

## 目录

1. 试验一：优化 Spark Streaming 配置.....	3
1.1. 实验目的 .....	3
1.2. 实验要求 .....	3
1.3. 实验环境 .....	3
1.4. 试验过程 .....	3
1.4.1. 实验任务一：SparkConf 提交参数 .....	3
1.4.2. 实验任务二：Spark-submit 参数提交 .....	4
1.4.3. 实验任务三：Spark-defaults.conf 优化 .....	6
2. 实验二：优化 Spark 读取 Kafka .....	7
2.1. 实验内容与目标 .....	7
2.2. 实验环境 .....	8
2.3. 实验过程 .....	8
2.3.1. 实验任务一：Spark Streaming 优化读取 Kafka .....	8
3. 实验三：优化 Spark 读取 Flume .....	10
3.1. 实验内容与目标 .....	10
3.2. 实验环境 .....	10
3.3. 实验过程 .....	11
3.3.1. 实验任务一：Flume 推送数据 .....	11
4. 实验四：优化 Spark 写入 HDFS .....	13
4.1. 实验内容与目标 .....	13
4.2. 实验环境 .....	13
4.3. 实验过程 .....	14
4.3.1. 实验任务一：Spark 读取并写入 HDFS .....	14
4.3.2. 实验任务二：显示调用 Hadoop API 写入 HDFS .....	14
4.3.3. 实验任务三：Spark Streaming 实时监控 HDFS .....	15
5. 实验五：优化 Spark Scala 代码 .....	17
5.1. 实验内容与目标 .....	17
5.2. 实验视图 .....	错误!未定义书签。
5.3. 实验环境 .....	17

5.4.	实验过程 .....	17
5.4.1.	实验任务一：Spark 代码优化.....	17



```

/_/
Using Scala version 2.11.8 (Java HotSpot(TM) 64-Bit Server VM, Java 1.8.0_144)
Type in expressions to have them evaluated.
Type :help for more information.

scala>

```

#### 1.4.1.2. 步骤二：SparkConf 参数设置

任何 Spark 程序都是 SparkContext 开始的, SparkContext 的初始化需要一个 SparkConf 对象, SparkConf 包含了 Spark 集群配置的各种参数。Spark 程序通过代码设置, 既在 Scala 编程中按照规则进行设置。如设置运行程序 App 名称。

首先导入 SparkConf 包, 初始化参数后设置, 此步骤在命令框下执行。

```

scala> import org.apache.spark.SparkConf
import org.apache.spark.SparkConf
scala> val conf = new SparkConf().setMaster("master").setAppName("appName")
conf: org.apache.spark.SparkConf = org.apache.spark.SparkConf@6d1c15cb

```

#### 1.4.1.3. 步骤三：SparkConf 连续参数设置

当需要设置的参数过多时可以连续输入:

```

scala> import org.apache.spark.SparkConf
scala> var SparkConf = new SparkConf()
.set("spark.driver.cores","4") //设置 driver 的 CPU 核数
.set("spark.driver.maxResultSize","2g") //设置 driver 端结果存放的最大容量, 这里设置
成为 2G, 超过 2G 的数据, job 就直接放弃, 不运行了
.set("spark.driver.memory","4g") //driver 给的内存大小
.set("spark.executor.memory","8g")// 每个 executor 的内存
.set("spark.streaming.kafka.maxRatePerPartition", "5000") //使用 directStream 方式消
费 kafka 当中的数据, 获取每个分区数据最大速率
.set("spark.streaming.backpressure.enabled", "true") //开启 sparkStreaming 背压机
制, 接收数据的速度与消费数据的速度实现平衡
.set("spark.streaming.backpressure.pid.minRate","10")
....
...
.

```

```

scala> val conf = new SparkConf().set("spark.driver.cores","4").set("spark.driver.maxResultSize","2g").
| set("spark.driver.memory","4g").
| set("spark.executor.memory","8g").
| set("spark.streaming.kafka.maxRatePerPartition", "5000").
| set("spark.streaming.backpressure.enabled", "true").
| set("spark.streaming.backpressure.pid.minRate","10").
| set("spark.files.fetchTimeout","120s")
conf: org.apache.spark.SparkConf = org.apache.spark.SparkConf@8383a14
scala>

```

图 1-1 SparkConf 连续参数设置

### 1.4.2. 实验任务二：Spark-submit 参数提交

#### 1.4.2.1. 步骤一：spark-submit 参数设置

spark-submit 通过 shell 命令框提交已经打包好的 jar 文件, 按照语法设置参数。本实验调用 Spark 自带的样例计算圆周率, 程序命名为“Spark Pi”。--后跟随相关参数名以及

值。

语法: `spark-submit [options] <app jar | python file | R file> [app arguments]`

Hadoop 用户下打开一个新的终端

```
[hadoop@master ~]$ cd /usr/local/src/spark/bin/
```

```
[hadoop@master bin]$ ./spark-submit --class org.apache.spark.examples.SparkPi --master
spark://master:7077 --executor-memory 1g --executor-cores 1
/usr/local/src/spark/examples/jars/spark-examples_2.11-2.0.0.jar 20
```

--class: 导入引入包的类入口。

--master: 设置 Master 服务入口。

--executor-memory: 设置程序可使用内存。

--executor-cores: 设置程序可使用 CPU 核心数。

以及最后程序应用包, 20 为程序参数, 表示为精确到小数点后 20 位。

#### 1.4.2.2. 步骤二: spark-submit 简单优化

优化前:

```
[hadoop@master bin]$ ./spark-submit --class org.apache.spark.examples.SparkPi --master
spark://master:7077 --executor-memory 2g --executor-cores 1
/usr/local/src/spark/examples/jars/spark-examples_2.11-2.0.0.jar 10
```

假设需要加快运算, 设置程序可使用内存为 2g 时, 程序反馈以下结果:

