，运行中的 Topologies 不会受到影响。

以下是启动 Storm 各个后台进程的方式：

Nimbus: 在 Storm 主控节点上运行"bin/storm nimbus >/dev/null 2>&1 &"启动 Nimbus 后台程序，并放到后台执行；

Supervisor: 在 Storm 各个工作节点上运行"bin/storm supervisor >/dev/null 2>&1 &"启动 Supervisor 后台程序，并放到后台执行；

UI: 在 Storm 主控节点上运行"bin/storm ui >/dev/null 2>&1 &"启动 UI 后台程序，并放到后台执行，启动后可以通过 http://{nimbus host}:8080 观察集群的 worker 资源使用情况、Topologies 的运行状态等信息。

注意事项：

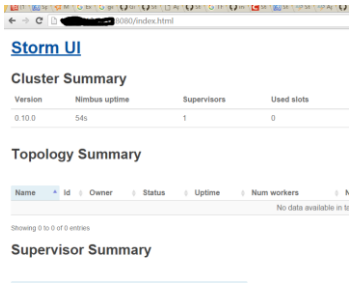
Storm 后台进程被启动后，将在 Storm 安装部署目录下的 logs/子目录下生成各个进程的日志文件。

经测试，Storm UI 必须和 Storm Nimbus 部署在同一台机器上，否则 UI 无法正常工作，因为 UI 进程会检查本机是否存在 Nimbus 链接。

为了方便使用，可以将 bin/storm 加入到系统环境变量中。

至此，Storm 集群已经部署、配置完毕，可以向集群提交拓扑运行了。

UI 展示界面



向集群提交任务

代码

topology.java

```
import  java.util.HashMap;
import  java.util.Map;

import  backtype.storm.Config;
import  backtype.storm.LocalCluster;
import  backtype.storm.StormSubmitter;
import  backtype.storm.spout.SchemeAsMultiScheme;
import  backtype.storm.topology.TopologyBuilder;
import  storm.kafka.BrokerHosts;
import  storm.kafka.KafkaSpout;
import  storm.kafka.SpoutConfig;
import  storm.kafka.ZkHosts;
import  storm.kafka.bolt.KafkaBolt;

public  class  topology  {

      public  static  void  main(String  []  args)  throws  Exception{
            //配置 zookeeper  主机:端口号
BrokerHosts  brokerHosts  =new  ZkHosts("110.64.76.130:2181,110.64.76.131:2181,
110.64.76.132:2181");
            //接收消息队列的主题
```

```
        String topic="test";
        //zookeeper 设置文件中的配置，如果 zookeeper 配置文件中设置为主机名：端口号 ，该项为空
        String zkRoot="";
        //任意
        String spoutId="test_consumer_group";
        SpoutConfig spoutConfig=new SpoutConfig(brokerHosts, topic, zkRoot, spoutId);
        //设置如何处理 kafka 消息队列输入流
        spoutConfig.scheme=new SchemeAsMultiScheme(new MessageScheme());
        // 从 offset 最小值开始读取数据，否则从启动后开始读
        spoutConfig.forceFromStart=true;
        Config conf=new Config();
        //不输出调试信息
        conf.setDebug(false);
        //设置一个 spout task 中处于 pending 状态的最大的 tuples 数量
        conf.put(Config.TOPOLOGY_MAX_SPOUT_PENDING, 1);
        Map<String, String> map=new HashMap<String,String>();
        // 配置 Kafka broker 地址
map.put("metadata.broker.list", "master:9092,slave1:9092,slave2:9092");
        // serializer.class 为消息的序列化类
        map.put("serializer.class", "kafka.serializer.StringEncoder");
        conf.put("kafka.broker.properties", map);
        // 配置 KafkaBolt 生成的 topic
        conf.put("topic", "receiver");
        TopologyBuilder builder =new TopologyBuilder();
        builder.setSpout("spout", new KafkaSpout(spoutConfig),1);
        builder.setBolt("bolt1", new QueryBolt(),1).setNumTasks(1).shuffleGrouping("spout");
        builder.setBolt("bolt2", new KafkaBolt<String, String>(),1).setNumTasks(1).shuffleGrouping("bolt1");
```

```
            String name= topology.class.getSimpleName();
            if (args != null && args.length > 0) {
                // Nimbus host name passed from command line
                conf.put(Config.NIMBUS_HOST, args[0]);
                conf.setNumWorkers(3);
                StormSubmitter.submitTopologyWithProgressBar(name, conf, builder.
createTopology());
        } else {
                conf.setMaxTaskParallelism(3);
                LocalCluster cluster = new LocalCluster();
                cluster.submitTopology(name, conf, builder.createTopology());
                Thread.sleep(60000);
                cluster.shutdown();
        }
    }
}
```

MessageScheme.java

```
import java.io.UnsupportedEncodingException;
import java.util.List;

import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

import backtype.storm.spout.Scheme;
import backtype.storm.tuple.Fields;
import backtype.storm.tuple.Values;

public class MessageScheme implements Scheme {
    private static final Logger LOGGER = LoggerFactory.getLogger(MessageSc
heme.class);
    public List<Object> deserialize(byte[] ser) {
        try {
```

```
            //从 kafka 中读取的值直接序列化为 UTF-8 的 str
            String mString=new String(ser, "UTF-8");
            return new Values(mString);
        } catch (UnsupportedEncodingException e) {
            // TODO Auto-generated catch block
            LOGGER.error("Cannot parse the provided message");
        }
        return null;
    }

    public Fields getOutputFields() {
        // TODO Auto-generated method stub
        return new Fields("msg");
    }
}
```

