消息队列 CKafka

最佳实践

产品文档

腾讯云

文档目录

最佳实践
- Spark Streaming接入Ckafka最佳实践
- Flume 接入Ckafka最佳实践
- Kafka Connect 接入 CKafka 实践
- Storm 接入 Ckafka 最佳实践
- Logstash接入CKafka最佳实践

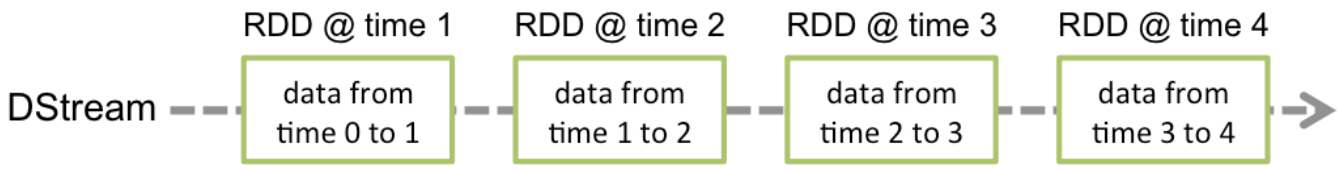
最佳实践
Spark Streaming接入Ckafka最佳实践

最近更新时间：2018-02-08 11:18:53

Spark Streaming简介

Spark Streaming是Spark Core的一个扩展，用于高吞吐且容错地处理持续性的数据，目前支持的外部输入有Kafka，Flume，HDFS/S3，Kinesis，Twitter和TCP socket。

Spark Streaming将连续数据抽象成DStream(Discretized Stream)，而DStream由一系列连续的RDD(弹性分布式数据集)组成，每个RDD是一定时间间隔内产生的数据。使用函数对DStream进行处理其实即为对这些RDD进行处理。



目前Spark Streaming对kafka作为数据输入的支持分为稳定版本与实验版本：

Kafka Version	spark-streaming-kafka-0.8	spark-streaming-kafka-0.10
Broker Version	0.8.2.1 or higher	0.10.0 or higher
Api Stability	Stable	Experimental
Language Support	Scala, Java, Python	Scala, Java
Receiver DStream	Yes	No
Direct DStream	Yes	Yes
SSL / TLS Support	No	Yes
Offset Commit Api	No	Yes
Dynamic Topic Subscription	No	Yes

目前ckafka支持0.9.0.x，0.10.0.x，0.10.1.x，0.10.2.x版本，本次实践使用0.10.2.1版本的kafka依赖

Spark Streaming接入CKafka

申请Ckafka实例

确认网络类型是否与当前使用网络相符

创建topic

这里创建了一个名为spark_test的topic，接下来将以该topic为例子介绍如何生产消费
[内网IP与端口]即为生产消费需要用到的bootstrap-server

云主机环境

Centos6.8系统

package	version
sbt	0.13.16
hadoop	2.7.3
spark	2.1.0
protobuf	2.5.0

package	version
ssh	CentOS默认安装
Java	1.8

向Ckafka中生产

目前ckafka支持0.9.0.x，0.10.0.x，0.10.1.x，0.10.2.x版本
这里使用0.10.2.1版本的kafka依赖
build.sbt

```
name := "Producer Example"
version := "1.0"
scalaVersion := "2.11.8"
libraryDependencies += "org.apache.kafka" % "kafka-clients" % "0.10.2.1"
```

producer_example.scala

```
import java.util.Properties
import org.apache.kafka.clients.producer._

object ProducerExample extends App {
   val  props = new Properties()
   props.put("bootstrap.servers", "172.16.16.12:9092") //实例信息中的内网ip与端口

   props.put("key.serializer", "org.apache.kafka.common.serialization.StringSerializer")
   props.put("value.serializer", "org.apache.kafka.common.serialization.StringSerializer")

   val producer = new KafkaProducer[String, String](props)
   val TOPIC="test"  //指定要生产的topic
   for(i<- 1 to 50){
         val record = new ProducerRecord(TOPIC, "key", s"hello $i") //生产key是"key",value是 hello i的消息
         producer.send(record)
   }
   val record = new ProducerRecord(TOPIC, "key", "the end "+new java.util.Date)
   producer.send(record)
   producer.close() //最后要断开
}
```

有关更多ProducerRecord的用法可以查阅
https://kafka.apache.org/0100/javadoc/org/apache/kafka/clients/producer/ProducerRecord.html

从Ckafka消费

DirectStream

在 build.sbt 添加依赖

```
name := "Consumer Example"
version := "1.0"
scalaVersion := "2.11.8"
libraryDependencies += "org.apache.spark" %% "spark-core" % "2.1.0"
libraryDependencies += "org.apache.spark" %% "spark-streaming" % "2.1.0"
libraryDependencies += "org.apache.spark" %% "spark-streaming-kafka-0-10" % "2.1.0"
```

DirectStream_example.scala

```
import org.apache.kafka.clients.consumer.ConsumerRecord
import org.apache.kafka.common.serialization.StringDeserializer
import org.apache.kafka.common.TopicPartition
import org.apache.spark.streaming.kafka010._
import org.apache.spark.streaming.kafka010.LocationStrategies.PreferConsistent
import org.apache.spark.streaming.kafka010.ConsumerStrategies.Subscribe
import org.apache.spark.streaming.kafka010.KafkaUtils
import org.apache.spark.streaming.kafka010.OffsetRange
import org.apache.spark.streaming.{Seconds, StreamingContext}
import org.apache.spark.SparkConf
import org.apache.spark.SparkContext
import collection.JavaConversions._
import Array._

object Kafka {
  def main(args: Array[String]) {
    val kafkaParams = Map[String, Object](
      "bootstrap.servers" -> "172.16.16.12:9092",
      "key.deserializer" -> classOf[StringDeserializer],
      "value.deserializer" -> classOf[StringDeserializer],
      "group.id" -> "spark_stream_test1",
      "auto.offset.reset" -> "earliest",
      "enable.auto.commit" -> "false"
    )

    val sparkConf = new SparkConf()
    sparkConf.setMaster("local")
    sparkConf.setAppName("Kafka")
    val ssc = new StreamingContext(sparkConf, Seconds(5))
    val topics = Array("spark_test")

    val offsets : Map[TopicPartition, Long] = Map()
```

```
        for (i <- 0 until 3){
            val tp = new TopicPartition("spark_test", i)
            offsets.updated(tp , 0L)
        }
        val stream = KafkaUtils.createDirectStream[String, String](
            ssc,
            PreferConsistent,
            Subscribe[String, String](topics, kafkaParams)
        )
        println("directStream")
        stream.foreachRDD{ rdd=>
            //输出获得的消息
            rdd.foreach{iter =>
                val i = iter.value
                println(s"${i}")
            }
            //获得offset
            val offsetRanges = rdd.asInstanceOf[HasOffsetRanges].offsetRanges
            rdd.foreachPartition { iter =>
                val o: OffsetRange = offsetRanges(TaskContext.get.partitionId)
                println(s"${o.topic} ${o.partition} ${o.fromOffset} ${o.untilOffset}")
            }
        }

        // Start the computation
        ssc.start()
        ssc.awaitTermination()
    }
}
```

