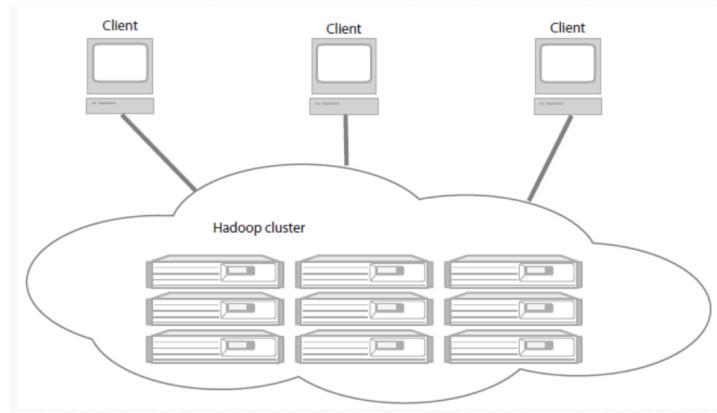


实验报告

1. 安装Hadoop

1.1. 原理简述

Hadoop是一个由Apache基金会所开发的分布式系统基础架构。用户可以在不了解分布式底层细节的情况下，开发分布式程序。充分利用集群的威力进行高速运算和存储。Hadoop的框架最核心的设计就是：HDFS和MapReduce。HDFS为海量的数据提供了存储，而MapReduce则为海量的数据提供了计算。



集群系统示意图

正如前几次实验中的，Linux虚拟机在我的Mac上运行实在太慢，并且考虑到Mac也是类Unix系统，所以我暂时没有使用虚拟机，项目都在Mac环境下运行。也因此，Mac上的安装都需要自己摸索。而这次Hadoop在Mac上的安装费尽周折，并且遇到网上没有先例的问题（由于编译版本不同）、包括Hadoop2.2.0源码中的许多bug。

在此将我的安装过程记录下来，或许对后人有用。

环境：`MacOS-10.14`，`java-1.7.0_80`，`cmake3.13.0-rc3` with `clang-1000.10.44.4`，
`Maven 3.6.0`，`protobuf 2.5.0`。

参考：[hadoop2.2.0 centos 编译安装详解](#), [hadoop2.2.0 版本编译64位native库的问题](#), [mac下hadoop环境的搭建以及碰到的坑点](#), [mac下hadoop 2.6.0编译native library](#), [重新编译hadoop 32bit-64bit](#), [Running Hadoop On OS X 10.5 64-bit\(Single-NodeCluster\)](#), [HOW TO SETUP HADOOP ON MAC OS X 10.9 MAVERICKS](#), [Undefined symbols for architecture x86_64: “fcloseall”](#)

1.2. Mac上安装Hadoop2.2.0

- **ssh配置**

单机上用Hadoop要配置伪分布式环境，则需要使用远程登录，所以先确认能够远程登录：

系统偏好设置-共享-远程登录，然后在终端运行

```
1 ssh-keygen -t rsa
2 cat ~/ssh/id_rsa.pub >> ~/ssh/authorized_keys
3 chmod og-wx ~/ssh/authorized_keys
4 ssh localhost
```

这样就创建了ssh的localhost登录方式。然而我在运行时出现了权限问题，经查资料，应该检查ssh的父目录的文件权限并执行 `chmod 775 ~/ssh`。

- 安装配置Hadoop文件

首先在Apache官网上下载 `hadoop-2.2.0.tar.gz`，解压后进入文件夹，在目录下的 `etc/hadoop` 中，进行以下文件的修改：

1. `core-site.xml`

```
1 <configuration>
2   <!-- 指定HDFS老大 (namenode) 的通信地址 -->
3   <property>
4     <name>fs.defaultFS</name>
5     <value>hdfs://localhost:9000</value>
6   </property>
7   <!-- 指定hadoop运行时产生文件的存储路径 -->
8   <property>
9     <name>hadoop.tmp.dir</name>
10    <value>/Users/markdana/hadoop-2.2.0/temp</value>
11  </property>
12 </configuration>
```

2. `hdfs-site.xml`

默认副本数3，修改为1。`dfs.namenode.name.dir` 指明fsimage存放目录，多个目录用逗号隔开。`dfs.datanode.data.dir` 指定块文件存放目录，多个目录逗号隔开。

```
1 <configuration>
2   <property>
3     <name>dfs.replication</name>
4     <value>1</value>
5   </property>
6   <property>
7     <name>dfs.namenode.name.dir</name>
8     <value>file:/Users/markdana/hadoop-2.2.0/temp/hdfs/name</value>
9   </property>
10  <property>
11    <name>dfs.datanode.data.dir</name>
12    <value>file:/Users/markdana/hadoop-2.2.0/temp/hdfs/data</value>
13  </property>
14  <property>
15    <name>dfs.namenode.secondary.http-address</name>
16    <value>localhost:9001</value>
17  </property>
18  <property>
19    <name>dfs.webhdfs.enabled</name>
20    <value>true</value>
21  </property>
22 </configuration>
```

