Hadoop上Java如何通过Jni调用C++共享库

应用场景：我们有一个性能很好的分词器，用c++实现的，现在想在java写的Hadoop的程序中使用它，咋办？

如果只是使用hadoop，用c++ pipes实现hadoop程序，再调用c++实现的分词器（源代码调用或者动态库调用）就很简单，不存在上面的问题。从网上了解到，Java调用c++用的是JNI（java native interface）技术，只是JNI怎么放到hadoop中？而且分词器要读取资源文件，这个文件在hadoop中的路径设定有什么规矩？

尝试分三阶段进行：

阶段一：在linux跑通一个单机版的JNI程序，即用java调用c++。

阶段二：将上面的程序放到hadoop上跑通。

阶段三：让c++编出来的动态库（so文件,注意要在前面加lib前缀）load资源，并在hadoop上跑通。

现在进行阶段一的工作。

1. 建一个工程，名为jni，其下建立3个文件夹bin, src, lib, 在src下新建一个包loadlib，所有java源代码放在此包下，再建一个c++包用于存放所有c++代码。

写一个Java类LoadLib.java，用来包装c++代码的接口。示意性代码，如下：

```
wang@kubuntu:~/Workspace/jni$ vim src/loadlib/LoadLib.java

package loadlib;

public class LoadLib {
                public static native String SegmentALine(String line);
                static{
                        System.loadLibrary("LoadLib");
                }
```

这里面声明了静态函数接口，并用了”native“关键字，表示是native函数（非java的、本地函数）。在”static“语句块儿中，用LoadLibrary调用（即将生成的）c++动态库。

2. 用javac命令编译LoadLib类，生成.class文件，命令如下：

```
wang@kubuntu:~/Workspace/jni$ javac -d ./bin ./src/loadlib/LoadLib.java
```

3. 在LoadLib.class的基础上，用javah命令生成c++函数的头文件，命令如下：

```
wang@kubuntu:~/Workspace/jni$ javah -d src/c++/ -jni -classpath bin/
loadlib.LoadLib
```

其中,src/c++表示c++头文件输出的目录，bin/表示.class文件所在目录；后面的参数，第一个“loadlib”表示 package名称，第二个“LoadLib”表示class名称。敲完命令后，就能在src/c++目录下发现c++函数的头文件loadlib_LoadLib.h,

打开看一下，主要将java类中的static函数转成了c++接口，内容如下：

```
/* DO NOT EDIT THIS FILE - it is machine generated */
```

```
#include <jni.h>
/* Header for class loadlib_LoadLib */

#ifndef _Included_loadlib_LoadLib
#define _Included_loadlib_LoadLib
#ifdef __cplusplus
extern "C" {
#endif
/*
 * Class:     loadlib_LoadLib
 * Method:    SegmentALine
 * Signature: (Ljava/lang/String;)Ljava/lang/String;
 */
JNIEXPORT jstring JNICALL Java_loadlib_LoadLib_SegmentALine
  (JNIEnv *, jclass, jstring);

#ifdef __cplusplus
}
#endif
#endif
```

文件上第一句“DO NOT EDIT THIS FILE .....”表示这是个自动生成的文件。其时，不必用 javah 命令来生成这个文件，手写也没问题。不过毕竟自动生成方便，尤其是在接口函数比较多的情况。

4. 费了这么半天事情，就是生成了一个 c++头文件，正经事还没干呢。什么是正经事？既然非用 c++不可，正经事就是用 c++对所需功能的实现。其时用 c++ 的理由是尽量利用现有的、成熟的代码，所以这一步，一般不是功能性开发，而是写个 wrapper 包装现有的代码——如果是纯功能性开发，那直接用 java 的了，费这么多事干嘛？

在 src/c++下面新建一个 loadlib_LoadLib.cpp, 名字都可以随便取的，这是一个 c++代码完成的一个将接受到的字符串拼接后返回的函数。这只是为了测试 java 将要通过 jni 调用这个函数。

```
#include <jni.h>
#include <stdio.h>
#include <string.h>
#include "loadlib_LoadLib.h"

JNIEXPORT jstring JNICALL Java_loadlib_LoadLib_SegmentALine
(JNIEnv *env, jclass obj, jstring line)
{
      char buf[128];
```

```
    const char *str = NULL;
    str = env->GetStringUTFChars(line, false);
    if (str == NULL)
        return NULL;
    strcpy (buf, str);
    strcat (buf, "-- c++ codes \n");
    env->ReleaseStringUTFChars(line, str);
    return env->NewStringUTF(buf);
}
```

