Hadoop 下 Pipes 调用 C++ 实例程序

-------- Pipes 管道技术

Hadoop 管道是 Hadoop MapReduce 的 C++接口的代称。与流不同，流使用标准输入和输出让 map 和 reduce 节点之间相互交流，管道使用 sockets 作为 tasktracker 与 C++编写的 map 或者 reduce 函数的进程之间的通道。JNI 未被使用。

我们将用 C++重写贯穿 WordCount 示例，然后，我们将看到如何使用管道来运行它。下面的例子显示了用 C++语言编写的 map 函数和 reduce 函数的源代码。

用 C++语言编写的 WordCount 词频统计程序

```
/*
 * WordCount.h
 *
 *  Created on: 2016-4-19
 *      Author: wang
 */

#ifndef WORDCOUNT_H_
#define WORDCOUNT_H_

class WordCount {
public:
       WordCount();
       virtual ~WordCount();
};

#endif /* WORDCOUNT_H_ */



/*
 * WordCount.cpp
 *
 *  Created on: 2016-4-19
 *      Author: wang
 */

#include "Pipes.hh"
#include "TemplateFactory.hh"
#include "StringUtils.hh"
#include<iostream>
#include<string.h>
using namespace std;
class WordCountMap: public HadoopPipes::Mapper {
public:
  WordCountMap(HadoopPipes::TaskContext &context){}
  void map(HadoopPipes::MapContext &context) {
  vector<string> words =HadoopUtils::splitString( context.getInputValue(), " ");;
    for(unsigned int i=0; i < words.size(); ++i) {
      context.emit(words[i], "1");
    }
  }
};

class WordCountReduce: public HadoopPipes::Reducer {
public:
  WordCountReduce(HadoopPipes::TaskContext& context){}
  void reduce(HadoopPipes::ReduceContext& context) {
    int sum = 0;
    while (context.nextValue()) {
       sum += HadoopUtils::toInt(context.getInputValue());
    }
    context.emit(context.getInputKey(), HadoopUtils::toString(sum));
  }

};

int main(int argc, char *argv[]) {
  return HadoopPipes::runTask(HadoopPipes::TemplateFactory<WordCountMap, WordCountReduce>());
}
```

程序说明：

此应用程序连接到 Hadoop C++库，后者是一个用于与 tasktracker 子进程进行通信的轻量级的封装器。通过扩展在 HadoopPipes 命名空间的 Mapper 和 Reducer 类且提供 map()和 reduce()方法的实现，我们便可定义 map 和 reduce 函数。这些方法采用了一个上下文对象(MapContext 类型或 ReduceContext 类型)，后者提供读取输入和写入输出，通过 JobConf 类来访问作业配置信息。本例中的处理过程非常类似于 Java 的处理方式。

与 Java 接口不同，C++接口中的键和值是字节缓冲，表示为标准模板库(Standard Template Library，STL)的字符串。这使接口变得更简单，尽管它把更重的负担留给了应用程序的开发人员，因为开发人员必须将字符串 convert to and from 表示 to 和 from 两个逆操作。开发人员必须在字符串及其他类型之间进行转换。这一点在 MapTemperatureReducer 中十分明显，其中，我们必须把输入值转换为整数的输入值(使用 HadoopUtils 中的便利方法)。main()方法是应用程序的入口点。它调用 HadoopPipes::runTask，连接到从 Mapper 或 Reducer 连接到 Java 的父进程和 marshals 数据。runTask()方法被传入一个 Factory 参数，使其可以创建 Mapper 或 Reducer 的实例。它创建的其中一个将受套接字连接中 Java 父进程控制。我们可以用重载模板 factory 方法来设置一个 combiner(combiner)、partitioner(partitioner)、记录读取函数(record reader)或记录写入函数(record writer)。

编译运行

现在我们可以用 makefile 编译连接程序。

C++版本的 WordCount MapReduce 程序的 makefile,在工程下的一级目录下新建一个 makefile

```
CC = g++
HADOOP_INSTALL = /usr/local/development/hadoop-2.6.0
INC = -I$(HADOOP_INSTALL)/include
LIBS = -L$(HADOOP_INSTALL)/lib/native -lhadooppipes -lhadooputils -lpthread -lssl -lcrypto
CPPFLAGS = $(INC) -Wall -g -O2
```

```
WordCount: src/WordCount.cpp
	$(CC) $(CPPFLAGS) -o $@ $< $(LIBS)
```