3.配置yarn,

`mapred-site.xml` :

```
1 <configuration>
2   <property>
3     <name>mapreduce.framework.name</name>
4     <value>yarn</value>
5   </property>
6   <property>
7     <name>mapreduce.admin.user.env</name>
8     <value>HADOOP_MAPRED_HOME=$HADOOP_COMMON_HOME</value>
9   </property>
10  <property>
11    <name>yarn.app.mapreduce.am.env</name>
12    <value>HADOOP_MAPRED_HOME=$HADOOP_COMMON_HOME</value>
13  </property>
14
15  <property>
16    <name>mapreduce.application.classpath</name>
17    <value>
18      /Users/markdana/hadoop-2.2.0/etc/hadoop,
19      /Users/markdana/hadoop-2.2.0/share/hadoop/common/*,
20      /Users/markdana/hadoop-2.2.0/share/hadoop/common/lib/*,
21      /Users/markdana/hadoop-2.2.0/share/hadoop/hdfs/*,
22      /Users/markdana/hadoop-2.2.0/share/hadoop/hdfs/lib/*,
23      /Users/markdana/hadoop-2.2.0/share/hadoop/mapreduce/*,
24      /Users/markdana/hadoop-2.2.0/share/hadoop/mapreduce/lib/*,
25      /Users/markdana/hadoop-2.2.0/share/hadoop/yarn/*,
26      /Users/markdana/hadoop-2.2.0/share/hadoop/yarn/lib/*
27    </value>
28  </property>
29 </configuration>
```

yarn-site.xml :

```
1 <configuration>
2   <!-- Site specific YARN configuration properties -->
3     <property>
4       <name>yarn.nodemanager.aux-services</name>
5       <value>mapreduce_shuffle</value>
6     </property>
7     <property>
8       <name>yarn.application.classpath</name>
9       <value>
10      /Users/markdana/hadoop-2.2.0/etc/hadoop,
11      /Users/markdana/hadoop-2.2.0/share/hadoop/common/*,
12      /Users/markdana/hadoop-2.2.0/share/hadoop/common/lib/*,
13      /Users/markdana/hadoop-2.2.0/share/hadoop/hdfs/*,
14      /Users/markdana/hadoop-2.2.0/share/hadoop/hdfs/lib/*,
15      /Users/markdana/hadoop-2.2.0/share/hadoop/mapreduce/*,
16      /Users/markdana/hadoop-2.2.0/share/hadoop/mapreduce/lib/*,
17      /Users/markdana/hadoop-2.2.0/share/hadoop/yarn/*,
18      /Users/markdana/hadoop-2.2.0/share/hadoop/yarn/lib/*
19    </value>
20  </property>
21</configuration>
```

• 配置Hadoop环境变量

`vim ~/.bash_profile` 在环境变量中添加：

```
1 export HADOOP_HOME=/Users/markdana/hadoop-2.2.0
2 export PATH=$PATH:$HADOOP_HOME/sbin:$HADOOP_HOME/bin
3
4 export LD_LIBRARY_PATH=$HADOOP_HOME/lib/native/
5 export HADOOP_COMMON_LIB_NATIVE_DIR=$HADOOP_HOME/lib/native
6 export HADOOP_OPTS="-Djava.library.path=$HADOOP_HOME/lib/native:$HADOOP_COMMON_LIB_NATIVE_D
```

按照网络教程的说法，到这一步配置基本完成，没什么问题了，应该可以直接开始运行Hadoop了。这时我也可以正常打开hdfs管理界面和hadoop进程管理界面等：

NameNode 'localhost:9000' (active)			
Started:	Mon Nov 12 22:59:24 CST 2018		
Version:	2.2.0, 1529768		
Compiled:	2013-10-07T06:28Z by hortonmu from branch-2.2.0		
Cluster ID:	CID-a5a3fc0d-cff8-4972-af12-4db3aa1cc782		
Block Pool ID:	BP-1144016192-192.168.1.108-1541816977679		
Browse the filesystem			
NameNode Logs			
Cluster Summary			
Security is OFF			
4 files and directories, 0 blocks = 4 total.			
Heap Memory used 50.65 MB is 35% of Committed Heap Memory 142 MB. Max Heap Memory is 889 MB.			
Non Heap Memory used 32.66 MB is 98% of Committed Non Heap Memory 33.22 MB. Max Non Heap Memory is -1 B.			
Configured Capacity	:	233.47 GB	
DFS Used	:	4 KB	
Non DFS Used	:	190.97 GB	
DFS Remaining	:	42.50 GB	
DFS Used%	:	0.00%	
DFS Remaining%	:	18.20%	
Block Pool Used	:	4 KB	
Block Pool Used%	:	0.00%	
DataNodes usages	:	Min % Median % Max % stdev %	
		0.00% 0.00% 0.00% 0.00%	
Live Nodes	:	1 (Decommissioned: 0)	
Dead Nodes	:	0 (Decommissioned: 0)	
Decommissioning Nodes	:	0	
Number of Under-Replicated Blocks	:	0	

NameNode Journal Status:

Current transaction ID: 20

在loclahost查看hdfs

于是按照PPT上的教程，创建目录、写新文件、统计词频等操作，然而却报

错 `Unable to load native-hadoop library for your platform`。经过查资料得知，运行hadoop需要用到 `native library` 动态库，而在官网上明确说明：

The pre-built 32-bit i386-Linux native hadoop library is available as part of the hadoop distribution and is located in the lib/native directory. You can download the hadoop distribution from Hadoop Common Releases.