**INFO scheduler.TaskSchedulerImpl: Adding task set 0.0 with 50 tasks**

**WARN scheduler.TaskSchedulerImpl: Initial job has not accepted any resources; check your cluster UI to ensure that workers are registered and have sufficient resources**

```
20/07/18 03:36:40 INFO spark.SparkContext: Created broadcast 0 from broadcast at DAGScheduler.scala:1012
20/07/18 03:36:41 INFO scheduler.DAGScheduler: Submitting 50 missing tasks from ResultStage 0 (MapPartitionsRDD[1] at map at SparkPi.scala:34)
20/07/18 03:36:41 INFO scheduler.TaskSchedulerImpl: Adding task set 0.0 with 50 tasks
20/07/18 03:36:56 WARN scheduler.TaskSchedulerImpl: Initial job has not accepted any resources; check your cluster UI to ensure that workers are registered and have sufficient resources
20/07/18 03:37:11 WARN scheduler.TaskSchedulerImpl: Initial job has not accepted any resources; check your cluster UI to ensure that workers are registered and have sufficient resources
20/07/18 03:37:26 WARN scheduler.TaskSchedulerImpl: Initial job has not accepted any resources; check your cluster UI to ensure that workers are registered and have sufficient resources
```

打开浏览器输入: `master:8080` 查看 sparkUI 界面

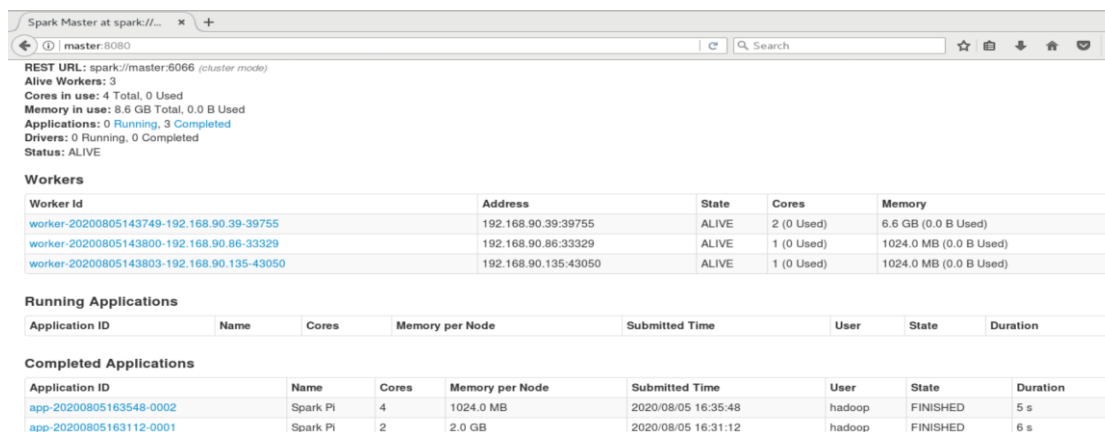
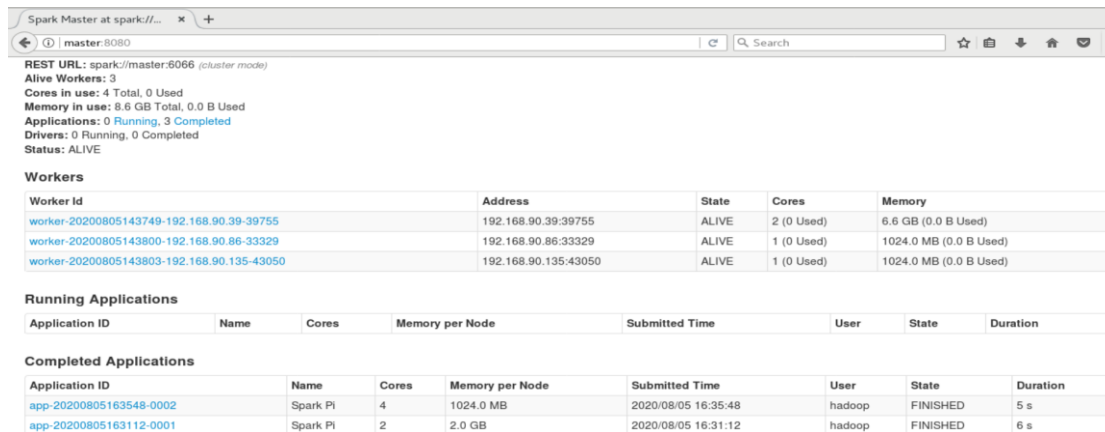


图 1-2 spark-submit 优化前

优化后: 内存修改为 1g。

```
[hadoop@master bin]$ ./spark-submit --class org.apache.spark.examples.SparkPi --master
spark://master:7077 --executor-memory 1g --executor-cores 1
/usr/local/src/spark/examples/jars/spark-examples_2.11-2.0.0.jar 10
```

即要根据项目的需要, 机器的配置, 并不是一味的增加参数能加快运行速度。最后运行结果只需要 5s。



Spark Master at spark://... x +

REST URL: spark://master:6066 (cluster mode)

Alive Workers: 3

Cores in use: 4 Total, 0 Used

Memory in use: 8.6 GB Total, 0.0 B Used

Applications: 0 Running, 3 Completed

Drivers: 0 Running, 0 Completed

Status: ALIVE

### Workers

Worker id	Address	State	Cores	Memory
worker-20200805143749-192.168.90.39-39755	192.168.90.39:39755	ALIVE	2 (0 Used)	6.6 GB (0.0 B Used)
worker-20200805143800-192.168.90.86-33329	192.168.90.86:33329	ALIVE	1 (0 Used)	1024.0 MB (0.0 B Used)
worker-20200805143803-192.168.90.135-43050	192.168.90.135:43050	ALIVE	1 (0 Used)	1024.0 MB (0.0 B Used)

### Running Applications

Application ID	Name	Cores	Memory per Node	Submitted Time	User	State	Duration
----------------	------	-------	-----------------	----------------	------	-------	----------

### Completed Applications

Application ID	Name	Cores	Memory per Node	Submitted Time	User	State	Duration
app-20200805163548-0002	Spark Pi	4	1024.0 MB	2020/08/05 16:35:48	hadoop	FINISHED	5 s
app-20200805163112-0001	Spark Pi	2	2.0 GB	2020/08/05 16:31:12	hadoop	FINISHED	6 s