RDD

build.sbt 配置同上

RDD_example

```
import org.apache.kafka.clients.consumer.ConsumerRecord
import org.apache.kafka.common.serialization.StringDeserializer
import org.apache.spark.streaming.kafka010._
import org.apache.spark.streaming.kafka010.LocationStrategies.PreferConsistent
import org.apache.spark.streaming.kafka010.ConsumerStrategies.Subscribe
import org.apache.spark.streaming.kafka010.KafkaUtils
import org.apache.spark.streaming.kafka010.OffsetRange
import org.apache.spark.streaming.{Seconds, StreamingContext}
import org.apache.spark.SparkConf
```

```
import org.apache.spark.SparkContext
import collection.JavaConversions._
import Array._


object Kafka {
  def main(args: Array[String]) {
    val kafkaParams = Map[String, Object](
      "bootstrap.servers" -> "172.16.16.12:9092",
      "key.deserializer" -> classOf[StringDeserializer],
      "value.deserializer" -> classOf[StringDeserializer],
      "group.id" -> "spark_stream",
      "auto.offset.reset" -> "earliest",
      "enable.auto.commit" -> (false: java.lang.Boolean)
    )
    val sc = new SparkContext("local", "Kafka", new SparkConf())
    val java_kafkaParams : java.util.Map[String, Object] = kafkaParams
    //按顺序向parition拉取相应offset范围的消息，如果拉取不到则阻塞直到超过等待时间或者新生产消息达到
    val offsetRanges = Array[OffsetRange](
      OffsetRange("spark_test", 0, 0, 5),
      OffsetRange("spark_test", 1, 0, 5),
      OffsetRange("spark_test", 2, 0, 5)
    )
    val range = KafkaUtils.createRDD[String, String](
      sc,
      java_kafkaParams,
      offsetRanges,
      PreferConsistent
    )
    range.foreach(rdd=>println(rdd.value))
    sc.stop()
  }
}
```

更多 kafkaParams 用法参考http://kafka.apache.org/documentation.html#newconsumerconfigs

配置环境

安装sbt

1. 在sbt官网上下载sbt包

2. 解压后在sbt的目录下创建一个sbt_run.sh脚本并增加可执行权限
   脚本内容如下：

```
#!/bin/bash
SBT_OPTS="-Xms512M -Xmx1536M -Xss1M -XX:+CMSClassUnloadingEnabled -XX:MaxPermSize=256
java $SBT_OPTS -jar `dirname $0`/bin/sbt-launch.jar "$@"
```

```
chmod u+x ./sbt_run.sh
```

3. 执行

```
./sbt-run.sh sbt-version
```

若能看到sbt版本说明可以正常运行

安装protobuf

4. 下载protobuf相应版本
5. 解压后进入目录

```
./configure
make && make install
```

需要预先安装gcc-g++，执行中可能需要root权限

6. 重新登录，在命令行中输入

```
protoc --version
```

7. 若能看到protobuf版本说明可以正常运行

安装hadoop

1. 访问hadoop官网下载所需要的版本
2. 增加hadoop用户

```
useradd -m hadoop -s /bin/bash
```

3. 增加管理员权限

```
visudo
```

4. 在 root ALL=(ALL) ALL 下增加一行
   `hadoop ALL=(ALL) ALL`
   保存退出
5. 使用hadoop进行操作

```
su hadoop
```

6. ssh无密码登录

```
cd ~/.ssh/                    # 若没有该目录，请先执行一次ssh localhost
ssh-keygen -t rsa             # 会有提示，都按回车就可以
cat id_rsa.pub >> authorized_keys  # 加入授权
chmod 600 ./authorized_keys    # 修改文件权限
```

7. 安装java

```
sudo yum install java-1.8.0-openjdk java-1.8.0-openjdk-devel
```

8. 配置${JAVA_HOME}

```
vim /etc/profile
```

在文末加上

```
export JAVA_HOME=/usr/lib/jvm/java-1.8.0-openjdk-1.8.0.121-0.b13.el6_8.x86_64/jre
export PATH=$PATH:$JAVA_HOME
```

根据安装情况修改对应路径

9. 解压hadoop，进入目录

```
./bin/hadoop version
```

若能显示版本信息说明能正常运行

0. 配置单机伪分布式(可根据需要搭建不同形式的集群)

```
vim /etc/profile
```

在文末加上

```
export HADOOP_HOME=/usr/local/hadoop
export PATH=$HADOOP_HOME/bin:$PATH
```

根据安装情况修改对应路径

1. 修改 /etc/hadoop/core-site.xml

```
<configuration>
<property>
    <name>hadoop.tmp.dir</name>
    <value>file:/usr/local/hadoop/tmp</value>
    <description>Abase for other temporary directories.</description>
</property>
```

```
<property>
    <name>fs.defaultFS</name>
    <value>hdfs://localhost:9000</value>
</property>
</configuration>
```

2. 修改 /etc/hadoop/hdfs-site.xml

```
<configuration>
<property>
    <name>dfs.replication</name>
    <value>1</value>
</property>
<property>
    <name>dfs.namenode.name.dir</name>
    <value>file:/usr/local/hadoop/tmp/dfs/name</value>
</property>
<property>
    <name>dfs.datanode.data.dir</name>
    <value>file:/usr/local/hadoop/tmp/dfs/data</value>
</property>
</configuration>
```

3. 修改 /etc/hadoop/hadoop-env.sh 中的JAVA_HOME为java的路径

```
export JAVA_HOME=/usr/lib/jvm/java-1.8.0-openjdk-1.8.0.121-0.b13.el6_8.x86_64/jre
```

4. 执行NameNode格式化

```
./bin/hdfs namenode -format
```

看到 Exitting with status 0 说明成功

5. 启动hadoop

```
./sbin/start-dfs.sh
```

成功启动会存在 NameNode 进程， DataNode 进程， SecondaryNameNode 进程

安装spark

访问spark官网下载所需要的版本

这里因为之前安装了hadoop选择使用*Pre-build with user-provided Apache Hadoop*

这里同样使用 hadoop 用户进行操作

6. 解压进入目录
7. 修改配置文件

```
cp ./conf/spark-env.sh.template ./conf/spark-env.sh
vim ./conf/spark-env.sh
```

在第一行添加

```
export SPARK_DIST_CLASSPATH=$(/usr/local/hadoop/bin/hadoop classpath)
```

根据hadoop安装情况修改路径

8. 运行例子

```
bin/run-example SparkPi
```

若成功安装可以看到程序输出π的近似值

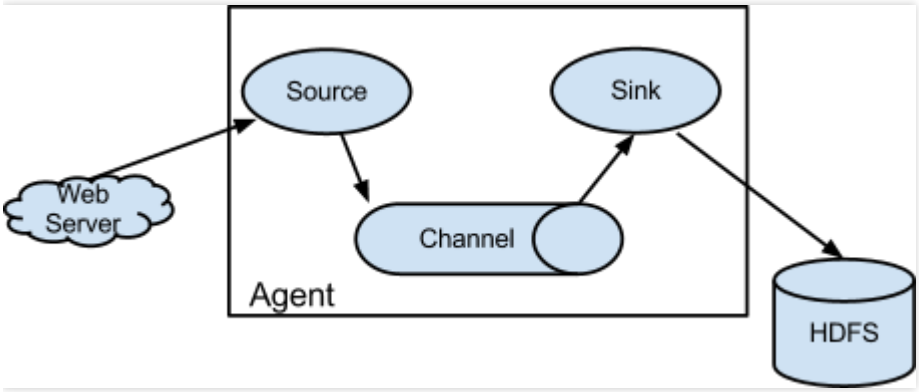
Flume 接入Ckafka最佳实践

最近更新时间：2018-02-08 11:19:16

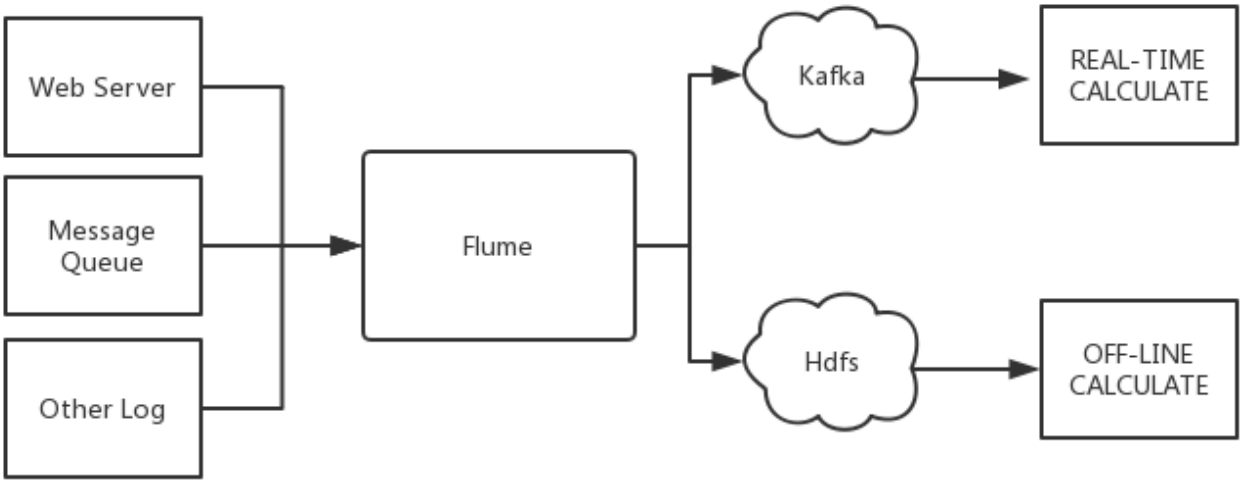
Apache Flume简介

Apache Flume 是一个从可以收集例如日志，事件等数据资源，并将这些数量庞大的数据从各项数据资源中集中起来存储的工具/服务，或者数集中机制。flume具有高可用，分布式，配置工具，其设计的原理也是基于将数据流。