The native hadoop library is supported on *nix platforms only. The library **does not** work with Cygwin or the **Mac OS X** platform.

在官网下载的Hadoop版本中自带的 `native library` 只针对32位Linux。不适用于**Mac OS X**。因此我必须要在Mac下重新编译 `native library`。现在我们将迎来整个安装过程中最坑爹的部分。

1.3. Mac上编译 `native library`

- 准备编译环境

0. 下载hadoop2.2.0的二进制源码，解压成文件夹 `hadoop-2.2.0-src`

1. 安装cmake

没有版本要求

2. 安装protoc

版本必须是2.5.0，否则编译失败。网上说不能用brew install protobuf的方式安装，因为该方式安装的版本不一定是2.5.0，需要自己下源码编译。

我的做法是先 `brew search protobuf`，找到需要的版本后，`brew install protobuf@2.5`，再配置环境变量：

```
1 | export PROTOBUF=/usr/local/Cellar/protobuf@2.5/2.5.0
2 | export PATH=$PROTOBUF/bin:$PATH
```

为了编译器能找到protobuf还需要设置：

```
1 | export LDFLAGS="-L/usr/local/opt/protobuf@2.5/lib"
2 | export CPPFLAGS="-I/usr/local/opt/protobuf@2.5/include"
```

证明也是可行的。

3. 安装maven

我使用的Apache Maven 3.6.0，设置环境变量

```
1 | export M2_HOME=/Users/markdara/apache-maven-3.6.0
2 | exxport PATH=$M2_HOME/bin:$PATH
```

4. 创建Java软链接

编译过程中报错 `Missing tools.jar`，是由于Linux系统中默认储存 `tools.jar` 的路径和Mac中不同。根据Building.txt要求，设置 `JAVA_HOME` 是 `....jdk/Contents/Home` 的Java路径，在其中创立 `Classes` 文件夹，并执行

```
1 | sudo ln -s $JAVA_HOME/lib/tools.jar $JAVA_HOME/Classes/classes.jar`
```

创建软链接就不会找不到文件了。另外，关于 `JAVA_HOME` 的设置，后续还会提起。

• 开始编译

进入源码根目录中，执行

```
1 | mvn package -Pdist,native -DskipTests -Dtar
```

期望的结果是经过漫长编译后，全部BUILD SUCCESS，然后将编译出的 `native library` 库（路径 `hadoop-2.2.0-src/hadoop-dist/target/hadoop-2.2.0/lib/native`）替换到之前安装的hadoop的库目录中（路径 `hadoop-2.2.0/lib/native`）。然而，一共54个编译项目，出现的WARNING可以不管，但任何一个有ERROR都会在该处卡住，需要debug后才能往下跑通。

• Apache Hadoop Annotations编译失败

报错是关于Java的，比如：

```
1 | ..tools/RootDocProcessor.java:[20,22] 错误：程序包com.sun.javadoc不存在
```

```
1 | ..tools/RootDocProcessor.java:[55,16] 错误：找不到符号
```

在之前已经设置了Java的软链接，那为什么找不到程序包呢？难道是本机安装的脚步问题？于是进入 `hadoop-2.2.0-src/hadoop-common-project/hadoop-annotations` 中查看POM文件，发现其指定的JDK是1.7，而我使用的是1.8，需要手动将1.7改成1.8，如下图：

```
51 <groupId>jdk.tools</groupId>
52 <artifactId>jdk.tools</artifactId>
53 <version>1.7</version>
54 <scope>system</scope>
55 <systemPath>${java.home}/../lib/tools.jar</systemPath>
56 </dependency>
57 </dependencies>
58 </profile>
59 <profile>
60 <id>jdk1.7</id>
61 <activation>
62 <jdk>1.7</jdk>
63 </activation>
64 <dependencies>
65 <dependency>
66 <groupId>jdk.tools</groupId>
67 <artifactId>jdk.tools</artifactId>
68 <version>1.7</version>
69 <scope>system</scope>
70 <systemPath>${java.home}/../lib/tools.jar</systemPath>
71 </dependency>
```

重新编译，问题解决。

也就是说，Hadoop源码的pom文件中，对Java的版本进行了固化，而不是动态根据识别当前环境的Java版本。想到此，考虑到是否还有其他pom文件里面对也有这种Java版本固化声明呢？

对 `requireJavaVersion` 语句进行查询。运行：

```
1 | find . -name pom.xml | xargs grep requireJavaVersion
```

结果发现以下pom文件中都需要修改jdk至1.8：

```
1 | ./hadoop-assemblies/pom.xml:
2 | ./hadoop-assemblies/pom.xml:
3 | ./hadoop-project/pom.xml:
4 | ./hadoop-project/pom.xml:
5 | ./pom.xml:
6 | ./pom.xml:
```