图 1-3 spark-submit 优化前

## 1.4.3. 实验任务三：Spark-defaults.conf 优化

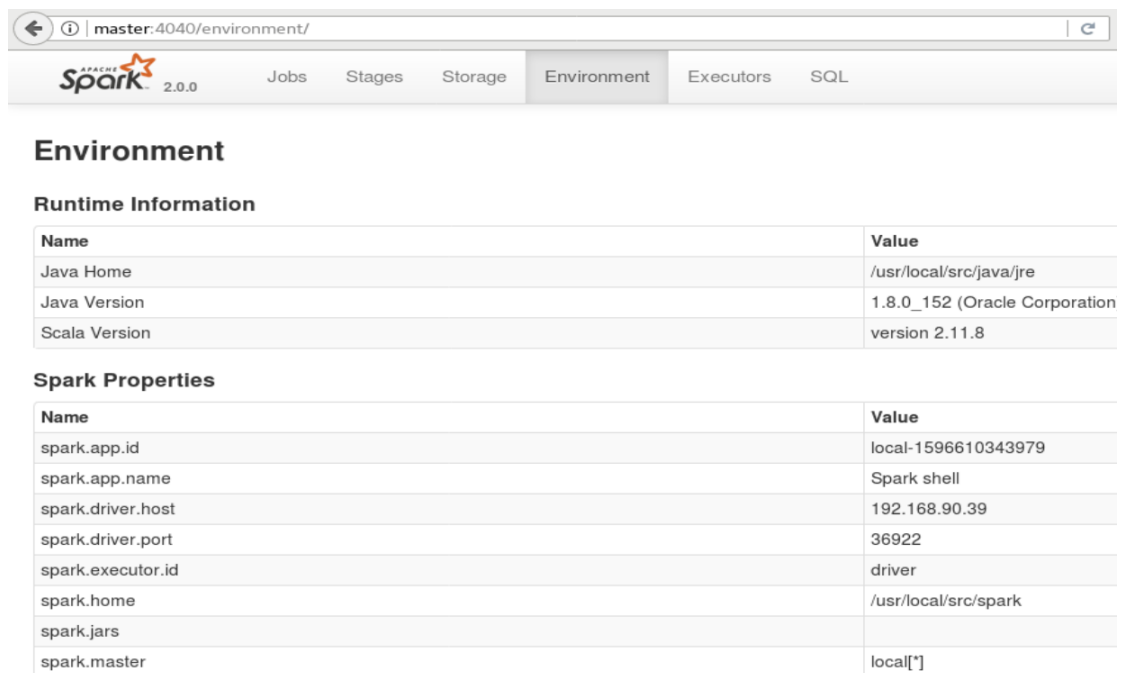
### 1.4.3.1. 步骤一：spark-defaults.conf 参数设置

程序运行时也可以从 conf/spark-defaults.conf 中读取配置选项，这个配置文件中，每一行都包含一对以空格或者等号分开的键和值。例如：

```
spark.master spark://master:7077
spark.executor.memory 512m
spark.eventLog.enabled true
spark.serializer org.apache.spark.serializer.KryoSerializer
```

任何标签指定的值或者在配置文件中的值将会传递给应用程序，并且通过 SparkConf 合并这些值。当提交应用程序时，若参数设置不一致，系统会根据优先级顺序一次对参数进行修改：SparkConf > spark-submit > spark-defaults.conf。

网址 **master:4040** 可以通过 4040 端口查看 Spark 属性，需要注意的是只有当程序运行时才能查看。要根据项目的需要进行修改，需要注意的是，Spark-defaults.conf 修改后为默认参数，任何项目不指定参数时，都会自动按照 Spark-defaults.conf 的设置执行。



master:4040/environment/

SPARK 2.0.0

Jobs Stages Storage **Environment** Executors SQL

### Environment

#### Runtime Information

Name	Value
Java Home	/usr/local/src/java/jre
Java Version	1.8.0_152 (Oracle Corporation)
Scala Version	version 2.11.8

#### Spark Properties

Name	Value
spark.app.id	local-1596610343979
spark.app.name	Spark shell
spark.driver.host	192.168.90.39
spark.driver.port	36922
spark.executor.id	driver
spark.home	/usr/local/src/spark
spark.jars	
spark.master	local[*]

图 1-4 查看 Spark 配置

### 1.4.3.2. 步骤二：Spark Streaming 常用优化

Spark Streaming 是 Spark 的一个核心组件，主要用来处理实时数据流，能够在处理数据时可以通过分布集群方式对读取到的大量实时流数据进行计算。

在程序运行不指定参数或者不修改 `spark-defaults.conf` 参数情况下，Spark Streaming 会调用默认参数执行，本实验以 Spark Streaming 背压机制修改，引入 Spark Streaming 优化。

在默认参数下，Spark Streaming 通过 receivers (或者是 Direct 方式) 以生产者生产数据的速率接收数据。为了 Spark Streaming 应用能在生产中稳定、有效的执行，每批次数据处理时间 (batch processing time) 必须非常接近批次调度的时间间隔 (batch interval)。如果批处理时间一直高于批调度间隔，调度延迟就会一直增长并且不会恢复。最终，Spark Streaming 应用会变得不再稳定。另一方面，如果批处理时间长时间远小于批调度间隔，就会浪费集群资源。

```
[hadoop@master ~]$ cd /usr/local/src/spark/conf/
```

```
[hadoop@master conf]$ vi spark-defaults.conf
```

```
spark.streaming.backpressure.enabled      true
```

```
spark.streaming.backpressure.initialRate  1000
```