首先，在写此程序之前，Hadoop 我默认为已经安装好了，并且相关配置也没问题的前提下。对于上面 makefile 中的一些变量，HADOOP_INSTALL/include 下有我们 C++的一些头文件，必须在此处声明包含进来，另外，HADOOP_INSTALL/lib/native 下面的 Hadoop 提供的共享库也必须包含进来。可以看出，我的 cpp 源文件是放在 src 下的，这点在 makefile 中要注意路径，不然编译肯定会失败。

安装依赖库：

在正式编译之前，我们必须先安装 C++的依赖库

```
apt-get update && apt-get install gcc g++ make automake libtool glibc-devel libssl-dev openssl -y
|| yum install gcc g++ make automake libtool glibc-devel libssl-dev openssl -y
```

上面的这些包如果一个命令安装失败，请分别安装。

现在文件目录如下：

```
|-- src

|     |-- WordCount.cpp

|     |-- WordCount.h

| -- makefile
```

编译：

```
make

make
g++ -I/usr/local/development/hadoop-2.6.0/include -Wall -g -O2 -o wordcount wordcount.cpp
-L/usr/local/development/hadoop-2.6.0/lib/native -lhadooppipes -lhadooputils -lpthread -lssl
-lcrypto

**** Build Finished ****
```

成功编译后在工程根目录下生成了可执行文件 WordCount

```
|-- src

|     |-- WordCount.cpp

|     |-- WordCount.h

| -- makefile

| -- WordCount
```

配置 yarn-site.xml

接下来运行管道作业，我们需要在 Hadoop 伪分布式(pseudo_distrinuted)模式下运行。管道不在本地模式中运行，因为它依赖于 Hadoop 的分布式缓存机制，仅在 HDFS 运行时才运行。

在启动 Hadoop 之前，我们还需要做一个配置文件 yarn-site.xml,将如下 xml 代码添加到文件

中:

```
.... ....
<property>
        <name>hadoop.pipes.java.recordreader</name>
        <value>true</value>
    </property>
    <property>
        <name>hadoop.pipes.java.recordwriter</name>
        <value>true</value>
    </property>
```

添加此配置后为的是方便后面提交作业。

启动 Hadoop

```
$ start-all.sh
This script is Deprecated. Instead use start-dfs.sh and start-yarn.sh
Starting namenodes on [localhost]
localhost: starting namenode, logging to /usr/local/development/hadoop-2.6.0/logs/hadoop-wang-namenode-kubuntu.out
localhost: starting datanode, logging to /usr/local/development/hadoop-2.6.0/logs/hadoop-wang-datanode-kubuntu.out
Starting secondary namenodes [localhost]
localhost: starting secondarynamenode, logging to /usr/local/development/hadoop-2.6.0/logs/hadoop-wang-secondarynamenode-kubuntu.out
starting yarn daemons
starting resourcemanager, logging to /usr/local/development/hadoop-2.6.0/logs/yarn-wang-resourcemanager-kubuntu.out
localhost: starting nodemanager, logging to /usr/local/development/hadoop-2.6.0/logs/yarn-wang-nodemanager-kubuntu.out
```

Hadoop5 个守护进程开始运行后，第一步是把可执行文件复制到 HDFS，以便它们启动 map 和 reduce 任务时，它能够被 tasktracker 取出：

```
$ hadoop dfs -put WordCount in/
```

示例数据也需要从本地文件系统复制到 HDFS：

```
$ hadoop fs -put file1.txt in/
```

现在可以运行这个作业。为了使其运行，我们用 Hadoop 管道命令，使用-program 参数来传递在 HDFS 中可执行文件的 URI。

```
$ hadoop pipes -input in/file1.txt -output out/cwordcount -program in/WordCount
```

为了进一步提交作业方便，我将上面一条命令放到了 shell 脚本里面执行了，取了个名 run_wordcount.sh，内容像这样写：

```
#! /bin/bash

hadoop pipes -input in/file1.txt -output out/cwordcount -program in/WordCount
```