Flume基本结构如下图所示：



Flume以agent为最小的独立运行单位。一个agent就是一个JVM。单agent由Source、Sink和Channel三大组件构成。



Why Flume + Kafka

把数据存储到hdfs或者hbase等下游存储模块或者计算模块时需要考虑各种复杂的场景，比如并发写入的量以及系统承载压力，网络延迟等等问题。flume设计为灵活的分布式系统具有多种接口，同时提供可定制化的管道。
在成产处理环节中，往往出现生产处理速度不一致的情况，此时kafka可以充当缓存角色。拥用partition结构以及采用append追加数据，使kafka具有优秀的吞吐能力。同时其拥有replication具有很高的容错性。
所以将两者结合起来，可以满足生产环境中绝大多数要求。

开源Kafka接入方式

版本支持

1. Flume当前版本 – Apache Flume 1.7.0 Released （ October 17, 2016发布，1.6之后才兼容kafka ）
2. 支持Kafka 0.9.x series， 0.8已经不支持

准备工作

1. Apache Flume （ 版本1.6.0以上 ）
2. Kafka （ 版本0.9.x以上 ）
3. Flume的Kafka –Source、 Sink组件（ 确认已经在Flume中 ）

Flume与Kafka

Kafka可作为Source或者Sink端对消息进行导入或者导出。

1. Kafka Source

配置kafka作为消息来源，即将自己作为消费者，从Kafka中拉取数据传入到指定Sink中
主要配置选项：

配置项	说明
channels	自己配置的channel
type	必须为：org.apache.flume.source.kafka.KafkaSource
kafka.bootstrap.servers	Kafka broker的服务器
kafka.consumer.group.id	作为Kafka消费端的group id
kafka.topics	Kafka中数据来源topic.
batchSize	每次写入channel的大小

配置项	说明
batchDurationMillis	每次写入最大间隔时间

example：

```
tier1.sources.source1.type = org.apache.flume.source.kafka.KafkaSource
tier1.sources.source1.channels = channel1
tier1.sources.source1.batchSize = 5000
tier1.sources.source1.batchDurationMillis = 2000
tier1.sources.source1.kafka.bootstrap.servers = localhost:9092
tier1.sources.source1.kafka.topics = test1, test2
tier1.sources.source1.kafka.consumer.group.id = custom.g.id
```

1. Kafka Sink

   配置kafka作为内容接收方，即将自己作为生产者，推到Kafka Server中等待后续操作

主要配置选项：

配置项	说明
channel	自己配置的channel
type	必须为：org.apache.flume.sink.kafka.KafkaSink
kafka.bootstrap.servers	Kafka broker的服务器
kafka.topics	Kafka中数据来源topic.
flumeBatchSize	每次写入的Bacth大小
kafka.producer.acks	Kafka生产者的生产策略

```
a1.sinks.k1.channel = c1
a1.sinks.k1.type = org.apache.flume.sink.kafka.KafkaSink
a1.sinks.k1.kafka.topic = mytopic
a1.sinks.k1.kafka.bootstrap.servers = localhost:9092
a1.sinks.k1.kafka.flumeBatchSize = 20
a1.sinks.k1.kafka.producer.acks = 1
```

更详细的内容可以参考官网链接：

https://flume.apache.org/FlumeUserGuide.html

Flume接入CKafka

准备工作

- 在Ckafka申请页中申请实例，并且创建对应的Topic
- apache flume ，本教程使用的是最新的flume 1.7.0 （ http://flume.apache.org/download.html)

Ckafka创建

1 ）拥有实例后，可从控制台中可以看到自己的实例信息

2 ）点击实例名称可以看到实例分配的具体信息：

3）点击topic管理，创建topic，此处名字为flume_test

至此，Ckafka相关的工作环境完成。

Flume

1）从官方下载Apache flume压缩包，进行解压

2）配置Flume选项

- 使用 Ckafka 作为Sink

  a) 编写配置文件，此处重点放在flume 与ckafka作为Sink结合上，所以Source和Channel使用默认配置，不做详细介绍。以下是一个简单的demo（配置在解压目录的conf文件夹下），需要注意的是，若无特殊要求则将自己的实例ip与topic替换到配置文件当中即可：

```
# 以kafka 作为sink 的demo
agentckafka.sources = exectail
agentckafka.channels = memoryChannel
agentckafka.sinks = kafkaSink

# 设置source类型,根据不同需求而设置
agentckafka.sources.exectail.type = exec
agentckafka.sources.exectail.command = tail -F ./flume-test
agentckafka.sources.exectail.batchSize=20
# 设置source channel
agentckafka.sources.exectail.channels = memoryChannel

# 设置sink类型，此处设置为kafka
agentckafka.sinks.kafkaSink.type= org.apache.flume.sink.kafka.KafkaSink
# 此处设置ckafka提供的ip:port
agentckafka.sinks.kafkaSink.brokerList= 172.16.16.12:9092
# 此处设置需要导入数据的topic，请先再控制台提前创建好topic
agentckafka.sinks.kafkaSink.topic= flume_test
# 设置sink channel
agentckafka.sinks.kafkaSink.channel = memoryChannel

# Each channel's type is defined.
agentckafka.channels.memoryChannel.type = memory
agentckafka.channels.memoryChannel.keep-alive = 10

# 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
agentckafka.channels.memoryChannel.capacity = 1000
agentckafka.channels.memoryChannel.transactionCapacity =1000
```


  b) 此处使用的source为tail -F flume-test ，即文件中新增的信息

  c) 启动flume ：

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

  d) 写入消息到flume-test文件中，此时消息将由flume写入到ckafka

```
[root@VM_16_17_centos apache-flume-1.7.0-bin]# cat flume-test
ckafka
```