通过设置，启动应用程序时，可以有效的防止内存突然间占用过大，限制首次处理的数量。其他常用参数如表 1-2。

表 1-2 spark.streaming 常用优化参数

属性名	默认值	描述
spark.streaming.backpressure.enabled	false	背压机制，根据前一批次数据的处理情况，动态、自动的调整后续数据的摄入量
spark.streaming.backpressure.initialRate	not set	启用背压机制时每个接收器接收第一批数据的初始最大速率
spark.streaming.blockInterval	200ms	接收数据到 Spark 中之前被分块到数据块中的时间间隔。
spark.streaming.receiver.maxRate	not set	每个接收器接收数据的最大速率
spark.streaming.receiver.writeAheadLog.enable	false	启用预写日志，通过接收器接收到的所有输入数据都将保存到预写日志中，以便在驱动程序发生故障后恢复这些数据
spark.streaming.unpersist	True	智能地去持久化 (unpersist) RDD。
spark.streaming.kafka.maxRatePerPartition	not set	Kafka partition 数据的最大速率
spark.streaming.kafka.maxRetries	1	控制读取 kafka 最大数据摄入量
spark.streaming.ui.retainedBatches	1000	日志接口在 gc 时保留的 batch 个数

除此之外更多参数可以根据业务需求查阅官方网站 Spark Streaming Configuration: <http://spark.apache.org/docs/latest/configuration.html#spark-streaming> 修改。

## 2. 实验二：优化 Spark 读取 Kafka

### 2.1. 实验内容与目标

完成本实验，您应该能够：

- 掌握 Spark Streaming 中 Kafka 参数设置

## 2.2. 实验环境

服务器集群	3 个以上节点，节点间网络互通，各节点最低配置：双核 CPU、8GB 内存、100G 硬盘
运行环境	CentOS 7.4 （gui 英文版本）
用户名/密码	root/password hadoop/password
服务和组件	HDFS、YARN、MapReduce 等，其他服务根据实验需求安装

## 2.3. 实验过程

### 2.3.1. 实验任务一：Spark Streaming 优化读取 Kafka

#### 2.3.1.1. 步骤一： 优化批处理时间

Spark Streaming 流处理读取 Kafka 时，由于本身服务器配置以及流量的不可控，当数据量较小，默认配置和使用便能够满足情况，但是当数据量大的时候，就需要进行一定的调整和优化。

```
[hadoop@master ~]$ cd /usr/local/src/spark/conf/
```

```
[hadoop@master conf]$ vi spark-defaults.conf
```

```
spark.streaming.kafka.maxRatePerPartition          10000
```

参数会限制读取所有限制每秒每个消费线程读取每个 kafka 分区最大的数据量，防止通道拥挤，造成内存占用过高。也可以修改 `spark.streaming.receiver.maxRate` 参数，整体对数据流进行限制。

#### 2.3.1.2. 步骤二： 优化数据存储

强制通过 Spark Streaming 生成并持久化的 RDD 自动从 Spark 内存中非持久化。通过 Spark Streaming 接收的原始输入数据也将清除。设置这个属性为 `false` 允许流应用程序访问原始数据和持久化 RDD，因为它们没有被自动清除。但是它会造成更高的内存花费

```
[hadoop@master ~]$ cd /usr/local/src/spark/conf/
```

```
[hadoop@master conf]$ vi spark-defaults.conf
```

```
spark.streaming.unpersist                          true
```

#### 2.3.1.3. 步骤三： 防止数据丢失

确保在 kill 任务时，能够处理完最后一批数据，再关闭程序，不会发生强制 kill 导致数据处理中断，没处理完的数据丢失。

```
[hadoop@master ~]$ cd /usr/local/src/spark/conf/
```

```
[hadoop@master conf]$ vi spark-defaults.conf
```

```
spark.streaming.stopGracefullyOnShutdown          true
```

### 2.3.2. 实验任务二：Kafka 参数简述

#### 2.3.2.1. 步骤一： producer.properties 生产端

Kafka 每秒可以处理几十万条消息，它的延迟最低只有几毫秒，每个 topic 可以分多个 partition, consumer group 对 partition 进行 consume 操作。消息队列的性能好坏，其文件存



储机制设计是衡量一个消息队列服务技术水平和最关键指标之一。

当使用 Kafka 自带的 `kafka-console-producer.sh` 程序生产数据需要配置 `producer.properties` 文件。下面将从 `producer.properties` 生产端配置文件实现数据优化，实现高效数据转运。

```
[hadoop@master ~]$ cd /usr/local/src/kafka/config/
[hadoop@master config]$ vi producer.properties
#在文档最下方添加如下配置
#指定节点列表
metadata.broker.list=master:9092,slaves1:9092, slaves2:9092
#指定分区处理类。默认 kafka.producer.DefaultPartitioner
#partitioner.class=kafka.producer.DefaultPartitioner
#是否压缩,0 代表不压缩,1 代表用 gzip 压缩,2 代表用 snappy 压缩
compression.codec=0
#指定序列化处理类
serializer.class=kafka.serializer.DefaultEncoder
#如果要压缩消息，这里指定哪些 topic 要压缩消息，默认是 empty，表示不压缩
#compressed.topics=
#设置发送数据是否需要服务端的反馈，有三个值 0，1，-1
# 0:producer 不会等待 broker 发送 ack
# 1:当 leader 接收到消息后发送 ack
# -1:当所有的 follower 都同步消息成功后发送 ack
request.required.acks=0
#在向 producer 发送 ack 之前，broker 均需等待的最大时间
request.timeout.ms=10000
#sync 同步（默认），async 异步可以提高发送吞吐量
producer.type=async
#在 async 模式下，当 message 缓存超时后，将会批量发送给 broker,默认 5000ms
#queue.buffering.max.ms=5000
#在 async 模式下，Producer 端允许 buffer 的最大消息量
queue.buffering.max.messages=20000
#在 async 模式下，指定每次批量发送的数据量，默认 200
batch.num.messages=500
#当消息在 producer 端沉积的条数达到“queue.buffering.max.messages”后
#阻塞一定时间后，队列仍然没有 enqueue（producer 仍然没有发送出任何消息）
#此时 producer 可以继续阻塞，或者将消息抛弃
# -1：无阻塞超时限制，消息不会被抛弃
# 0：立即清空队列，消息被抛弃
queue.enqueue.timeout.ms=-1
```