- Apache Hadoop Maven Plugins编译失败

仍然是关于Java运行的错误：

```
1 [ERROR] Failed to execute goal org.apache.maven.plugins:maven-javadoc-plugin:2.8.1:jar (modu
2 [ERROR] Exit code: 1 - /Users/markdana/Downloads/hadoop-2.2.0-src/hadoop-maven-plugins/src/m
3 [ERROR]   * @param command List<String> containing command and all arguments
4 [ERROR]           ^
5 [ERROR] /Users/markdana/Downloads/hadoop-2.2.0-src/hadoop-maven-plugins/src/main/java/org/ap
6 [ERROR]   * @param output List<String> in/out parameter to receive command output
7 [ERROR]           ^
8 [ERROR] /Users/markdana/Downloads/hadoop-2.2.0-src/hadoop-maven-plugins/src/main/java/org/ap
9 [ERROR]   * @return List<File> containing every element of the FileSet as a File
10 [ERROR]           ^
11 [ERROR]
12 [ERROR] Command line was: /Library/Java/JavaVirtualMachines/jdk1.8.0_131.jdk/Contents/Home/b
13 [ERROR]
14 [ERROR] Refer to the generated Javadoc files in '/Users/markdana/Downloads/hadoop-2.2.0-src/
```

也就是处理 `String`, `File` 等对象时出现bug。查资料后发现，这是由于Java1.7和Java1.8中部分语法做了更新。为了使这里运行通过，我只能使用Java1.7也难怪会在POM中进行Java的版本固化。

于是将上一步做的调整全部调回来，并且下载Java1.7的jdk安装到本地（别忘了创建软链接）。那么以后如何在不同版本的Java之间切换呢？在环境变量中添加：

```
1 export JAVA_6_HOME=/Library/Java/JavaVirtualMachines/1.6.0.jdk/Contents/Home
2 export JAVA_7_HOME=/Library/Java/JavaVirtualMachines/jdk1.7.0_80.jdk/Contents/Home
3 export JAVA_8_HOME=/Library/Java/JavaVirtualMachines/jdk1.8.0_131.jdk/Contents/Home
4 alias jdk1.6='export JAVA_HOME=$JAVA_6_HOME'
5 alias jdk1.7='export JAVA_HOME=$JAVA_7_HOME'
6 alias jdk1.8='export JAVA_HOME=$JAVA_8_HOME'
7
8 #use java7 default
9 export JAVA_HOME=$JAVA_7_HOME
10 export CLASSPATH=.:$JAVA_HOME/lib/dt.jar:$JAVA_HOME/lib/tools.jar
11 export PATH=$JAVA_HOME/bin:$PATH:
```

这样就可以通过诸如命令 `jdk1.8` 使用Java1.8了。而在接下来的实验中，都使用的Java1.7。本条编译通过。

• Apache Hadoop Common编译失败

首先是发现maven编译时下载包比较慢。为了提速，添加国内的阿里云镜像源，感谢阿里云。即在maven根目录下的 `conf` 文件夹中的 `setting.xml` 文件添加：

```
1 <mirrors>
2 <mirror>
3   <id>alimaven</id>
4   <name>aliyun maven</name>
5   <url>http://maven.aliyun.com/nexus/content/groups/public/</url>
6   <mirrorOf>central</mirrorOf>
7 </mirror>
8 </mirrors>
```

然后运行，报错如下。

```
1 The C compiler identification is unknown
2 The CXX compiler identification is unknown
3 Check for working C compiler: /usr/bin/cc
4 CMake Error at /Applications/CMake.app/Contents/share/cmake-3.13/Modules/CMakeTestCCompiler.cmake:6: 
5 The C compiler
6
7 "/usr/bin/cc" is not able to compile a simple test program.
8 It fails with the following output:
9 Change Dir: /Users/markdana/Downloads/hadoop-2.2.0-src/hadoop-common-project/hadoop-common/t
10 Run Build Command:"/usr/bin/make" "cmTC_88f64/fast"
11 xcrun: error: invalid active developer path (/Library/Developer/CommandLineTools), missing x
```

经分析是cmake的编译器路径出现错误。更新了Mojave系统后我刚更新了Xcode，可能相关组件出现问题。查看路径 `/Library/Developer/CommandLineTools`，果然缺失了 `command line tools`。在终端运行

```
1 | xcode-select --install
```

一路按照步骤安装即可。另外还有一个问题，这里报错 `missing xcrun`，但是安装之后我在 `CommandLineTools/usr/bin` 中寻找 `xcrun`，仍然找不到。卸载，换方法在[Apple开发者网站](#)下载安装，结果也没有 `xcrun`。运行仍报错

```
1 | Run Build Command:"/usr/bin/make" "cmTC_670ed/fast"
2 | xcrun: error: invalid active developer path (/Library/Developer/CommandLineTools), missing x
```

`CommandLineTools/usr/bin/xcrun` 仍然无文件，但是根据报错提示和错误log日志，运行了 `/usr/bin/make` 这个文件，于是进入目录查找，发现 `/usr/bin/` 中有 `xcrun` 文件，于是直接软链接过去，可以跑通了。猜测是由于该Build Command中也出现了与上文类似的路径错位的问题。

接下来重新编译，`C compiler`，`CXX compiler` 等都check到了。但是有新的报错：

```
1 Configuring incomplete, errors occurred!
2 See also "/Users/markdara/Downloads/hadoop-2.2.0-src/hadoop-common-project/hadoop-co
3 CMake Error at /Applications/CMake.app/Contents/share/cmake-3.13/Modules/FindPackageHandleSt
4 Could NOT find ZLIB (missing: ZLIB_INCLUDE_DIR)
5 Call Stack (most recent call first):
6 /Applications/CMake.app/Contents/share/cmakemmon/target/native/CMakeFiles/CMakeOutput.log".
7 See also "/Users/markdara/Downloads/hadoop-2.2.0-src/hadoop-common-project/hadoop-3.13/Modul
8 /Applications/CMake.app/Contents/share/cmake-3p-common/target/native/CMakeFiles/CMakeError.l
9 .13/Modules/FindZLIB.cmake:114 (FIND_PACKAGE_HANDLE_STANDARD_ARGS)
10 CMakeLists.txt:89 (find_package)
```