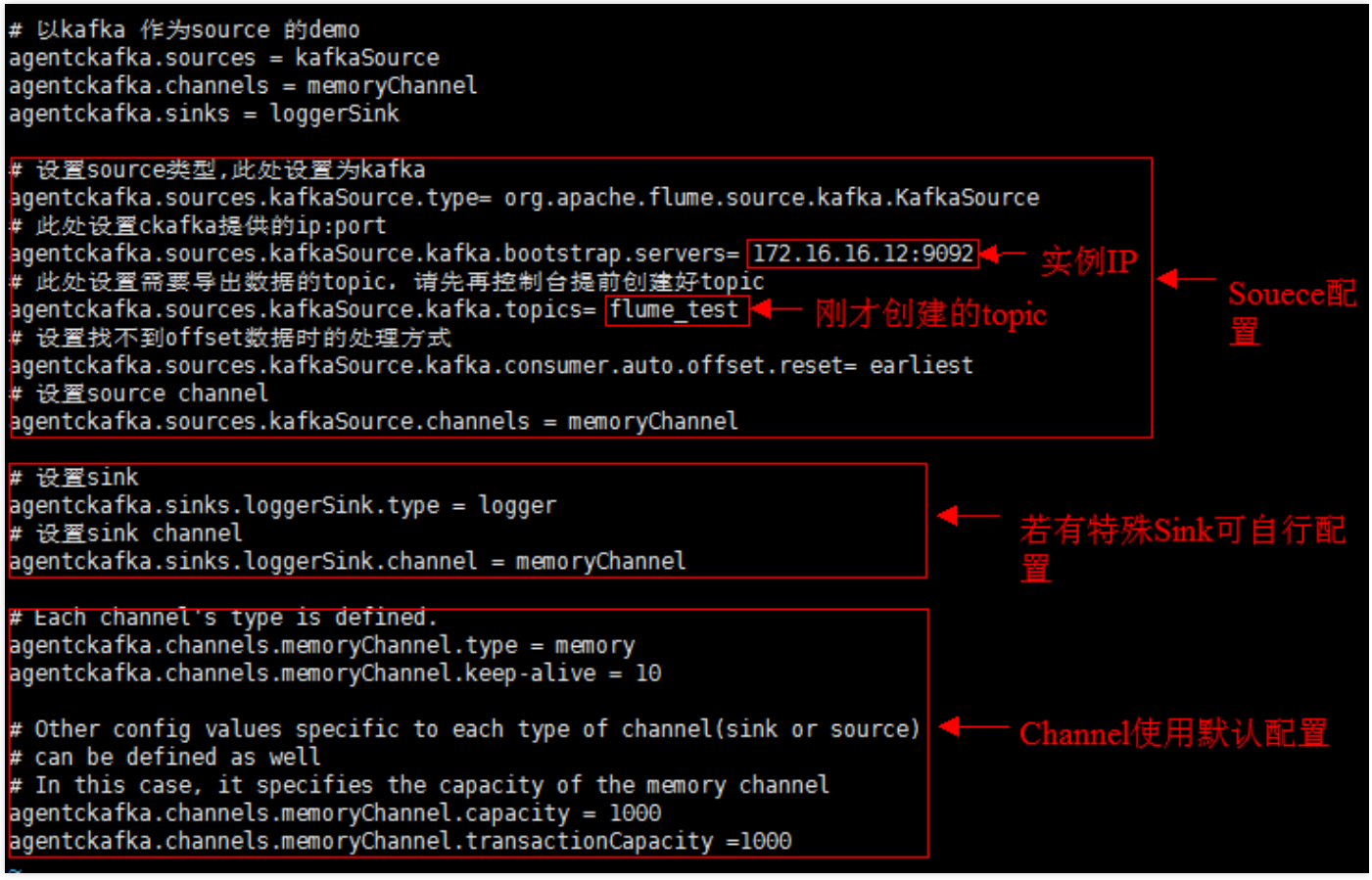
  e) 启动ckafka客户端进行消费：

```
./kafka-console-consumer.sh --bootstrap-server 172.16.16.12:9092 --topic flume_test --from-beginnin
```

可以看到刚刚的消息被消费出来了

- 使用 Ckafka 作为Source

  a) 编写配置文件，此处重点放在flume 与ckafka作为Source结合上，所以Sink和Channel使用默认配置，不做详细介绍。以下是一个简单的demo（配置在解压目录的conf文件夹下）。需要注意的是，若无特殊要求则将自己的实例ip与topic替换到配置文件当中即可：

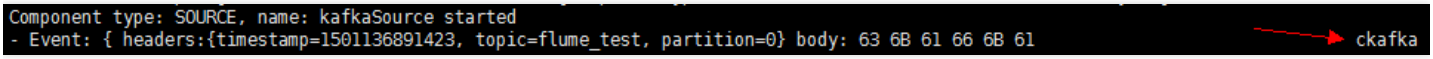


b) 此处使用的sink为logger

c) 启动flume ：

```
./bin/flume-ng agent -n agentckafka -c conf -f conf/flume-kafka-source.properties
```

d) 查看logger输出信息（默认路径 logs/flume.log）



Kafka Connect 接入 CKafka 实践

最近更新时间：2018-06-22 19:12:45

Kafka Connect 目前支持两种执行模式：standalone 和 distributed。

以 standalone 模式启动 connect

通过以下命令以 standalone 模式启动 connect：

```
bin/connect-standalone.sh config/connect-standalone.properties connector1.properties [connector2.pr
```

接入ckafka 与接入开源 kafka 没有区别，仅需要修改 bootstrap.servers 为申请实例时分配的 IP。

以 distributed 模式启动 connect

通过以下命令以 distributed 模式启动 connect：

```
bin/connect-distributed.sh config/connect-distributed.properties
```

该模式下，kafka connect 会将 offsets、configs 和 task status 信息存储在 kafka topic 中，存储的 topic 在 connect-distributed 中的以下字段配置：

```
config.storage.topic
offset.storage.topic
status.storage.topic
```

这三个 topic 需要手动创建，才能保证创建的属性符合 connect 的要求。

- config.storage.topic 需要保证只有一个 partition，多副本且为 compact 模式。
- offset.storage.topic 需要有多个 partition，多副本且为 compact 模式。
- status.storage.topic 需要有多个 partition，多副本且为 compact 模式。

配置 bootstrap.servers 为申请实例是分配的 IP；

配置 group.id，用于标识 connect 集群，需要与消费者分组区分开来。

Storm 接入 Ckafka 最佳实践

最近更新时间：2018-02-08 11:20:46

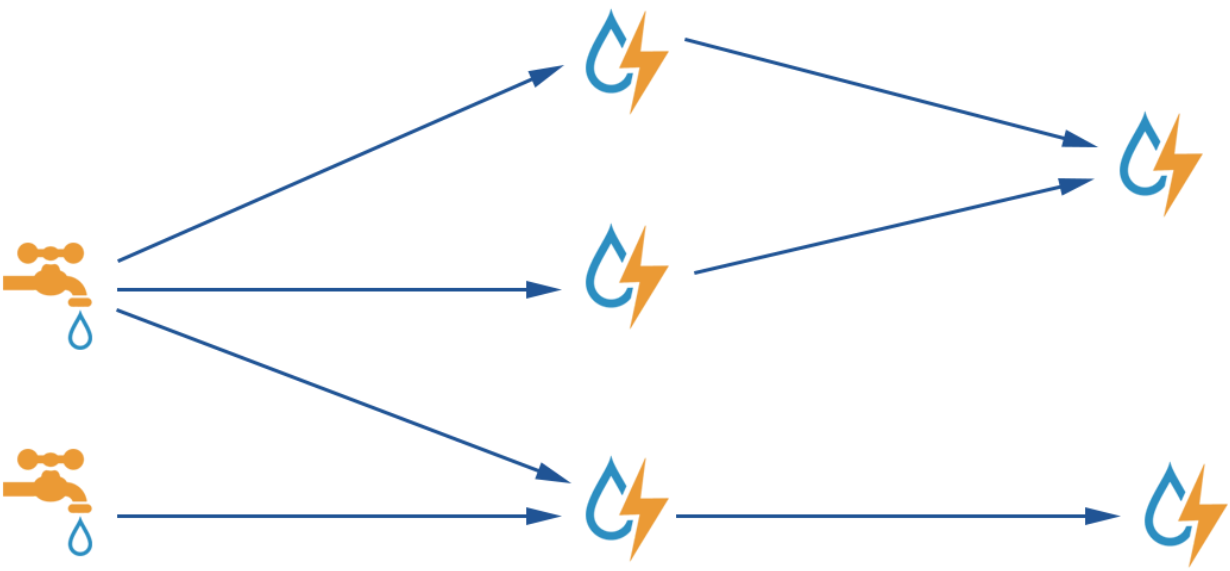
Storm 简介

Storm 是一个分布式实时计算框架，能够对数据进行流式处理和提供通用性分布式 RPC 调用，可以实现处理事件亚秒级的延迟，适用于对延迟要求比较高的实时数据处理场景。

Storm 工作原理

在 Storm 的集群中有两种节点，控制节点 Master Node 和工作节点 Worker Node 。 Master Node 上运行 Nimbus 进程，用于资源分配与状态监控。 Worker Node 上运行 Supervisor 进程，监听工作任务，启动 executor 执行。整个 Storm 集群依赖 zookeeper 负责公共数据存放、集群状态监听、任务分配等功能。