### 2.3.2.2. 步骤二： consumer.properties 消费端

使用 kafka 就是缓冲消息，消费端才是处理消息的中心，数据处理的业务逻辑都在消费者。当数据在通道内被使用后，数据的处理方式，可以根据项目的需要进行。

当使用命令行使用 Kafka 自带的 `kafka-console-consumer.sh` 消费数据时需要配置文件。

```
[hadoop@master ~]$ cd /usr/local/src/kafka/config/
[hadoop@master config]$ vi consumer.properties
#在文档最下方添加如下配置
```

```

#zookeeper 连接服务器地址
zookeeper.connect=master:2181,slaves1:2181, slaves2:2181
#zookeeper 的 session 的过期时间
zookeeper.session.timeout.ms=5000
# timeout in ms for connecting to zookeeper
zookeeper.connectiontimeout.ms=10000
#指定多久消费者更新 offset 到 zookeeper 中
zookeeper.sync.time.ms=2000
#consumer group id
group.id=test-group
#自动向 zookeeper 提交 offset 信息
auto.commit.enable=true
#自动更新时间
auto.commit.interval.ms=1000
#最大取多少块缓存到消费者（默认 10）
queued.max.message.chunks=50
#当有新的 consumer 加入到 group 时，将会 rebalance.
rebalance.max.retries=5
#获取消息的最大尺寸,broker 不会向 consumer 输出大于此值得 chunk
fetch.min.bytes=655360
#当消息尺寸不足时，server 阻塞的时间,如果超时，立即发送给 consumer
fetch.wait.max.ms=5000
socket.receive.buffer.bytes=655360
auto.offset.reset=smallest

```

## 3. 实验三：优化 Spark 读取 Flume

### 3.1. 实验内容与目标

完成本实验，您应该能够：

- 掌握 Flume 配置文件设置

### 3.2. 实验环境

服务器集群	3 个以上节点，节点间网络互通，各节点最低配置：双核 CPU、8GB 内存、100G 硬盘
运行环境	CentOS 7.4 （gui 英文版本）
用户名/密码	root/password hadoop/password
服务和组件	HDFS、YARN、MapReduce 等，其他服务根据实验需求安装

### 3.3. 实验过程

#### 3.3.1. 实验任务一：Spark 充当 Flume 的 sink

##### 3.3.1.1. 步骤一：导入依赖

首先删除 Flume 自带 lib 下的 scala-library-2.10.1.jar 旧版文件

```
[hadoop@master ~]$ cd /usr/local/src/flume/lib/
```

```
[hadoop@master lib]$ rm -rf scala-library-2.10.1.jar
```

把下载后的包复制到 lib 下。

```
[hadoop@master lib]$ cp /opt/software/scala-library-2.11.8.jar /usr/local/src/flume/lib/
```

```
[hadoop@master lib]$ cp /opt/software/spark-streaming-flume-sink_2.11-2.0.0.jar  
/usr/local/src/flume/lib/
```

```
[hadoop@master lib]$ cp /opt/software/spark-streaming-flume_2.11-2.0.0.jar  
/usr/local/src/flume/lib/
```

```
[hadoop@master lib]$ cp /opt/software/spark-streaming-flume_2.11-2.0.0.jar  
/usr/local/src/spark/jars/
```

将 flume 的 jar 包复制到 spark 的 jars 目录下

```
[hadoop@master lib]$ cp /usr/local/src/flume/lib/* /usr/local/src/spark/jars/
```

##### 3.3.1.2. 步骤二：Flume 参数设置

新建配置文件 flume-spark.conf

```
[hadoop@master lib]$ vi /usr/local/src/flume/conf/flume-spark.conf
```

```
a1.sources = r1
```

```
a1.sinks = k1
```

```
a1.channels = c1
```

```
#source
```

```
a1.sources.r1.type = netcat
```

```
a1.sources.r1.bind = master
```

```
a1.sources.r1.port = 3333
```

```
#channel
```

```
a1.channels.c1.type = memory
```

```
a1.channels.c1.capacity = 10000
```

```
a1.channels.c1.transactionCapacity = 1000
```

```
#sink
```

```
a1.sinks.k1.type = avro
```

```
a1.sinks.k1.hostname = master
```

```
a1.sinks.k1.port = 4444
```

```
#相互关联
```

```
a1.sources.r1.channels = c1
```

```
a1.sinks.k1.channel = c1
```

##### 3.3.1.3. 步骤三：启动 Spark Shell

Flume 设置后，由于需要发送给 Spark，此时 Spark 需要先启动，设置接收处理消息。

注意的是，由于 Spark 需要接收消息而且处理消息，Spark cores 需要的核心数应大于 2。

即服务器的 CPU 核心数应大于 2。

```
[hadoop@master bin]$ cd /usr/local/src/spark/bin/
[hadoop@master bin]$ ./spark-shell
scala> import org.apache.spark.SparkConf
scala> import org.apache.spark.streaming.dstream.ReceiverInputDStream
scala> import org.apache.spark.streaming.flume.{FlumeUtils, SparkFlumeEvent}
scala> import org.apache.spark.streaming.{Seconds, StreamingContext}
scala> val ssc = new StreamingContext(sc, Seconds(5))
scala> val          inputDstream:ReceiverInputDStream[SparkFlumeEvent]          =
FlumeUtils.createStream(ssc, "master", 4444)
scala> val tupDstream = inputDstream.map(event=>{
    val event1 = event.event
    val byteBuff = event1.getBody
    val body = new String(byteBuff.array())//ByteBuff.array()
    (body,1)
  }).reduceByKey(_+_ )
scala> tupDstream.print()
scala> ssc.start()
scala> -----
Time: 1595069280000 ms
-----
```

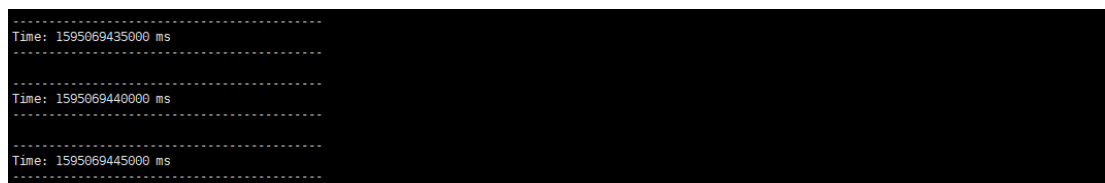


图 1-5 等待数据

系统会按照 5s 的速度进行刷新等待 Flume 推送消息至 Spark。

如果报错 failed to master,这是因为在 HA 环境下是不需要配置 master 的, 所以将 spark-env.sh 配置中的 SPARK\_MASTER\_HOST 和 SPARK\_MASTER\_PORT 这两行注释掉再重启启动即可