于是初步了解cmake的运行方式。cmake的关键是编写CMakeLists.txt文件，然后用cmake命令将CMakeLists.txt文件转化为make所需要的makefile文件，最后用make命令编译源码生成可执行程序或共享库(shared object)。CMakeLists.txt可以理解为交响乐团中的乐谱，指向各个编译的路径。要测试编译某个文件，就直接对该文件中

```
1 cmake '...src路径...'
2 make
```

那么根据报错信息，查看当前项目的

CMakeLists.txt (hadoop-2.2.0-src/hadoop-common-project/hadoop-common/src/CMakeLists.txt) 的第89行：

```
1 | find_package(ZLIB REQUIRED)
```

cmake中的 `find_package` 函数在Unix系统尤为重要。`find_package` 可以根据cmake内置的.cmake的脚本去找相应的库的模块。调用了 `find_package` 之后，相应的变量就能生成。其中有一些是内置库，比如调用了 `find_package(Qt5Widgets)`，`find_package(Qt4 COMPONENTS QCORE QTGUI QTOPENGL QTSVG)`，就会有变量 `Qt5Widgets_FOUND`，`Qt5Widgets_INCLUDE_DIRS` 生效，可以在CMakeLists.txt里使用了。

而另外一些库是没有内置的，需要自己安装。比如这里报出缺少的 `ZLIB` 库。于是通过 `brew` 安装 `ZLIB`，同时安装了 `OPENSSL` 并在环境变量中设置

```
1 export ZLIB_LIBRARY_PATH=/usr/local/Cellar/zlib/1.2.11/lib:$ZLIB_LIBRARY_PATH
2 export ZLIB_INCLUDE_DIR=/usr/local/Cellar/zlib/1.2.11/include:$ZLIB_INCLUDE_DIR
3 export CPLUS_INCLUDE_PATH=/usr/local/Cellar/zlib/1.2.11/include:$CPLUS_INCLUDE_PATH
4
5 export OPENSSL_ROOT_DIR=/usr/local/Cellar/openssl/1.0.2m
6 export OPENSSL_INCLUDE_DIR=/usr/local/Cellar/openssl/1.0.2m/include
```

重新运行，却仍然报错 `Could NOT find ZLIB (missing: ZLIB_INCLUDE_DIR)`，但环境变量中确实已经设置ZLIB的路径，这是为何呢？

对 `find_package` 函数深入探究后发现，由于使用时加入了参数 `REQUIRED`，即一定需要该库，那么cmake的原生库里会有相应的查找该库的脚本。查找后果然如此，路

径 `/Applications/CMake.app/Contents/share/cmake-3.13/Modules/FindZLIB.cmake`，进入该文件后，发现其中有一个变量 `# ZLIB_FOUND - True if zlib found.` 那么，报错找不到路径，是由于该变量由于某种原因没有被设置为True。本来以为手动设置成True即可，却仍然报错：

```
1 | An argument named "True" appears in a conditional statement.  
2 | Policy CMP0012 is not set.
```

由于某种协议，在cmake文件中直接设置True是不合法的。于是理清该程序寻找ZLIB包的逻辑，其中有提及

A user may set `ZLIB_ROOT` to a zlib installation root to tell this module where to look.

而如果没有设置 `ZLIB_ROOT`，该程序会启用其定义的 `Normal search`，在

```
1 | PATHS  
2 | "[HKEY_LOCAL_MACHINE\\SOFTWARE\\GnuWin32\\Zlib;InstallPath]"  
3 | "$ENV{ProgramFiles}/zlib"  
4 | "$ENV{ProgramFiles${_ZLIB_x86}}/zlib")`
```

中进行搜索，显然是找不到的，因此路径错误，更读不出后续的环境变量。因此设置：

```

1 # Search ZLIB_ROOT first if it is set.
2 set (ZLIB_ROOT /usr/local/Cellar/zlib/1.2.11/)
3 if(ZLIB_ROOT)
4     set(_ZLIB_SEARCH_ROOT PATHS ${ZLIB_ROOT} NO_DEFAULT_PATH)
5     list(APPEND _ZLIB_SEARCHES _ZLIB_SEARCH_ROOT)
6 endif()
7
8 .....
9
10 if(ZLIB_FOUND)
11     set(ZLIB_INCLUDE_DIRS ${ZLIB_INCLUDE_DIR})
12
13     if(NOT ZLIB_LIBRARIES)
14         set(ZLIB_LIBRARIES ${ZLIB_LIBRARY})
15     endif()
16
17     if(NOT TARGET ZLIB::ZLIB)
18         add_library(ZLIB::ZLIB UNKNOWN IMPORTED)
19         set_target_properties(ZLIB::ZLIB PROPERTIES
20             INTERFACE_INCLUDE_DIRECTORIES "${ZLIB_INCLUDE_DIRS}")
21
22     if(ZLIB_LIBRARY_RELEASE)
23         set_property(TARGET ZLIB::ZLIB APPEND PROPERTY
24             IMPORTED_CONFIGURATIONS RELEASE)
25         set_target_properties(ZLIB::ZLIB PROPERTIES
26             IMPORTED_LOCATION_RELEASE "${ZLIB_LIBRARY_RELEASE}")
27     endif()
28
29     if(ZLIB_LIBRARY_DEBUG)
30         set_property(TARGET ZLIB::ZLIB APPEND PROPERTY
31             IMPORTED_CONFIGURATIONS DEBUG)
32         set_target_properties(ZLIB::ZLIB PROPERTIES
33             IMPORTED_LOCATION_DEBUG "${ZLIB_LIBRARY_DEBUG}")
34     endif()
35
36     if(NOT ZLIB_LIBRARY_RELEASE AND NOT ZLIB_LIBRARY_DEBUG)
37         set_property(TARGET ZLIB::ZLIB APPEND PROPERTY
38             IMPORTED_LOCATION "${ZLIB_LIBRARY}")
39     endif()
40     endif()
41 endif()