用户提交给 Storm 的数据处理程序称为 topology ，它处理的最小消息单位是 tuple ，一个任意对象的数组。 topology 由 spout 和 bolt 构成， spout 是产生 tuple 的源头， bolt 可以订阅任意 spout 或 bolt 发出的 tuple 进行处理。



Storm with ckafka

Storm 可以把 CKafka 作为 spout ，消费数据进行处理；也可以作为 bolt ，存放经过处理后的数据提供给其它组件消费。

测试环境

Centos6.8系统

package	version
maven	3.5.0
storm	1.1.0
ssh	5.3
Java	1.8

申请创建 CKafka 实例

创建 Topic

maven 依赖

pom.xml配置如下

```
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-
  <modelVersion>4.0.0</modelVersion>
  <groupId>storm</groupId>
  <artifactId>storm</artifactId>
  <version>0.0.1-SNAPSHOT</version>
  <name>storm</name>
    <properties>
      <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
   </properties>
   <dependencies>
      <dependency>
         <groupId>org.apache.storm</groupId>
         <artifactId>storm-core</artifactId>
         <version>1.1.0</version>
         <scope>provided</scope>
      </dependency>

      <dependency>
         <groupId>org.apache.storm</groupId>
         <artifactId>storm-kafka</artifactId>
         <version>1.1.0</version>
         <exclusions>
```

```
            <exclusion>
              <groupId>org.apache.kafka</groupId>
              <artifactId>kafka-clients</artifactId>
            </exclusion>
          </exclusions>
        </dependency>
        <dependency>
          <groupId>org.apache.storm</groupId>
          <artifactId>storm-kafka-client</artifactId>
          <version>1.1.1</version>
        </dependency>
        <dependency>
          <groupId>org.apache.kafka</groupId>
          <artifactId>kafka_2.11</artifactId>
          <version>0.10.2.1</version>
          <exclusions>
            <exclusion>
              <groupId>org.slf4j</groupId>
              <artifactId>slf4j-log4j12</artifactId>
            </exclusion>
          </exclusions>
        </dependency>
        <dependency>
          <groupId>junit</groupId>
          <artifactId>junit</artifactId>
          <version>3.8.1</version>
        </dependency>
      </dependencies>

      <build>
        <plugins>
          <plugin>
            <artifactId>maven-assembly-plugin</artifactId>
            <configuration>
              <descriptorRefs>
                <descriptorRef>jar-with-dependencies</descriptorRef>
              </descriptorRefs>
              <archive>
                <manifest>
                  <mainClass>ExclamationTopology</mainClass>
                </manifest>
              </archive>
            </configuration>
            <executions>
              <execution>
```

```
                <id>make-assembly</id>
                <phase>package</phase>
                <goals>
                  <goal>single</goal>
                </goals>
              </execution>
            </executions>
          </plugin>
          <plugin>
            <groupId>org.apache.maven.plugins</groupId>
            <artifactId>maven-compiler-plugin</artifactId>
            <configuration>
              <source>1.8</source>
              <target>1.8</target>
            </configuration>
          </plugin>
        </plugins>
    </build>
</project>
```

写入 CKafka

使用 spout/bolt

topology 代码

```
//KafkaProduceTopology.java
import org.apache.storm.Config;
import org.apache.kafka.clients.producer.ProducerConfig;
import org.apache.storm.generated.StormTopology;
import org.apache.storm.kafka.bolt.KafkaBolt;
import org.apache.storm.kafka.bolt.mapper.FieldNameBasedTupleToKafkaMapper;
import org.apache.storm.kafka.bolt.selector.KafkaTopicSelector;
import org.apache.storm.kafka.bolt.selector.DefaultTopicSelector;
import org.apache.storm.topology.TopologyBuilder;
import org.apache.storm.LocalCluster;
import org.apache.storm.StormSubmitter;
import org.apache.storm.topology.TopologyBuilder;
import org.apache.storm.tuple.Fields;
import org.apache.storm.tuple.Values;
import org.apache.storm.utils.Utils;

import java.io.Serializable;
import java.util.Arrays;
import java.util.List;
```

```
import java.util.concurrent.TimeUnit;

import java.util.Properties;

public class KafkaProduceTopology {
    public static void main(String[] args) throws Exception {
        //申请到的ckafka实例ip:port
        String bootstrapServers = "111.230.216.45:9092";
        //指定要将消息写入的topic
        String topic = "storm-topology-test";
          //设置producer属性
          //函数参考：https://kafka.apache.org/0100/javadoc/index.html?org/apache/kafka/clients/consum
          //属性参考：http://kafka.apache.org/0102/documentation.html

          Properties props = new Properties();
          props.put("bootstrap.servers", bootstrapServers);
          props.put("acks", "1");
          props.put("key.serializer", "org.apache.kafka.common.serialization.StringSerializer");
          props.put("value.serializer", "org.apache.kafka.common.serialization.StringSerializer");
        //创建写入kafka的bolt，默认使用fields("key" "message")作为生产消息的key和message，也可以在Fiel
          KafkaBolt kafkaBolt = new KafkaBolt()
           .withProducerProperties(props)
           .withTopicSelector(new DefaultTopicSelector(topic))
           .withTupleToKafkaMapper(new FieldNameBasedTupleToKafkaMapper());

          TopologyBuilder builder = new TopologyBuilder();
          //一个顺序生成消息的spout类，输出field是sentence
        SerialSentenceSpout spout = new SerialSentenceSpout();
        AddMessageKeyBolt bolt = new AddMessageKeyBolt();
         builder.setSpout("spout", spout, 1);
         //为tuple加上生产到ckafka所需要的fields
        builder.setBolt("add-key", bolt, 1).shuffleGrouping("spout");
        //写入ckafka
          builder.setBolt("forwardToKafka", kafkaBolt, 8).shuffleGrouping("add-key");

        Config conf = new Config();
        //conf.setDebug(true);
        if (args != null && args.length > 0) {
            conf.setNumWorkers(1);

            StormSubmitter.submitTopologyWithProgressBar(args[0], conf, builder.createTopology());
        }
        else {
            LocalCluster cluster = new LocalCluster();
             cluster.submitTopology("test", conf, builder.createTopology());
```

```
            Utils.sleep(10000);
            cluster.killTopology("test");
            cluster.shutdown();
        }
    }
}
```

为 tuple 加上 key、message 两个字段，当 key 为 null 时，生产的消息均匀分配到各个 partition，指定了 key 后将按照 key 值 hash 到特定 partition 上

```
//AddMessageKeyBolt.java
public class AddMessageKeyBolt extends BaseBasicBolt {
    public void prepare(Map conf, TopologyContext context, OutputCollector collector) {
    }
    //Add key for message
    public void execute(Tuple tuple, BasicOutputCollector collector) {
        String message = tuple.getString(0);
        collector.emit(new Values(null, message));
    }
    public void declareOutputFields(OutputFieldsDeclarer declarer) {
        declarer.declare(new Fields("key", "message"));
    }
}
```