#### 3.3.1.4. 步骤四： 启动 Flume

```
[hadoop@master lib]$ cd /usr/local/src/flume/
[hadoop@master flume]$ flume-ng agent --conf conf --conf-file
/usr/local/src/flume/conf/flume-spark.conf --name a1
-Dflume.root.logger=INFO,console
Info: Including Hadoop libraries found via (/usr/local/src/hadoop/bin/hadoop) for HDFS
access
.....
...
.
20/07/18 06:50:11 WARN api.NettyAvroRpcClient: Using default maxIOWorkers
20/07/18 06:50:11 INFO sink.AbstractRpcSink: Rpc sink k1 started.
```

注意的是，flume 监听 3333 端口，然后再通过 4444 端口发送给 Spark。

3.3.1.5. 步骤五： nc 写消息

打开新的命令框。

```
[hadoop@master ~]$ nc master 3333
hello
OK
Wang ^H
OK
Zhangsan
OK
12345678
OK
```

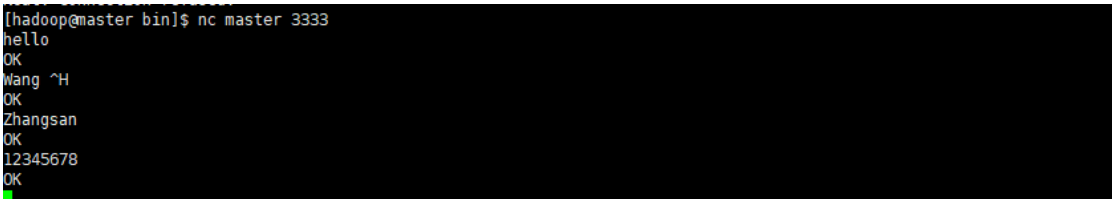


图 1-6 输入数据

此时可以从 Spark 界面处看到接受的数据，并对数据进行相关的处理。

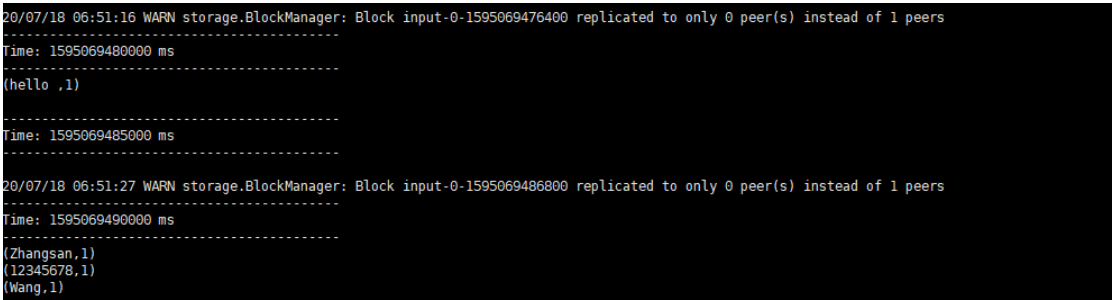


图 1-7 Spark 展示

# 4. 实验四：优化 Spark 写入 HDFS

## 4.1. 实验内容与目标

完成本实验，您应该能够：

- 掌握 Spark 读取 HDFS
- 掌握 Spark 实时监控 HDFS

## 4.2. 实验环境

服务器集群	3 个以上节点，节点间网络互通，各节点最低配置：双核 CPU、8GB 内存、100G 硬盘
运行环境	CentOS 7.4 （gui 英文版本）
用户名/密码	root/password hadoop/password

服务和组件 HDFS、YARN、MapReduce 等，其他服务根据实验需求安装

## 4.3. 实验过程

### 4.3.1. 实验任务一：Spark 读取并写入 HDFS

在 HDFS 根目录新建测试文件，文件内容如下。

```
[hadoop@master ~]$ vi test.txt
```

```
WangWu 12
```

```
LiuSi 34
```

```
Sun 78
```

```
[hadoop@master ~]$ hadoop fs -put ~/test.txt /
```

回到之前打开的 scala 终端页面

```
scala> val lines = sc.textFile("hdfs://master:8020/test.txt")
```

```
scala> lines.saveAsTextFile("hdfs://master:8020/copy_test")
```