在实现中，jni.h这个头文件是必须要包含的，将来在编译的时候，也要在系统搜索路径上。“JNIEnv *env, jclass obj, jstring line"这些东西到底是什么意思，怎么用，请参考《 在 Linux 平台下使用 JNI 》http://www.linuxidc.com/Linux/2012-12/75536.htm。。这段代码功能也很简单，就是再输入字符串的基础上，加上”-- c++ codes“的字样，并且返回。

5. 在本地环境下编译出c++动态库。为啥要强调“在本地环境”？字面上的意思就是，你在windows下用JNI，就到windows下编译 loadlib_LoadLib.cpp文件，生成dll；在linux下，就到linux下（32位还是64 位自己搞清楚）编译loadlib_LoadLib.cpp文件，生成.so文件(注意加上前缀lib)，为啥非要这样？这就涉及到动态库的加载过程，每个系统都不一样。

编译命令为：

```
wang@kubuntu:~/Workspace/jni$ g++ -I/usr/lib/jvm/java-7-openjdk-amd64/include/ src/c++/loadlib_LoadLib.cpp -fPIC -shared -o lib/libLoadLib.so
```

命令有点长，不过意思很容易。“-I”表示要包含的头文件。正常来讲，系统路径都已经为g++设置好了。不过jni.h是java的头文件，不是 c++的，g++找不到，只好在编译的时候告诉编译器。我这里用的是ubuntu14.04的hadoop伪分布式，jdk是默认的所以头文件是在usr/lib/jvm/java-7-openjdk-amd64/include/里的，”-shared“表示输出的是动态库（共享库，有别于静态库的.a文件）；”-o“表示输出文件路径名。

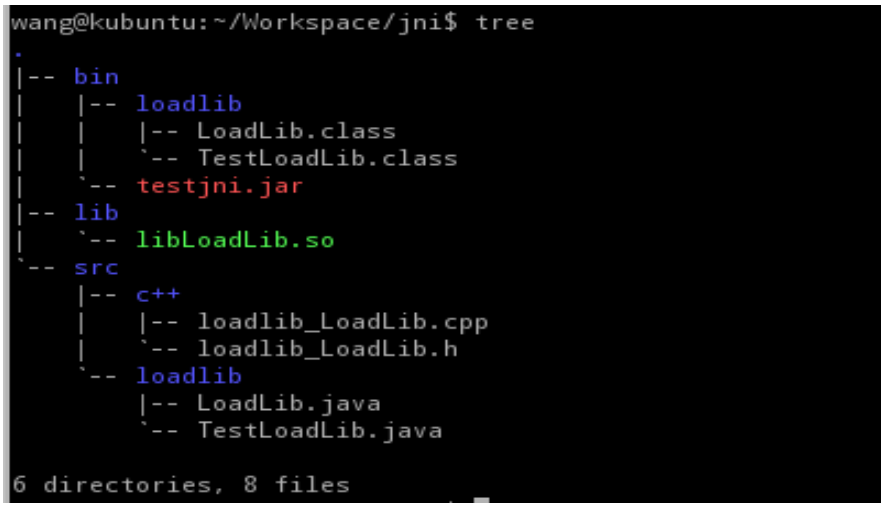
顺利的话，就能生成libLoadLib .so文件。

6. 本地java程序调用libLoadLib .so，代码很简单，在src/loadlib下建一个TestLoadLib.java如下：

```
package loadlib;


/**
 * This class is for verifying the jni technology.
 * It call the function defined in LoadLib.java
 *
 */


public class TestLoadLib {
```

```
    public static void main(String[] args) throws Exception {
        System.out.println ("In this project, we test jni to c++!\n");
        String s = LoadLib.SegmentALine("now we test LoadLib");
        System.out.print(s);
    }
}
```

测试代码也很简单，就是输入给 LoadLib.SegmentALine 一个字符串，并且打印它的返回结果。最后的文件结构图如下：



7. 将 bin/loadlib/*.class 文件打包成 testjni.jar

```
wang@kubuntu:~/Workspace/jni$ cd bin/
wang@kubuntu:~/Workspace/jni/bin$ jar -cvf testjni.jar loadlib/
已添加清单
正在添加: loadlib/(输入 = 0) (输出 = 0)(存储了 0%)
正在添加: loadlib/TestLoadLib.class(输入 = 812) (输出 = 475)(压缩了 41%)
正在添加: loadlib/LoadLib.class(输入 = 476) (输出 = 297)(压缩了 37%)
```