```

然后再运行，关于ZLIB路径缺失的错误就没有了。

但是又有新的错误：

```
1 ...src/main/native/src/org/apache/hadoop/security/JniBasedUnixGroupsNetgroupMapping.c:77:26:
2 invalid operands to binary expression ('void' and 'int')
3 if(setnetgrent(cgroup) == 1) {
4     ~~~~~^ ~
5 1 error generated.
```

进入 `JniBasedUnixGroupsNetgroupMapping.c` 文件的76行，`if(setnetgrent(cgroup) == 1)` 是一个返回VOID值的函数，然而文件中却与INT比较。于是直接将该句改成 `setnetgrent(cgroup);`，并将判断条件改成 `if(1)`，编译通过。

本项目至此才全部通过。

- Apache Hadoop HDFS编译失败

类似于上一个的源码bug：

```
1 ...hadoop-hdfs-project/hadoop-hdfs/target/native/javah  
2 -I .../hadoop-hdfs/src/main/native/util/posix_util.c:115:12: error: use of undeclared identi  
3 char tmp[PATH_MAX];
```

即开一个字符串数组 `tmp` 储存路径，大小最大为 `PATH_MAX`，但 `PATH_MAX` 之前没有被定义。于是在开头加入一句

```
1 | int PATH_MAX = 1000;
```

编译通过。（很好奇这种低级bug是由于系统不同导致的吗？那开发者是怎样编译通过的呢....）

- Apache hadoop-yarn-server-nodemanager编译失败

报错：

```
1 use of undeclared identifier 'LOGIN_NAME_MAX'
2 if (strncmp(*users, user, LOGIN_NAME_MAX) == 0) {
3     ^
4 ...
5
6 error: too many arguments to function call, expected 4, have 5
7 if (mount("none", mount_path, "cgroup", 0, controller) == 0) {
8     ~~~~~~ ^~~~~~
```

第一个报错容易解决，类似于上文设置1000即可。

第一个错误呢？

报错说 `mount` 函数应该传4个参，这里却传了5个。上百度搜 `mount` 函数，是一个Linux系统用来挂载文件的操作，定义

```
1 #include <sys/mount.h>
2 int mount(const char *source, const char *target,
3           const char *filesystemtype, unsigned long mountflags, const void *data);
```

其定义就是传5个参，没有问题啊？联想到为何没有给出Mac的版本只有Linux的，会不会也是两个系统中的 `mount` 函数使用方法不同呢？查找错误日志，果然发现有提醒：

```
1 /Library/Developer/CommandLineTools/SDKs/MacOSX10.14.sdk/usr/include/sys/mount.h:399:1:
2 note: 'mount' declared here
```

于是进入到系统定义 `mount` 函数的头文件，在这里确实是传4个参，于是手动修改为传5个参：

```
1 //int mount(const char *, const char *, int, void *);
2 //modified for hadoop, nov/11/18
3 int mount(const char *, const char *, const char *, int, const char *);
```

修改需要密码。当然这样修改系统的命令行工具无疑是危险的。但情非得已，只能改完后编译过这个bug再立即改回来。

改完后，这前两个报错都好了。但是还有最后一个，也是耗费了我最长时间的一个debug：

```
1 [ 71%] Linking C executable target/usr/local/bin/test-container-executor
2 /Applications/CMake.app/Contents/bin/cmake -E cmake_link_script CMakeFiles/test-container-e
3 /Library/Developer/CommandLineTools/usr/bin/cc -g -Wall -O2 -D_GNU_SOURCE
4 -D_REENTRANT -isysroot /Library/Developer/CommandLineTools/SDKs/MacOSX10.14.sdk -Wl,-search_
5 -Wl,-headerpad_max_install_names CMakeFiles/test-container-executor.dir/main/native/contain
6 -o target/usr/local/bin/test-container-executor libcontainer.a
7 Undefined symbols for architecture x86_64:
8   "_fcloseall", referenced from:
9     _launch_container_as_user in libcontainer.a(container-executor.c.o)
10    ld: symbol(s) not found for architecture x86_64
11    clang: error: linker command failed with exit code 1 (use -v to see invocation)
12 make[2]: *** [target/usr/local/bin/test-container-executor] Error 1
```