为 run_wordcount.sh 添加可执行权限，这样子后，每次我想提交作业，只需执行一下 run_wordcount.sh 这个命令就可以了

```
$ ./run_wordcount.sh

DEPRECATED: Use of this script to execute mapred command is deprecated.

Instead use the mapred command for it.


16/04/25 23:24:42 INFO client.RMProxy: Connecting to ResourceManager at /0.0.0.0:8032

16/04/25 23:24:43 INFO client.RMProxy: Connecting to ResourceManager at /0.0.0.0:8032

16/04/25 23:24:44 WARN mapreduce.JobSubmitter: No job jar file set.  User classes may not be found. See Job or Job#setJar(String).

16/04/25 23:24:44 INFO mapred.FileInputFormat: Total input paths to process : 1

16/04/25 23:24:44 INFO mapreduce.JobSubmitter: number of splits:2

16/04/25 23:24:45 INFO mapreduce.JobSubmitter: Submitting tokens for job: job_1461595534083_0003

16/04/25 23:24:45 INFO mapred.YARNRunner: Job jar is not present. Not adding any jar to the list of resources.

16/04/25 23:24:45 INFO impl.YarnClientImpl: Submitted application application_1461595534083_0003

16/04/25 23:24:45 INFO mapreduce.Job: The url to track the job: http://kubuntu:8088/proxy/application_1461595534083_0003/

16/04/25 23:24:45 INFO mapreduce.Job: Running job: job_1461595534083_0003

16/04/25 23:24:58 INFO mapreduce.Job: Job job_1461595534083_0003 running in uber mode : false

16/04/25 23:24:58 INFO mapreduce.Job:  map 0% reduce 0%

16/04/25 23:25:18 INFO mapreduce.Job:  map 33% reduce 0%

16/04/25 23:25:19 INFO mapreduce.Job:  map 100% reduce 0%

16/04/25 23:25:33 INFO mapreduce.Job:  map 100% reduce 100%

16/04/25 23:25:35 INFO mapreduce.Job: Job job_1461595534083_0003 completed successfully

16/04/25 23:25:36 INFO mapreduce.Job: Counters: 49

        File System Counters

                FILE: Number of bytes read=91

                FILE: Number of bytes written=321607

                FILE: Number of read operations=0

                FILE: Number of large read operations=0

                FILE: Number of write operations=0

                HDFS: Number of bytes read=254

                HDFS: Number of bytes written=63

                HDFS: Number of read operations=9

                HDFS: Number of large read operations=0
                HDFS: Number of write operations=2
        Job Counters
                Launched map tasks=2
                Launched reduce tasks=1
                Data-local map tasks=2
                Total time spent by all maps in occupied slots (ms)=37294
                Total time spent by all reduces in occupied slots (ms)=13062
                Total time spent by all map tasks (ms)=37294
                Total time spent by all reduce tasks (ms)=13062
                Total vcore-seconds taken by all map tasks=37294
                Total vcore-seconds taken by all reduce tasks=13062
                Total megabyte-seconds taken by all map tasks=38189056
                Total megabyte-seconds taken by all reduce tasks=13375488
        Map-Reduce Framework
                Map input records=2
                Map output records=11
                Map output bytes=63
                Map output materialized bytes=97
                Input split bytes=192
                Combine input records=0
                Combine output records=0
                Reduce input groups=11
                Reduce shuffle bytes=97
                Reduce input records=11
                Reduce output records=11
                Spilled Records=22
                Shuffled Maps =2
                Failed Shuffles=0
                Merged Map outputs=2
                GC time elapsed (ms)=219
                CPU time spent (ms)=4590
                Physical memory (bytes) snapshot=721281024
                Virtual memory (bytes) snapshot=2502393856
                Total committed heap usage (bytes)=524288000
        Shuffle Errors
                BAD_ID=0
                CONNECTION=0
                IO_ERROR=0
                WRONG_LENGTH=0
                WRONG_MAP=0
                WRONG_REDUCE=0
        File Input Format Counters
                Bytes Read=62
        File Output Format Counters
                Bytes Written=63
16/04/25 23:25:36 INFO util.ExitUtil: Exiting with status 0
```

读了 62 个字节，写了 63 个字节，退出状态是 0, 执行成功！

接下来我们进行查看一下我们统计的词频：

```
$ hdfs dfs -cat out/cwordcount/part-00000
,       1
.       1
?       1
It's    1
No      1
long    1
my      1
name    1
not     1
wang    1
yuan    1
```

参考网址：
1. https://github.com/alexanderkoumis/hadoop-wordcount-cpp

2. http://hadoop.apache.org/docs/current/hadoop-project-dist/hadoop-hdfs/LibHdfs.html

3. http://blog.csdn.net/lxxgreat/article/details/7755369

4. http://www.xuebuyuan.com/zh-tw/2125393.html