8. 在 linux 上执行如下命令运行上述代码：

```
wang@kubuntu:~/Workspace/jni$ java -Djava.library.path='lib/' -cp
bin/testjni.jar loadlib.TestLoadLib
In this project, we test jni to c++!


now we test LoadLib-- c++ codes
```

（用-Djava.library.path 指明 libLoadLib.so 文件所在的路径， 否则 jvm 找不到；后面 loadlib.TestLoadLib 是 main 函数所在的路径），由运行结果可以看出，java 程序已经成功地调用了 c++实现的函数并且取得了 c++中返回的值。到此第一步完成。

第二阶段：将上面的程序放到 hadoop 上跑通。

这个阶段的尝试我吃了不少苦头，主要是路径问题：hadoop 将我写好的 jar 包分发到每个 tasknode 上，同时，在各个节点上我们要把.so 也分发到相同路径下，并“告诉”tasknode，使得 jvm 在运行 jar 包的时候能够找到这个动态库。

１．写 hadoop 程序。这一阶段我将在 eclipse 下面来编写测试程序．接着阶段一的工程，我们在工程一的基础上新建一个包 hadoopjni,在 src 下新建．建好后在下面写 mapreduce 的程序，由于这个程序是将运行在 Hadoop 上的，所以首先我们需要先引入一些 Hadoop 的相关接口与依赖库．本人用的是 Hadoop-2.6.0，对于本程序，不需要将所有库都引入进来，但必须引入的库是 share/hadoop/common/*, share/hadoop/mapreduce/*, share/hadoop/hdfs/*, 注意，若在集群上运行出问题，请导入 hadoop 下的所有类库，为了方便，我将 mapper 与 reducer 的子类定义成静态类，并且外层用一个大类包裹起来。 share/hadoop/yarn/*,

mapper 类：

```
public static class MapTestJni extends Mapper<Writable, Text, Text, Text> {

        protected String s;
        protected boolean t;
        protected void setup(Context context) throws IOException, InterruptedException
        {
           s = LoadLib.SegmentALine("-value ");
        }

        protected void map(Writable key, Text value, Context context)
        throws IOException, InterruptedException {
           context.write(new Text("key"), new Text(s.toString()+t+"恭喜成功打开 lex.txt 文件，java 调用 C
++成功运行在 hadoop 上！"));
        }
    }
```

在 mapper 中，重写了 setup()和 map()函数，在 setup 函数中，调用动态库初始化了一个字符串。这个字符串 s 的值是“-value -- c++ codes”。在 map 函数中，简单的输出这个字符串。

reducer 类：

```
public static class ReduceTestJni extends Reducer<Text, Text, Text, Text> {
        protected void reduce(Text key, Iterable<Text> values, Context context)
        throws IOException, InterruptedException {
           String outString = "";
           boolean b = false;
           for (Text value: values)
           {
              outString = value.toString();

           }

           context.write(key, new Text(outString));
        }
}
```

reducer 中重写了 reduce 函数，功能是直接输出这个字符串。

控制函数：runTextJni。

```
public void runTestJni (String[] args) throws Exception {

    //  the configuration
    Configuration conf = new Configuration();
   GenericOptionsParser goparser = new GenericOptionsParser(conf, args);
    String otherargs [] = goparser.getRemainingArgs();

    // the job
    Job job;
    job = new Job(conf, "@here-TestLoadLib-hadoopJni");
    job.setJarByClass(MapReduceJni.class);

    // the mapper
    job.setMapperClass(MapTestJni.class);
    job.setMapOutputKeyClass(Text.class);
    job.setMapOutputValueClass(Text.class);

    // the reducer
    job.setReducerClass(ReduceTestJni.class);
    job.setOutputKeyClass(Text.class);
    job.setOutputValueClass(Text.class);
    job.setNumReduceTasks(1);

    // the path
    FileInputFormat.addInputPath(job, new Path(otherargs[1]));
    FileOutputFormat.setOutputPath(job, new Path(otherargs[2]));

    job.waitForCompletion(true);
}
```

其中 GenericOptionsParser 那两行是关键，用来分析和执行 hadoop 命令中传进来的特殊参数。配合命令行中的命令（下文写），他把动态库分发到 tasknode 上，路径与 jar 的执行路径相同。