使用 trident

使用 trident 类生成 topology

```
//TridentKafkaProduceTopology.java
import org.apache.storm.Config;
import org.apache.storm.kafka.trident.TridentKafkaState;
import org.apache.storm.kafka.trident.TridentKafkaStateFactory;
import org.apache.storm.kafka.trident.TridentKafkaUpdater;
import org.apache.storm.kafka.trident.mapper.FieldNameBasedTupleToKafkaMapper;
import org.apache.storm.kafka.trident.selector.DefaultTopicSelector;
import org.apache.storm.LocalCluster;
import org.apache.storm.StormSubmitter;
import org.apache.storm.trident.operation.BaseFunction;
import org.apache.storm.trident.operation.TridentCollector;
import org.apache.storm.trident.Stream;
import org.apache.storm.trident.TridentTopology;
import org.apache.storm.trident.tuple.TridentTuple;
import org.apache.storm.tuple.Fields;
import org.apache.storm.tuple.Values;
import org.apache.storm.utils.Utils;
```

```
import java.io.Serializable;
import java.util.Arrays;
import java.util.concurrent.TimeUnit;
import java.util.List;
import java.util.Properties;

public class TridentKafkaProduceTopology {
    //为tuple加上生产到ckafka所需要的fields
    public static class AddMessageKey extends BaseFunction {
        public void execute(TridentTuple tuple, TridentCollector collector)
        {
            String value = tuple.getString(0);
            int key = value.hashCode();
            //collector.emit(new Values(Integer.toString(key), tuple.getString(0)));
            collector.emit(new Values(null, tuple.getString(0)));
        }
    }
    public static void main(String[] args) throws Exception {
        //申请到的ckafka实例ip:port
        String bootstrapServers = "111.230.216.45:9092";
        //指定要将消息写入的topic
        String topic = "storm-trident-test";

            //设置producer属性
            //函数参考：https://kafka.apache.org/0100/javadoc/index.html?org/apache/kafka/clients/consum
            //属性参考：http://kafka.apache.org/0102/documentation.html
            Properties props = new Properties();
            props.put("bootstrap.servers", bootstrapServers);
            props.put("acks", "1");
            props.put("key.serializer", "org.apache.kafka.common.serialization.StringSerializer");
            props.put("value.serializer", "org.apache.kafka.common.serialization.StringSerializer");

        //一个批量产生句子的spout,输出field为sentence
        TridentSerialSentenceSpout spout = new TridentSerialSentenceSpout(5);
        //设置trident
        TridentKafkaStateFactory stateFactory = new TridentKafkaStateFactory()
            .withProducerProperties(props)
            .withKafkaTopicSelector(new DefaultTopicSelector(topic))
            //设置使用fields("key", "value")作为消息写入
            .withTridentTupleToKafkaMapper(new FieldNameBasedTupleToKafkaMapper("key", "value"));

        TridentTopology builder = new TridentTopology();
        Stream stream = builder.newStream("spout", spout)
            .each(new Fields("sentence"), new AddMessageKey(), new Fields("key", "value"))
```

```
        ;
        stream.partitionPersist(stateFactory, new Fields("key", "value"), new TridentKafkaUpdater(), new Fiel

          Config conf = new Config();
        //conf.setDebug(true);
        if (args != null && args.length > 0) {
            conf.setNumWorkers(1);
            StormSubmitter.submitTopologyWithProgressBar(args[0], conf, builder.build());
        }
        else {
            LocalCluster cluster = new LocalCluster();
             cluster.submitTopology("test", conf, builder.build());
             Utils.sleep(5000);
             cluster.killTopology("test");
             cluster.shutdown();
        }
    }
}
```

从 CKafka 消费

使用 spout/bolt

```
//KafkaConsumeTopology.java
import org.apache.kafka.clients.consumer.ConsumerConfig;
import org.apache.storm.Config;
import org.apache.storm.kafka.spout.KafkaSpout;
import org.apache.storm.kafka.spout.Func;
import org.apache.storm.kafka.spout.KafkaSpoutConfig;
import org.apache.storm.kafka.spout.KafkaSpoutConfig.FirstPollOffsetStrategy;
import org.apache.storm.kafka.spout.KafkaSpoutRetryExponentialBackoff;
import org.apache.storm.kafka.spout.KafkaSpoutRetryExponentialBackoff.TimeInterval;
import org.apache.storm.kafka.spout.KafkaSpoutRetryService;
import org.apache.storm.kafka.spout.trident.KafkaTridentSpoutOpaque;
import org.apache.storm.LocalCluster;
import org.apache.storm.StormSubmitter;
import org.apache.storm.topology.TopologyBuilder;
import org.apache.storm.tuple.Fields;
import org.apache.storm.tuple.Values;
import org.apache.storm.utils.Utils;

import java.io.Serializable;
import java.util.Arrays;
import java.util.List;
import java.util.concurrent.TimeUnit;
```

```
import static org.apache.storm.kafka.spout.KafkaSpoutConfig.FirstPollOffsetStrategy.LATEST;
import static org.apache.storm.kafka.spout.KafkaSpoutConfig.FirstPollOffsetStrategy.EARLIEST;

public class KafkaConsumeTopology {
    public static void main(String[]  args) throws Exception {
        //申请到的ckafka实例ip:port
        String bootstrapServers = "111.230.216.45:9092";
        //指定要将消息写入的topic
        String topic = "storm-topology-test";
        Config conf = new Config();
        //conf.setDebug(true);
        conf.setMaxSpoutPending(20);
        conf.setNumWorkers(1);
        //设置重试策略
        KafkaSpoutRetryService kafkaSpoutRetryService =  new KafkaSpoutRetryExponentialBackoff(
            KafkaSpoutRetryExponentialBackoff.TimeInterval.microSeconds(500),
            KafkaSpoutRetryExponentialBackoff.TimeInterval.milliSeconds(2),
            Integer.MAX_VALUE,
            KafkaSpoutRetryExponentialBackoff.TimeInterval.seconds(10));
        //设置consumer参数
        //函数参考http://storm.apache.org/releases/1.1.0/javadocs/org/apache/storm/kafka/spout/KafkaSp
        //参数参考http://kafka.apache.org/0102/documentation.html
        KafkaSpoutConfig spoutConf =  KafkaSpoutConfig.builder(bootstrapServers, topic)
            .setGroupId("test-group1")    //设置消费groupId
            .setProp(ConsumerConfig.ENABLE_AUTO_COMMIT_CONFIG, "true")    //设置自动确认
            .setProp(ConsumerConfig.SESSION_TIMEOUT_MS_CONFIG, "50000") //设置session超时时间
            .setProp(ConsumerConfig.REQUEST_TIMEOUT_MS_CONFIG, "60000") //设置请求超时时间
            .setOffsetCommitPeriodMs(10_000)    //设置自动确认的时间 ms
            .setFirstPollOffsetStrategy(LATEST)    //设置拉取最新的消息
            .setRetry(kafkaSpoutRetryService)
            .build();

        final TopologyBuilder builder = new TopologyBuilder();
        builder.setSpout("kafka-spout", new KafkaSpout(spoutConf), 1);

        if (args != null && args.length > 0) {
            conf.setNumWorkers(3);
            StormSubmitter.submitTopologyWithProgressBar(args[0], conf, builder.createTopology());
        }
        else {
            LocalCluster cluster = new LocalCluster();
            cluster.submitTopology("test", conf, builder.createTopology());
            Utils.sleep(200000);
            cluster.killTopology("test");
```

```
            cluster.shutdown();
        }
    }
}
```