刚开始看到 `clang: error` 我觉得是编译器的问题，于是下载了gcc，将cmake的编译器换成gcc，即在环境变量中设置

```

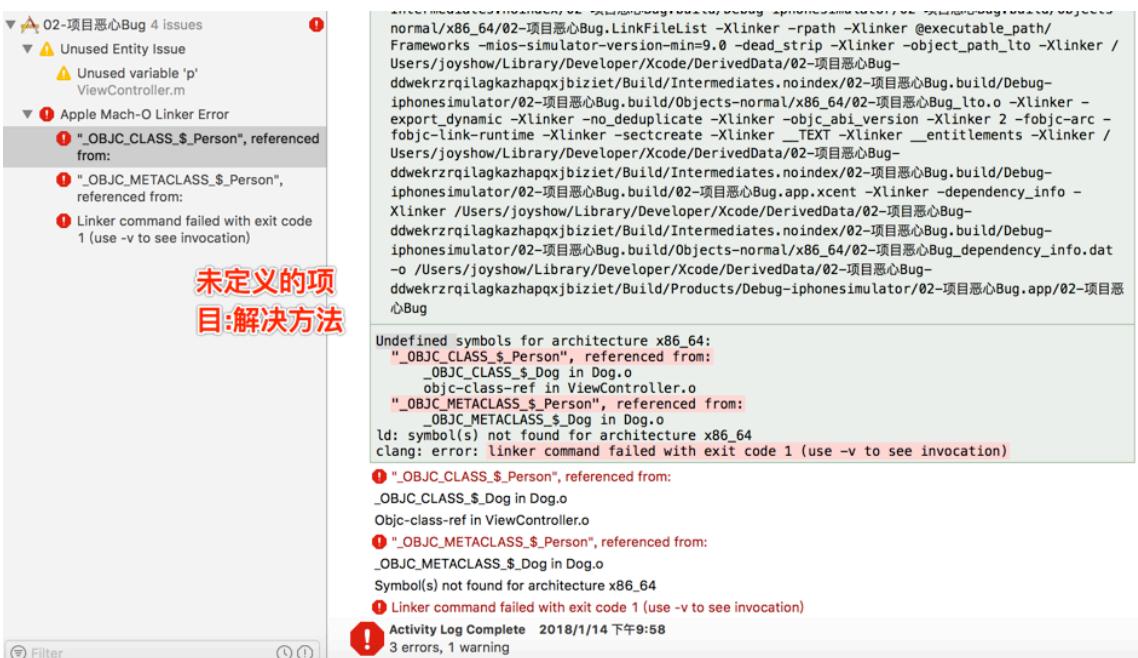
1 alias gcc=/Library/Developer/CommandLineTools/usr/bin/clang
2 alias cc=/Library/Developer/CommandLineTools/usr/bin/clang
3 alias g++=/Library/Developer/CommandLineTools/usr/bin/clang++
4 alias c++=/Library/Developer/CommandLineTools/usr/bin/clang++

5
6 alias gcc=/usr/local/Cellar/gcc/8.2.0/bin/gcc-8
7 alias cc=/usr/local/Cellar/gcc/8.2.0/bin/gcc-8
8 alias g++=/usr/local/Cellar/gcc/8.2.0/bin/g++-8
9 alias c++=/usr/local/Cellar/gcc/8.2.0/bin/g++-8

```

编译，却是同样的 `gcc: error`，看来不是编译器的问题。

然后分析 `Undefined symbols for architecture x86_64`。



这是一个在iOS开发中经常出现的bug，需要在Xcode的buildphases中手动添加类文件，否则找不到引用对象。但是这不是我开发的，我也不知道需要加什么样的类文件。

再分析 `Linking C executable target`，是在链接外部库的可执行文件过程中出现的bug。于是找遍了错误日志中提到的外部库的可执行文件，查看是否有路径缺失问题，的确有，但完善路径后也不行；在各个可执行文件中查找缺失的 `_fcloseall`，也都找不到。难道编译只能就此终结了吗？无奈之下我还加了付费的Hadoop交流群，但也没有得到答案…

最后选择在[StackOverflow](#)上询问该问题。得到一位大神的解答：

It seems that function `fcloseall` doesn't exist on OS X. From the [Porting UNIX/Linux Applications to OS X](#):

```
fcloseall
```

This function is an extension to `fclose`. Although OS X supports `fclose`, `fcloseall` is not supported. You can use `fclose` to implement `fcloseall` by storing the file pointers in an array

and iterating through the array.

You need to redesign the application and store every file which is supposed to be closed with `fcloseall`. After that, you may use simple `fclose` for every such file, as noted in the citation.