通常用 jar 包的形式运行 hadoop 程序，所需的参数，如：输入路径、输出路径、mapper、combiner、reducer 等都可以用 Job 来设置，不需要额外的参数。命令行中少了这些参数，会显得短很多。尤其 hadoop 命令行一般都挺长，就很方便。相反地，采用 c++ streaming 的方式来运行程序的时候，就需要用-input、-output 等参数来指定相关参数。不过，在 jar 包中，除了用 Job 外，也可以用 GenericOptionsParser 来解析上述命令行中的参数，只要命令行配合有相应的输入，GenericOptionsParser 就可以解 析。对于-input、-output 等来讲，没有必要这样做。还有一个参数是-files，就是把-files 后面的文件（多个文件用逗号间隔）同 jar 包一起分发到 tasknode 中，这个参数，刚好可以将我们的动态库分发下去。

“goparser.getRemainingArgs();”这条语句，是在 GenericOptionsParser 解析完特殊参数之后，获得剩下的参数列表，对于我们来讲，剩下的参数就是 main 函数所在的类名、输入路径和输出路径，参见下面的命令行。

main 函数：

```
package hadoopjni;

public  class TestJni {
        public static void main(String[] args) throws Exception {

    System.out.println ("In this project, we test jni!\n");

    // test jni on linux local
    /*String s =LoadLib.SegmentALine("now we test LoadLibi");
    System.out.print(s);*/

    // test jni on hadoop
    new MapReduceJni().runTestJni(args);
  } // main
}
```

再贴上上面 map 和 reduce 的完整代码：

```
package hadoopjni;

import java.io.IOException;
import java.lang.Iterable;

import loadlib.LoadLib;

import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.io.Writable;
import org.apache.hadoop.mapreduce.Job;
import org.apache.hadoop.mapreduce.Mapper;
import org.apache.hadoop.mapreduce.Reducer;
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;
import org.apache.hadoop.util.GenericOptionsParser;

public class MapReduceJni {

public static class MapTestJni extends Mapper<Writable, Text, Text, Text> {

            protected String s;
            protected boolean t;
            protected void setup(Context context) throws IOException, InterruptedException
            {   t = LoadLib.Init("src/input/lex.txt");
                System.out.println(t);
                s = LoadLib.SegmentALine("-value ");
            }

            protected void map(Writable key, Text value, Context context)
            throws IOException, InterruptedException {
                context.write(new Text("key"), new Text(s.toString()+t+"恭喜成功打开 lex.txt 文件，java 调用 C
++成功运行在 hadoop 上！"));
            }
        }
```

```
        public static class ReduceTestJni extends Reducer<Text, Text, Text, Text> {
          protected void reduce(Text key, Iterable<Text> values, Context context)
          throws IOException, InterruptedException {
             String outString = "";
             boolean b = false;
             for (Text value: values)
             {
                outString = value.toString();

             }

             context.write(key, new Text(outString));
          }
        }


        public void runTestJni (String[] args) throws Exception {

      //  the configuration
      Configuration conf = new Configuration();
      GenericOptionsParser goparser = new GenericOptionsParser(conf, args);
      String otherargs [] = goparser.getRemainingArgs();

      // the job
      Job job;
      job = new Job(conf, "@here-TestLoadLib-hadoopJni");
      job.setJarByClass(TestJni.class);

      // the mapper
      job.setMapperClass(MapTestJni.class);
      job.setMapOutputKeyClass(Text.class);
      job.setMapOutputValueClass(Text.class);

      // the reducer
      job.setReducerClass(ReduceTestJni.class);
      job.setOutputKeyClass(Text.class);
      job.setOutputValueClass(Text.class);
      job.setNumReduceTasks(1);

      // the path
      FileInputFormat.addInputPath(job, new Path(otherargs[0]));
      FileInputFormat.addInputPath(job, new Path(otherargs[1]));
      FileOutputFormat.setOutputPath(job, new Path(otherargs[2]));

      job.waitForCompletion(true);
   }

}
```

然后将 bin/hadoopjni/下的.class 打成 jar 包，由于用的 eclipse,所以保存时已经进行编译了。提交任务之前确保先启动 Hadoop，在命令行中提交 hadoop 任务：

```
wang@kubuntu:~/Workspace/jni$ hadoop jar bin/hadoopjni.jar hadoopjni.TestJni
in/libLoadLib.so  in/file1.txt out/outputjni
```

这个也是关键。说一下几个需要注意的地方吧：

hadoop jar 命令后面跟随的第一个参数一定是打好的 jar 包，在本例中是 hadoopjni.jar 文件及其路径,由于在控制函数中用了 GenericOptionsParser，jar 包后面就必须紧跟需要设定的参数，这里，in/libLoadLib.so, in/file1.txt, out/outputjni 他们分别是输入的 c++共享库，输入文件路径、输出路径。这些目录是 HDFS 文件系统上的，都是我们已经上传上去的文件，关于 hadoop 的文件上传这里就不演示了。