QueryBolt.java

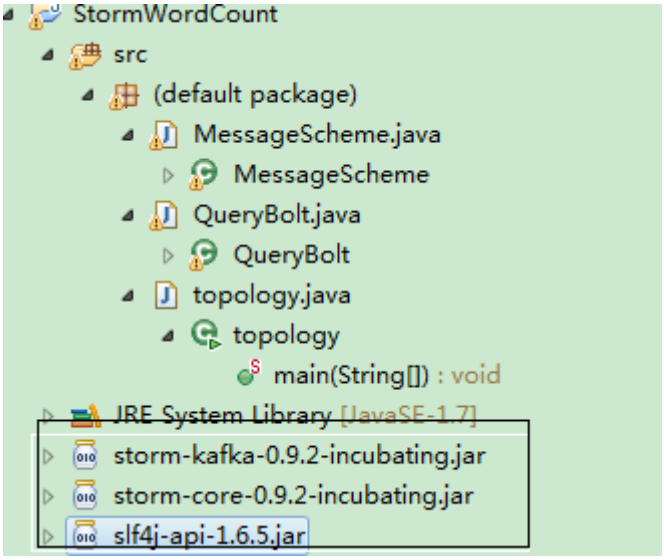
```
import java.io.FileNotFoundException;
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.PrintStream;
import java.util.ArrayList;
import java.util.List;
import java.util.Map;

import backtype.storm.task.OutputCollector;
import backtype.storm.task.TopologyContext;
import backtype.storm.topology.IRichBolt;
import backtype.storm.topology.OutputFieldsDeclarer;
import backtype.storm.tuple.Fields;
import backtype.storm.tuple.Tuple;
import backtype.storm.tuple.Values;
```

```
public class QueryBolt implements IRichBolt {

    List<String> list;
    OutputCollector collector;
    public void prepare(Map stormConf, TopologyContext context, OutputCollector collector) {
        list=new ArrayList<String>();
        this.collector=collector;
    }

    public void execute(Tuple input) {
        // TODO Auto-generated method stub
        String str=(String) input.getValue(0);
        //将 str 加入到 list
        list.add(str);
        //发送 ack
        collector.ack(input);
        //发送该 str
        collector.emit(new Values(str));
    }

    public void cleanup() {//topology 被 killed 时调用
        //将 list 的值写入到文件
        try {
            FileOutputStream outputStream=new FileOutputStream("/data/"+this+".txt");
            PrintStream p=new PrintStream(outputStream);
            p.println("begin!");
            p.println(list.size());
            for(String tmp:list){
                p.println(tmp);
            }
```

```
                p.println("end!");
                try {
                    p.close();
                    outputStream.close();
                } catch (IOException e) {
                    // TODO Auto-generated catch block
                    e.printStackTrace();
                }

            } catch (FileNotFoundException e) {
                // TODO Auto-generated catch block
                e.printStackTrace();
            }
        }

        public void declareOutputFields(OutputFieldsDeclarer declarer) {
            declarer.declare(new Fields("message"));
        }

        public Map<String, Object> getComponentConfiguration() {
            // TODO Auto-generated method stub
            return null;
        }
}
```

编译

从上面下载的 storm 包中找到下面 3 个包，导入到工程中。



拷贝集群运行依赖的 jar 包到默认路径

cd

//拷贝 kafka 的 jar 包到运行目录

cp /usr/local/kafka/libs/* apache-storm-0.9.2-incubating/lib

将 storm-kafka 的包拷贝的默认 lib 目录下

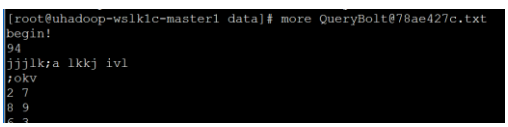
cp /storm-kafka/storm-kafka-0.9.2-incubating.jar apache-storm-0.9.2-incubating/lib/

启动 Storm Topology

bin/storm jar /data/sjwc.jar topology

其中，sjwc.jar 是包含 Topology 实现代码的 jar 包，topology 的 main 方法是 Topology 的入口。

运行结果可以在/data/下看到下面内容



94 是单词个数，下面是 topic 中所有内容。

停止 Storm Topology

storm kill {toponame}

其中，{toponame}为 Topology 提交到 Storm 集群时指定的 Topology 任务名称

2 常见问题

修改 spark 任务输出日志级别

修改两个 master 节点文件/home/hadoop/conf/log4j.properties，

修改前

hadoop.root.logger=INFO,console

修改后

hadoop.root.logger=WARN,console

然后重启 resource 服务

service hadoop-yarn-resourcemanager restart

UKafka 软件的部署情况？

Zookeeper 服务默认部署在 3 个默认节点上面。

节点名称	节点ID	内网IP	ZooKeeper	节点配置
ukafka-sxpy3c-kafka1	node-kas4s4	[redacted]	有	1 2 100
ukafka-sxpy3c-kafka2	node-gtqwd4	[redacted]	有	1 2 100
ukafka-sxpy3c-kafka3	node-ktadpu	[redacted]	有	1 2 100
ukafka-sxpy3c-kafka4	node-w2g1gr	[redacted]	无	8 24 16384

其他节点部署的服务全部相同。

Kafka 部署目录：/usr/local/kafka

Zookeeper 部署目录：/usr/local/zookeeper

监控数据没有了？

1. AgentMonitor.js 和 kafkaAgent.js 进程是否被 kill 掉了。
2. 65431 端口为 Agent 所需要端口是否做了防火墙限制。