使用 trident

```
//TridentKafkaTopology.java
import org.apache.kafka.clients.consumer.ConsumerConfig;
import org.apache.storm.Config;
import org.apache.storm.kafka.spout.KafkaSpout;
import org.apache.storm.kafka.spout.Func;
import org.apache.storm.kafka.spout.KafkaSpoutConfig;
import org.apache.storm.kafka.spout.KafkaSpoutConfig.FirstPollOffsetStrategy;
import org.apache.storm.kafka.spout.KafkaSpoutRetryExponentialBackoff;
import org.apache.storm.kafka.spout.KafkaSpoutRetryExponentialBackoff.TimeInterval;
import org.apache.storm.kafka.spout.KafkaSpoutRetryService;
import org.apache.storm.kafka.spout.trident.KafkaTridentSpoutOpaque;
import org.apache.storm.LocalCluster;
import org.apache.storm.StormSubmitter;
import org.apache.storm.trident.Stream;
import org.apache.storm.trident.TridentTopology;
import org.apache.storm.tuple.Fields;
import org.apache.storm.tuple.Values;
import org.apache.storm.utils.Utils;

import java.io.Serializable;
import java.util.Arrays;
import java.util.List;
import java.util.concurrent.TimeUnit;

import static org.apache.storm.kafka.spout.KafkaSpoutConfig.FirstPollOffsetStrategy.EARLIEST;
import static org.apache.storm.kafka.spout.KafkaSpoutConfig.FirstPollOffsetStrategy.LATEST;
import com.william.storm.trident.TridentPrinter;

public class TridentKafkaConsumeTopology {
    public static void main(String[]  args) throws Exception {
        //申请到的ckafka实例ip:port
        String bootstrapServers = "111.230.216.45:9092";
        //指定要将消息写入的topic
        String topic = "storm-trident-test";
        Config conf = new Config();

        conf.setMaxSpoutPending(20);
        conf.setNumWorkers(1);
```

```
        //设置重试策略
        KafkaSpoutRetryService kafkaSpoutRetryService =  new KafkaSpoutRetryExponentialBackoff(
            KafkaSpoutRetryExponentialBackoff.TimeInterval.microSeconds(500),
            KafkaSpoutRetryExponentialBackoff.TimeInterval.milliSeconds(2),
            Integer.MAX_VALUE,
            KafkaSpoutRetryExponentialBackoff.TimeInterval.seconds(10));
        //设置consumer参数
        //函数参考http://storm.apache.org/releases/1.1.0/javadocs/org/apache/storm/kafka/spout/KafkaSp
        //参数参考http://kafka.apache.org/0102/documentation.html
        KafkaSpoutConfig spoutConf =  KafkaSpoutConfig.builder(bootstrapServers, topic)
            .setGroupId("test-group1")    //设置消费groupId
            .setProp(ConsumerConfig.ENABLE_AUTO_COMMIT_CONFIG, "true")    //设置自动确认
            .setProp(ConsumerConfig.SESSION_TIMEOUT_MS_CONFIG, "50000") //设置session超时时间
            .setProp(ConsumerConfig.REQUEST_TIMEOUT_MS_CONFIG, "60000") //设置请求超时时间
            .setOffsetCommitPeriodMs(10_000)    //设置自动确认的时间 ms
            .setFirstPollOffsetStrategy(LATEST)    //设置拉取最新的消息
            .setRetry(kafkaSpoutRetryService)
            .build();

        TridentTopology builder = new TridentTopology();
        final Stream stream = builder.newStream("kafka-spout",
            new KafkaTridentSpoutOpaque(spoutConf))

        if (args != null && args.length > 0) {
            conf.setNumWorkers(3);
            StormSubmitter.submitTopologyWithProgressBar(args[0], conf, builder.build());
        }
        else {
            LocalCluster cluster = new LocalCluster();
            cluster.submitTopology("test", conf, builder.build());
            Utils.sleep(100000);
            cluster.killTopology("test");
            cluster.shutdown();
        }
    }
}
```

提交 Storm

使用 mvn package 编译后,可以提交到本地集群进行 debug 测试，也可以提交到正式集群进行运行

```
storm jar your_jar_name.jar topology_name
```

```
storm jar your_jar_name.jar topology_name tast_name
```

Logstash接入CKafka最佳实践

最近更新时间：2018-02-08 11:34:51

Logstash 简介

Logstash 是一个开源的日志处理工具，它可以从多个源头收集数据、过滤收集的数据以及对数据进行存储作为其他用途。

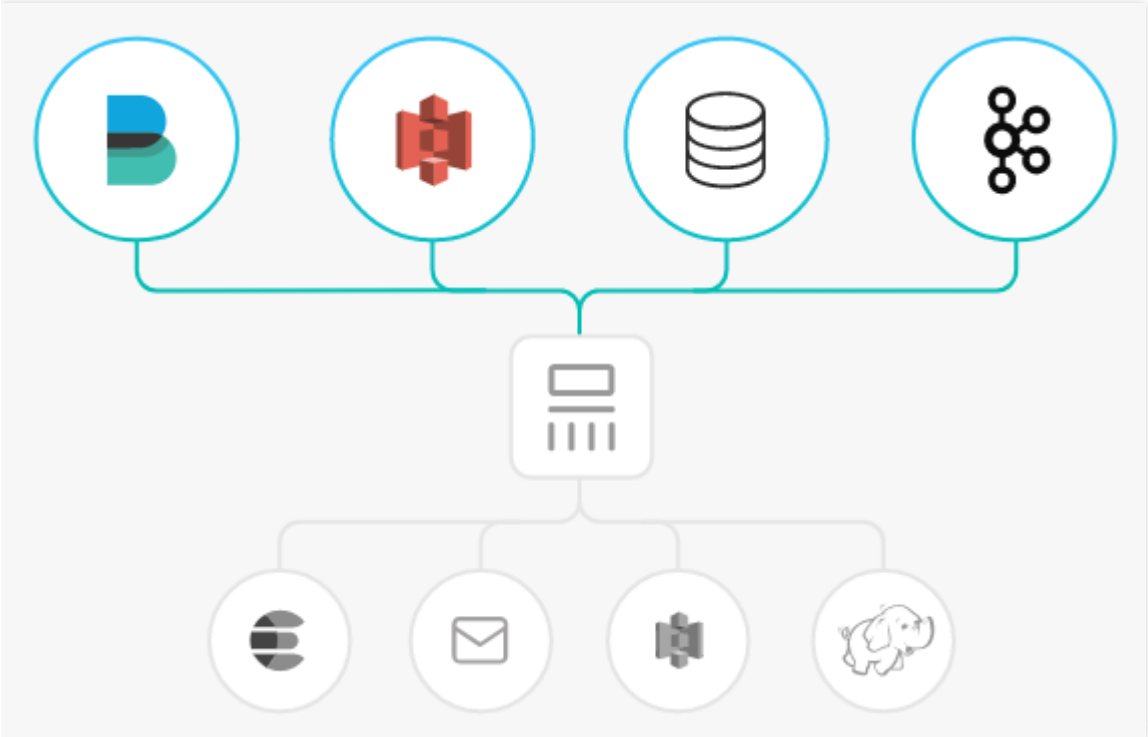
Logstash 灵活性强并且拥有强大的语法分析功能，其插件丰富，支持多种输入和输出源；同时其作为水平可伸缩的数据管道与 Elasticsearch 和 Kibana 配合在日志收集检索方面功能强大。

Logstash 工作原理

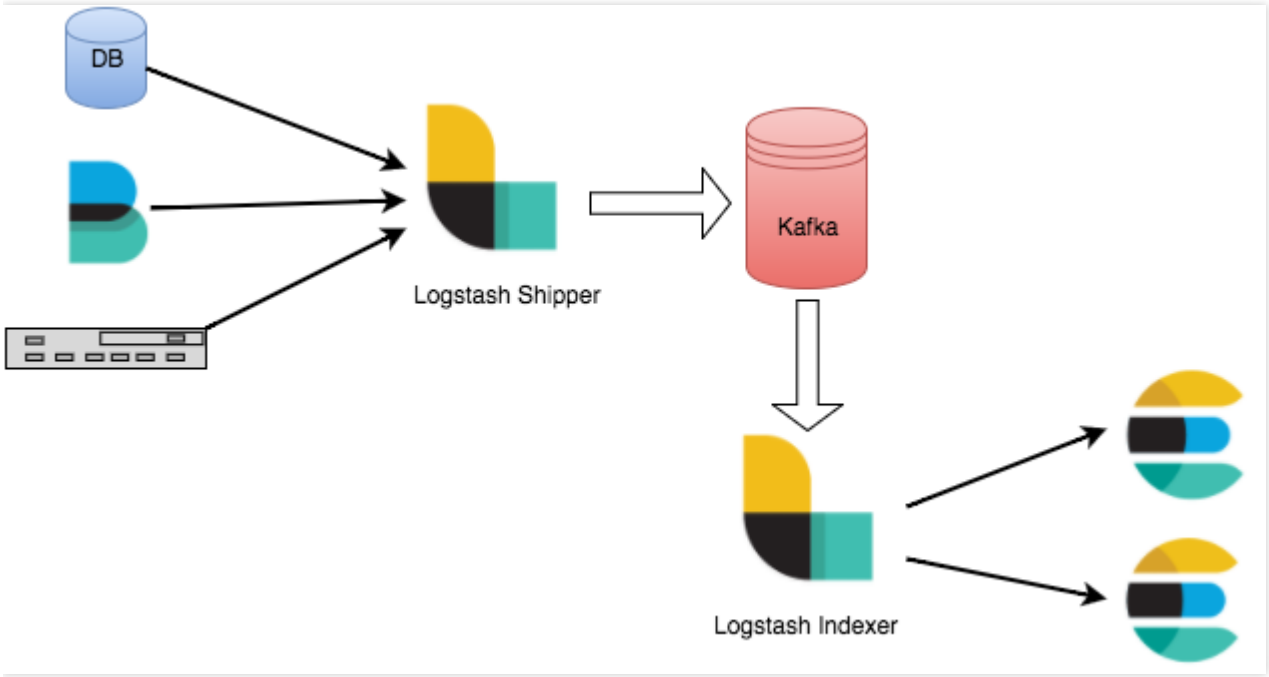
Logstash 数据处理可以分为三个阶段：inputs → filters → outputs。