第三阶段：Hadoop 上 java 调用 C++的.so，功能函数在 c++中实现，系统配置由 java 来配置。

首先，在动态库.so 的 LoadLib 增加一个 init 函数，用来 load 一个词典文件：

```
package loadlib;

public class LoadLib {
		public static native String SegmentALine(String line);
		public static native boolean Init (String file);
		static{
			System.loadLibrary("LoadLib");
		}
}
```

javac 命令生成 LoadLib .class 文件，用 javah 命令生成 c++头文件 loadlib_LoadLib.h。用 javah 的时候，要注意路径问题。loadlib_LoadLib.h 看起来是这个样子的：

```
/* DO NOT EDIT THIS FILE - it is machine generated */
#include <jni.h>
/* Header for class loadlib_LoadLib */

#ifndef _Included_loadlib_LoadLib
#define _Included_loadlib_LoadLib
#ifdef __cplusplus
extern "C" {
#endif
/*
 * Class:     loadlib_LoadLib
 * Method:    SegmentALine
 * Signature: (Ljava/lang/String;)Ljava/lang/String;
 */
JNIEXPORT jstring JNICALL Java_loadlib_LoadLib_SegmentALine
  (JNIEnv *, jclass, jstring);
```

```
JNIEXPORT jboolean JNICALL Java_loadlib_LoadLib_Init
  (JNIEnv *, jclass, jstring);

#ifdef __cplusplus
}
#endif
#endif
```

其实也可以自己来写。

第二，编写 loadlib_LoadLib.cpp 文件内容，实现相关功能，如下：

```
#include <jni.h>
#include <stdio.h>
#include <string.h>
#include<vector>
#include<fstream>
#include<iostream>

#include "loadlib_LoadLib.h"

using namespace std;
vector <string> WordVec;

JNIEXPORT jboolean JNICALL Java_loadlib_LoadLib_Init
          (JNIEnv *env, jclass obj, jstring line)
{
          const char *pFileName = NULL;
          pFileName = env->GetStringUTFChars(line,false);
          if(pFileName == NULL)
                    return false;
          ifstream in ( pFileName);
          if( !in)
          {
                    cerr << "can not open the file of " << pFileName << endl;
                    return false;
          }
          string sWord;
          while( getline(in, sWord))
          {
                    WordVec.push_back(sWord);

          }
          return true;
}


JNIEXPORT jstring JNICALL Java_loadlib_LoadLib_SegmentALine
(JNIEnv *env, jclass obj, jstring line)
{
    char buf[128];
    buf[0] = 0;
    const char *str = NULL;
    str = env->GetStringUTFChars(line, false);
    if (str == NULL)
```

```
        return NULL;
    strcpy (buf, str);
    if(!WordVec.empty()){
        strcat (buf, WordVec[0].c_str());
    }
        //strcat (buf, "c++ codes!");
    env->ReleaseStringUTFChars(line, str);
    return env->NewStringUTF(buf);
}
```

功能很简单，就是在 Init 函数中打开一个文件，将文件中的每一行存储在全局变量 WordVec 中；然后，在 SegmentALine 函数中，将输入字符串和 WordVec 中的第一个元素相连接，再输出。用 g++将.cpp 文件编译成.so 文件，命令如下：

```
$ g++ -I/usr/lib/jvm/java-7-openjdk-amd64/include/ src/c++/loadlib_LoadLib.cpp -fPIC -shared -o libLoadLib.so
```

第三步，写 hadoop 程序。

mapper 类：

```
public static class MapTestJni extends Mapper<Writable, Text, Text, Text> {

        protected String s;
        protected boolean t;
        protected void setup(Context context) throws IOException, InterruptedException
        {  t = LoadLib.Init("src/input/lex.txt");
           System.out.println(t);
           s = LoadLib.SegmentALine("-value ");
        }

        protected void map(Writable key, Text value, Context context)
        throws IOException, InterruptedException {
           context.write(new Text("key"), new Text(s.toString()+t+"恭喜成功打开 lex.txt 文件，java 调用 C++成功运行在 hadoop 上！"));
        }
    }
```