进入网址 master:50070 点击 utilities 查看文件

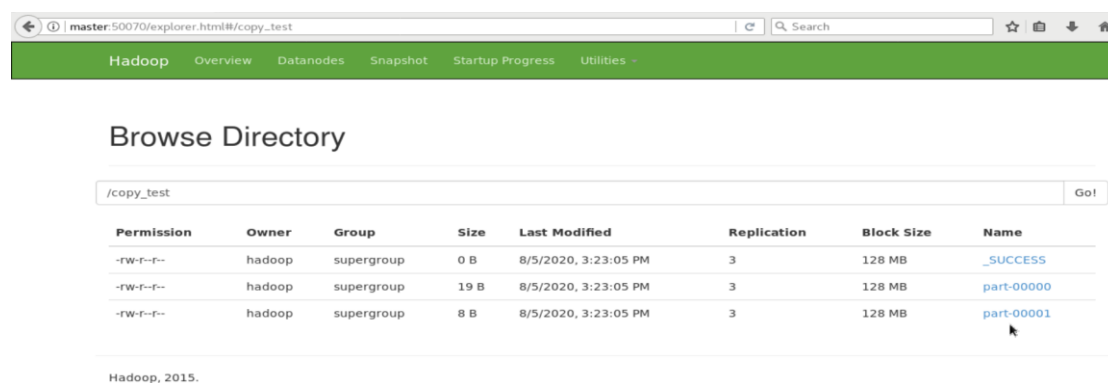


图 1-8 测试文件

### 4.3.2. 实验任务二：显示调用 Hadoop API 写入 HDFS

#### 4.3.2.1. 步骤一：saveAsNewAPIHadoopFile 函数

saveAsNewAPIHadoopFile 为新版数据输出 HDFS 工具，默认使用 TextOutputFormat 实现类。首先需要导入包。

```
scala> import org.apache.hadoop.io.{IntWritable, Text}
```

```
scala> import org.apache.hadoop.mapreduce.lib.output.TextOutputFormat
```

```
scala> import org.apache.spark.{SparkConf, SparkContext}
```

```
scala> val rddData = sc.parallelize(List(("WangWu ", 12), ("LiuSi dog", 34), ("Sun ", 78), ("Alex", 10)), 1)
```

```
scala> val path = "hdfs://master:8020/test_write.txt "
```

```
scala> rddData.saveAsNewAPIHadoopFile(path, classOf[Text], classOf[IntWritable], classOf[TextOutputFormat[Text, IntWritable]])
```

可通过 Hadoop WebUI 查看文件是否存在。

进入网址 master:50070 点击 utilities 查看文件

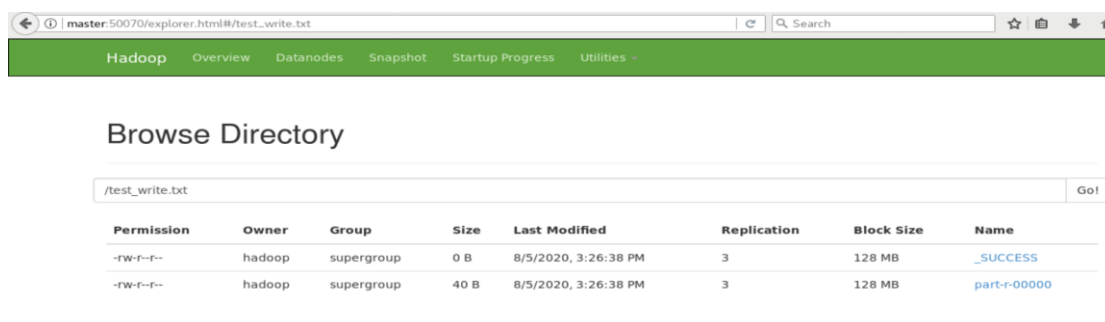


图 1-9 查看文件

### 4.3.3. 实验任务三：Spark Streaming 实时监控 HDFS

#### 4.3.3.1. 步骤一：HDFS 新建临时文件夹

HDFS 上新建文件夹：Spark\_HDFS 作为 Spark 监控 HDFS 使用。

```
[hadoop@master ~]$ hadoop fs -mkdir /Spark_HDFS
```

```
[hadoop@master ~]$ hadoop fs -ls /
```

Found 12 items

```
drwxr-xr-x  - hadoop supergroup      0 2020-08-05 15:52 /Spark_HDFS
drwxr-xr-x  - hadoop supergroup      0 2020-08-05 09:07 /benchmarks
drwxr-xr-x  - hadoop supergroup      0 2020-08-05 15:23 /copy_test
drwxr-xr-x  - hadoop supergroup      0 2020-08-05 11:20 /input1
drwxr-xr-x  - hadoop supergroup      0 2020-08-05 11:15 /input2
drwxr-xr-x  - hadoop supergroup      0 2020-08-05 13:44 /output1
drwxr-xr-x  - hadoop supergroup      0 2020-08-05 13:52 /output2
drwxr-xr-x  - hadoop supergroup      0 2020-08-05 13:59 /output3
-rw-r--r--  3 hadoop supergroup    27 2020-08-05 15:17 /test.txt
drwxr-xr-x  - hadoop supergroup      0 2020-08-05 15:26 /test_write.txt
drwxrwx---  - hadoop supergroup      0 2020-08-04 17:25 /tmp
drwx-----  - hadoop supergroup      0 2020-08-04 17:41 /user
```

#### 4.3.3.2. 步骤二：编写代码实时监控 spark\_HDFS 文件夹

进入到 Spark-shell 界面，调用两个核心。

```
[hadoop@master bin] cd /usr/local/src/spark/bin/
```

```
[hadoop@master bin]$ ./spark-shell --master local[2]
```

Setting default log level to "WARN".

To adjust logging level use sc.setLogLevel(newLevel).

20/07/16 09:47:21 WARN util.NativeCodeLoader: Unable to load native-hadoop library for your platform... using builtin-java classes where applicable

20/07/16 09:47:23 WARN spark.SparkContext: Use an existing SparkContext, some configuration may not take effect.

Spark context Web UI available at http://192.168.88.150:4040

Spark context available as 'sc' (master = local[2], app id = local-1594907243082).

Spark session available as 'spark'.

Welcome to

The diagram shows a series of connected line segments. The top part has a horizontal line with a small vertical tick on the left and a small vertical tick on the right. Below this, there are several horizontal segments connected by vertical ones. Some segments have arrows pointing to the right. There are labels 'V' and 'J' placed near some of the segments. The bottom part of the diagram has a horizontal line with a small vertical tick on the left and a small vertical tick on the right, similar to the top part.

Type in expressions to have them evaluated.

```
scala> import org.apache.spark.streaming.{Seconds, StreamingContext}
```

```
scala> import org.apache.spark.{SparkConf, SparkContext}
```

```
scala> val ssc = new StreamingContext(sc, Seconds(5))
```

org.apache.spark.streaming.StreamingContext@7bbc3b6e

```
lines:      org.apache.spark.streaming.dstream.DStream[String]      =
```

```
scala> lines.print()
```

```
scala> -----
```

-----

-----

---

```
ala> ssc.start()

ala> -----
me: 1594907440000 ms
-----

-----
me: 1594907450000 ms
-----

-----
me: 1594907460000 ms
```

#### 4.3.3.3. 步骤三：上传文件至 HDFS

```
[hadoop@master ~]$ hadoop fs -put ~/test.txt /Spark HDFS
```



```

-----
Time: 1594908045000 ms
-----
WangWu 12
LiuSi 34
Sun 78
-----
Time: 1594908050000 ms
-----