1. inputs - 产生数据来源，例如文件、syslog、redis 和 beats 此类来源。
2. filter - 修改过滤数据， 在 Logstash 数据管道中属于中间环节，可以根据条件去对事件进行更改。一些常见的过滤器如下：grok、mutate、drop 和 clone 等。
3. outputs - 将数据传输到其他地方，一个事件可以传输到多个 outputs，当传输完成后这个事件就结束。Elasticsearch 就是最常见的 outputs。

同时 Logstash 支持编码解码，可以在 inputs 和 outputs 端指定格式。



Why Logstash + Kafka



1. 可以异步处理数据，防止突发流量。
2. 解耦，当 Elasticsearch 异常的时候不会影响上游工作。
3. Logstash 过滤消耗资源，如果部署在生产 server 上会影响其性能。

CKafka 接入

版本支持

inputs

官网版本兼容性说明如下：

Kafka Client Version	Logstash Version	Plugin Version	Why?
0.8	2.0.0 - 2.x.x	<3.0.0	Legacy, 0.8 is still popular
0.9	2.0.0 - 2.3.x	3.x.x	Works with the old Ruby Event API (`event['product']['price'] = 10`)
0.9	2.4.x - 5.x.x	4.x.x	Works with the new getter/setter APIs (`event.set('[product][price]', 10)`)
0.10.0.x	2.4.x - 5.x.x	5.x.x	Not compatible with the ⇐ 0.9 broker

当前最新版本为 v5.1.8 ，其使用 0.10 版本的 consumer api 进行数据读取。

具体参数配置可见：https://www.elastic.co/guide/en/logstash/current/plugins-inputs-kafka.html

outputs

官网版本兼容性说明如下：

Kafka Client Version	Logstash Version	Plugin Version	Why?
0.8	2.0.0 - 2.x.x	<3.0.0	Legacy, 0.8 is still popular
0.9	2.0.0 - 2.3.x	3.x.x	Works with the old Ruby Event API (`event['product']['price'] = 10`)
0.9	2.4.x - 5.x.x	4.x.x	Works with the new getter/setter APIs (`event.set('[product][price]', 10)`)
0.10.0.x	2.4.x - 5.x.x	5.x.x	Not compatible with the ⇐ 0.9 broker

当前最新版本为 v5.1.7，其使用 0.10 版本的 producer api 进行数据生产。

具体参数配置可见：https://www.elastic.co/guide/en/logstash/current/plugins-outputs-kafka.html

准备工作

- Java 版本：java 8
- Logstash 版本：5.5.2 （ August 17, 2017 ）
- Ckafka 实例，并且创建相应 topic

CKafka 创建

- 拥有实例后，可从控制台中可以看到自己的实例信息

- 点击实例名称可以看到实例分配的具体信息

- 点击 topic管理，创建 topic，此处名字为 **logstash_test**

至此，CKafka 相关的工作环境完成。

CKafka 作为 inputs 接入

1. 执行 bin/logstash-plugin list，查看已经支持的插件是否含有 logstash-input-kafka

```
[root@VM_16_17_centos bin]# ./logstash-plugin list|grep kafka
logstash-input-kafka
logstash-output-kafka
```

2. 编写配置文件 input.conf

此处将标准输出作为数据重点，将 kafka 作为数据来源

```
input{
    kafka {
        bootstrap_servers => "172.16.16.12:9092"
        group_id => "logstash_group"
        topics => ["logstash_test"]
        consumer_threads => 3
        auto_offset_reset => "earliest"
    }
}

output{
    stdout{codec=>rubydebug}
}
```

ckafka实例ip

对应的topic

1. 启动 Logstash，进行消息消费

```
[root@VM_16_17_centos bin]# ./logstash -f input.conf
ERROR StatusLogger No log4j2 configuration file found. Using default configuration: logging only errors to the console.
Sending Logstash's logs to /data/ryan/logstash-5.5.2/logs which is now configured via log4j2.properties
[2017-09-06T18:07:41,926][INFO ][logstash.pipeline        ] Starting pipeline {"id"=>"main", "pipeline.workers"=>4, "pipe
[2017-09-06T18:07:41,943][INFO ][logstash.pipeline        ] Pipeline main started
[2017-09-06T18:07:41,999][INFO ][logstash.agent           ] Successfully started Logstash API endpoint {:port=>9600}
{
    "@timestamp" => 2017-09-06T10:07:42.256Z,
      "@version" => "1",
       "message" => "2017-09-06T09:24:23.039Z localhost ckafka"
}
{
    "@timestamp" => 2017-09-06T10:07:42.256Z,
      "@version" => "1",
       "message" => "2017-09-06T09:23:28.343Z localhost logstash"
}
{
    "@timestamp" => 2017-09-06T10:07:42.256Z,
      "@version" => "1",
       "message" => "2017-09-06T09:23:15.848Z localhost test"
}
```

可以看到刚才 topic 中的数据现在被消费出来

关于kafka作为output的配置更多参数请参考：https://www.elastic.co/guide/en/logstash/current/plugins-inputs-kafka.html#plugins-inputs-kafka-auto_offset_reset

CKafka 作为 outputs 接入

1. 执行 bin/logstash-plugin list，查看已经支持的插件是否含有 logstash-output-kafka

```
[root@VM_16_17_centos bin]# ./logstash-plugin list|grep kafka
logstash-input-kafka
logstash-output-kafka
```

2. 编写配置文件 output.conf

此处将标准输入作为数据来源，将kafka作为数据目的地

```
input {
    stdin{}
}

output {
    kafka {
        bootstrap_servers => "172.16.16.12:9092"
        topic_id => "logstash_test"
    }
}
```

ckafka的实例ip

对应的topic

1. 启动 Logstash，进行消息生产

```
[root@VM_16_17_centos bin]# ./logstash -f output.conf
ERROR StatusLogger No log4j2 configuration file found. Using default configuration: logging only errors to the console.
Sending Logstash's logs to /data/ryan/logstash-5.5.2/logs which is now configured via log4j2.properties
log4j:WARN No appenders could be found for logger (org.apache.kafka.clients.producer.ProducerConfig).
log4j:WARN Please initialize the log4j system properly.
log4j:WARN See http://logging.apache.org/log4j/1.2/faq.html#noconfig for more info.
[2017-09-06T17:22:56,185][INFO ][logstash.pipeline        ] Starting pipeline {"id"=>"main", "pipeline.workers"=>4, "pip
[2017-09-06T17:22:56,202][INFO ][logstash.pipeline        ] Pipeline main started
The stdin plugin is now waiting for input:
[2017-09-06T17:22:56,239][INFO ][logstash.agent           ] Successfully started Logstash API endpoint {:port=>9600}
test
logstash
ckafka
```

启动命令

生产消息

2. 校验刚刚的生产数据

```
[root@VM_16_17_centos bin]# ./kafka-console-consumer.sh --bootstrap-server 172.16.16.12:9092 --topic logstash_test --from-beginning --new-consumer
2017-09-06T09:23:28.343Z localhost logstash
2017-09-06T09:24:23.039Z localhost ckafka
2017-09-06T09:23:15.848Z localhost test
```

关于kafka作为output的配置更多参数请参考：https://www.elastic.co/guide/en/logstash/current/plugins-outputs-kafka.html