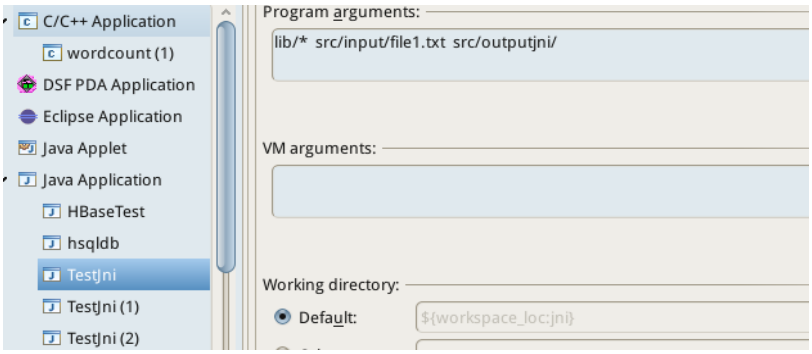
Reducer 类：

```
public static class ReduceTestJni extends Reducer<Text, Text, Text, Text> {
        protected void reduce(Text key, Iterable<Text> values, Context context)
        throws IOException, InterruptedException {
           String outString = "";
           boolean b = false;
           for (Text value: values)
           {
              outString = value.toString();

           }

           context.write(key, new Text(outString));
        }
    }
```

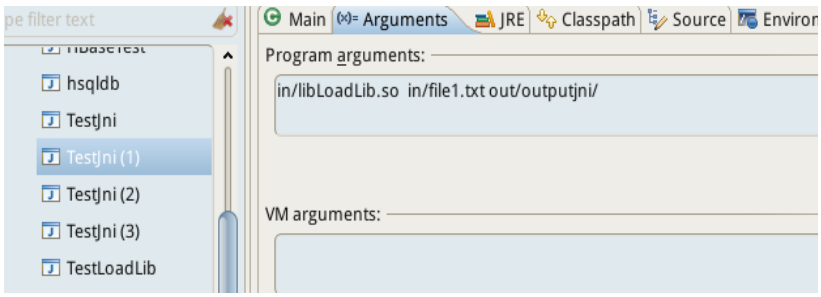
在 setup 函数中，我们调用了 LoadLib.Init 函数，来 load 文件 Lex.txt 中的内容。可以看到，相对路径就是本地当前路径。在下文分发过程中，会将 Lex.txt 文件分发到与 jar 文件相同的本地路径下。在 map 函数中，输出 s 的内容。

第四步，在 eclipse 下设置输入输出路径

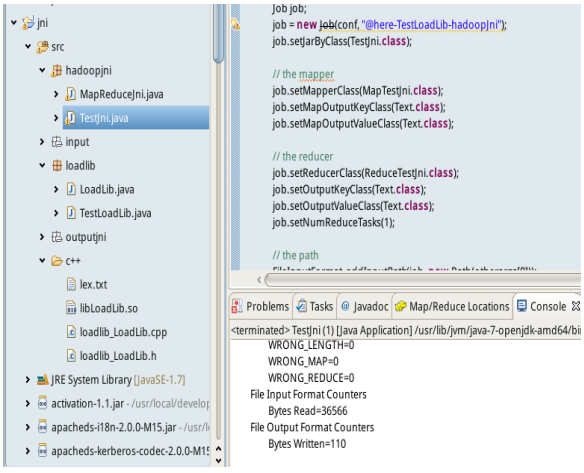
1） 本地模式输入输出文件设置：

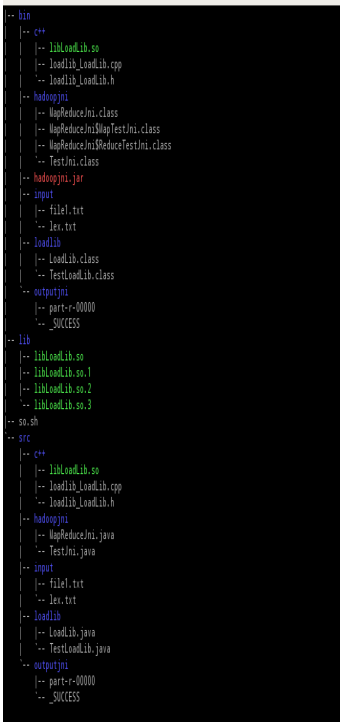


2） 集群模式输入输出路径设置：



3） 程序树形结构





4） 运行结果：

key-value 是 mapreduce 上的，wang yuan long 是 C++从 HDFS 文件系统上读取的第一行的元素，可以看到已经成功与 mapreduce 联接。true 是 c++成功打开文件是返回的值。结果已经完全没有问题。