```

图 1-11 监听数据

此时 Spark Streaming 自动读取/Spark\_HDFS 下的文件内容并输出的界面上。

## 5. 实验五：优化 Spark Scala 代码

### 5.1. 实验内容与目标

完成本实验，您应该能够：

- 掌握 Scala RDD 数据持久化

### 5.2. 实验环境

服务器集群	3 个以上节点，节点间网络互通，各节点最低配置：双核 CPU、8GB 内存、100G 硬盘
运行环境	CentOS 7.4 （gui 英文版本）
用户名/密码	root/password hadoop/password
服务和组件	HDFS、YARN、MapReduce 等，其他服务根据实验需求安装

### 5.3. 实验过程

#### 5.3.1. 实验任务一：Scala 简单编程代码优化

##### 5.3.1.1. 步骤一：模式匹配

Scala 内部自带一个强大的功能：模式匹配机制。一个模式匹配包含了一系列备选项，每个都开始于关键字 `case`。每个备选项都包含了一个模式及一到多个表达式。箭头符号 `=>` 隔开了模式和表达式。

Scala 没有 `switch case` 用法，但是提供了 `match case`。`match case` 可以匹配各种状况以及数据类型。

语法：[value] match { case 值 => [code] }

优化前需要通过使用 `for` 循环以及 `if` 判断，优化后则可以降低代码适应程度。

```
scala> def fun_match(x: Any): Any = x match {
      case 0 => "num Int"
```

```

    case "num" => 2
    case y: Int => "scala.Int"
    case _ => "other type"
  }
fun_match: (x: Any)Any
scala> println(fun_match("one"))
other type
scala> println(fun_match("num"))
2
scala> println(fun_match(1))
scala.Int
scala> println(fun_match(6))
scala.Int
scala> println(fun_match(0))
num Int

```

#### 5.3.1.2. 步骤二：正则表达式

Scala 通过 `scala.util.matching` 包中的 `Regex` 类来支持正则表达式。正则表达式是用来找出数据中的指定模式（或缺少该模式）的字符串。“`.r`”方法可使任意字符串变成一个正则表达式。

本实例首先定义 `String` 字符串,通过`.r`构建一个 `Regex` 类型数据。然后使用 `findFirstIn` 方法找到首个匹配项。

```

scala> val str = "H3C".r
str: scala.util.matching.Regex = H3C
scala> val lang = "H3C is a famous company"
lang: String = H3C is a famous company
scala> println(str findFirstIn lang)
Some(H3C)

```

如果不适用正则表达式，则会重复嵌套使用循环、判断语句。使用过后会大大加快运行速度以及节省代码空间。其他正则表达式可以参考官方文档进行查看。

### 5.3.2. 实验任务二：Scala 代码优化(数据)

#### 5.3.2.1. 步骤一：RDD 缓存

Scala 代码优化部分偏向于数据处理，由于 `Spark` 运行时数据在内存中占据着大量的空间，通过对数据的优化可以释放内存，增加运行的任务数，进而加快运行速度。

`RDD` 为 `Spark` 提供了丰富的数据处理操作，为了加快处理速度可以将长时间处理的数据放在内存上。

```

scala> val lines = sc.textFile("hdfs://master:8020/test.txt")
lines: org.apache.spark.rdd.RDD[String] = hdfs://master:8020/test.txt MapPartitionsRDD[
scala> lines.cache
res0: lines.type = hdfs://master:8020/test.txt MapPartitionsRDD[1] at textFile at <cons
scala> lines.count()
res2: Long = 4
进入网址 master:4040

```

RDD Name	Storage Level	Cached Partitions	Fraction Cached	Size in Memory	Size on Disk
hdfs://master:9000/test.txt	Memory Deserialized 1x Replicated	1	100%	248.0 B	0.0 B

图 1-12 缓存数据

### 5.3.2.2. 步骤二：Checkpoint 机制

当 RDD 缓存过多时，内存会被大量占用，此时可以使用 Checkpoint 机制，设置检查点，将数据暂时放置在本地磁盘上，当需要引用数据时会迅速调用。

```
scala> sc.setCheckpointDir("HDFS://master:8020/spark_checkpoint")
```

```
scala> val lines = sc.textFile("hdfs://master:8020/test.txt")
```

```
lines:      org.apache.spark.rdd.RDD[String]      =      hdfs://master:8020/test.txt
```

```
MapPartitionsRDD[3] at textFile at <console>:24
```

```
scala> lines.checkpoint
```

```
scala> lines.unpersist(true)
```

通过实例化 lines 会存储在 HDFS://master:8020/spark\_checkpoint 目录，当数据使用完毕后要释放缓存。

在网址 master:50070 下的 utilities 查看

Permission	Owner	Group	Size	Last Modified	Replication	Block Size	Name
drwxr-xr-x	hadoop	supergroup	0 B	8/5/2020, 4:05:45 PM	0	0 B	8407b366-6ba3-49f6-97c8-8d6820c88f6f

图 1-13 持久化点

### 5.3.2.3. 步骤三：数据的广播

由于内存有限，Spark 在计算时有可能对同一份数据在不同的任务，不同的时间内计算。此时如果重复的读取数据，则会造成数据的拥挤。

例如有一份数据为 200M，Spark 有 50 个任务需要对同一份数据进行处理，有可能是排序、查找、定位、删除等操作，并返回不同的结果。如果每一个任务都读取数据后存储在内存中需要的空间为：200\*50=10000M，对内存的需要巨大。

此时需要数据的广播，共享数据转换成广播变量。

需要注意的是，一旦一个 broadcast 初始化好了，今后对它的值的访问只能通过 broadcast。

```
scala> val broadcastVar = sc.broadcast(Array(1, 2, 3, 4, 5, 6, 7, 8, 9))
```

```
broadcastVar: org.apache.spark.broadcast.Broadcast[Array[Int]] = Broadcast(0)
```

```
scala> broadcastVar.value
```

```
res0: Array[Int] = Array(1, 2, 3, 4, 5, 6, 7, 8, 9)
```