`fcloseall` 函数是关闭除标准流（`stdin`、`stdout`、`stderr`、`stdprn`、`stdaux`）之外的所有打开的流，刷新所有的流缓冲区，并返回关闭的流数。但是在Mac系统的C编译器中，只有 `fclose` 函数，并没有 `fcloseall` 函数。于是进入出错的 `container-executor.c` 文件中，查看每个打开文件的操作是否关闭了文件。

```
File container-executor.c: Line 699: fprintf(LOGFILE, "Can't open %s for output -%s\n", out_filename,
File container-executor.c: Line 707: // open up the credentials file
File container-executor.c: Line 808: int cred_file = open_file_as_nm(mPrivate_credentials_file);
File container-executor.c: Line 856: // open launch script
File container-executor.c: Line 857: int container_file_source = open_file_as_nm(script_name);
File container-executor.c: Line 858: if (container_file_source < 0) {
File container-executor.c: Line 902:     cred_file_source = open_file_as_nm(cred_file);
File container-executor.c: Line 1084: FTSS* tree = fts_open(path, FTS_PHYSICAL | FTS_XDEV, NULL);
File container-executor.c: Line 1090:     "Cannot open file traversal structure for the path %s:%s.\n",
File container-executor.c: Line 1221: dp = opendir(dir_path);
```

```
File container-executor.c: Line 724: if (close(out_fd) != 0) {
File container-executor.c: Line 725:     fprintf(LOGFILE, "Failed to close file %s - %s\n", out_filename,
File container-executor.c: Line 729:     close(input);
File container-executor.c: Line 857: fclose(stdin);
File container-executor.c: Line 860: fclose(stdout);
File container-executor.c: Line 861: fclose(stderr);
File container-executor.c: Line 976: fcloseall();
File container-executor.c: Line 1158: ret = fts_close(tree);
File container-executor.c: Line 1160:     fprintf(LOGFILE, "Error in fts_close while deleting %s\n", full_path);
File container-executor.c: Line 1228: closedir(dp);
```

逐个关闭打开的文件

如果直接删去 `fcloseall`，可能文件没有关闭，要对文件进行的改动仍在缓冲区，没有写入文件，因此需要记录下每个打开的文件并在使用完后一个个关闭。另外，文件中有Linux+C的操作文件流的指令如 `open`，`close` 等，同样要注意检查配对。

• 编译大功告成

在de完上面的bug后，再运行，终于可以一边通过了。历时两天，运行了十万行代码。就为了等到下面这个全绿通过：

```

[INFO] --- maven-javadoc-plugin:2.8.1:jar (module-javadocs) @ hadoop-minicluster ---
[INFO] Building jar: /Users/markdara/Downloads/hadoop-2.2.0-src/hadoop-minicluster/target/ha
[INFO] -----
[INFO] Reactor Summary for Apache Hadoop Main 2.2.0:
[INFO]
[INFO] Apache Hadoop Main ..... SUCCESS [ 1.797 s]
[INFO] Apache Hadoop Project POM ..... SUCCESS [ 1.293 s]
[INFO] Apache Hadoop Annotations ..... SUCCESS [ 2.618 s]
[INFO] Apache Hadoop Assemblies ..... SUCCESS [ 0.308 s]
[INFO] Apache Hadoop Project Dist POM ..... SUCCESS [ 1.971 s]
[INFO] Apache Hadoop Maven Plugins ..... SUCCESS [ 3.161 s]
[INFO] Apache Hadoop Auth ..... SUCCESS [ 2.764 s]
[INFO] Apache Hadoop Auth Examples ..... SUCCESS [ 1.973 s]
[INFO] Apache Hadoop Common ..... SUCCESS [ 01:09 min]
[INFO] Apache Hadoop NFS ..... SUCCESS [ 5.486 s]
[INFO] Apache Hadoop Common Project ..... SUCCESS [ 0.055 s]
[INFO] Apache Hadoop HDFS ..... SUCCESS [ 01:04 min]
[INFO] Apache Hadoop HttpFS ..... SUCCESS [ 12.543 s]
[INFO] Apache Hadoop HDFS BookKeeper Journal ..... SUCCESS [ 6.233 s]
[INFO] Apache Hadoop HDFS-NFS ..... SUCCESS [ 2.879 s]
[INFO] Apache Hadoop HDFS Project ..... SUCCESS [ 0.036 s]
[INFO] hadoop-yarn ..... SUCCESS [ 0.145 s]
[INFO] hadoop-yarn-api ..... SUCCESS [ 31.446 s]
[INFO] hadoop-yarn-common ..... SUCCESS [ 20.435 s]
[INFO] hadoop-yarn-server ..... SUCCESS [ 0.081 s]
[INFO] hadoop-yarn-server-common ..... SUCCESS [ 6.984 s]
[INFO] hadoop-yarn-server-nodemanager ..... SUCCESS [ 14.270 s]
[INFO] hadoop-yarn-server-web-proxy ..... SUCCESS [ 2.957 s]
[INFO] hadoop-yarn-server-resourcemanager ..... SUCCESS [ 12.260 s]
[INFO] hadoop-yarn-server-tests ..... SUCCESS [ 0.632 s]
[INFO] hadoop-yarn-client ..... SUCCESS [ 3.747 s]
[INFO] hadoop-yarn-applications ..... SUCCESS [ 0.061 s]
[INFO] hadoop-mapreduce-distributedshell ..... SUCCESS [ 1.819 s]
[INFO] hadoop-mapreduce-client ..... SUCCESS [ 0.071 s]
[INFO] hadoop-mapreduce-client-core ..... SUCCESS [ 19.285 s]
[INFO] hadoop-yarn-applications-unmanaged-am-launcher ..... SUCCESS [ 1.634 s]
[INFO] hadoop-yarn-site ..... SUCCESS [ 0.126 s]
[INFO] hadoop-yarn-project ..... SUCCESS [ 9.379 s]
[INFO] hadoop-mapreduce-client-common ..... SUCCESS [ 12.987 s]
[INFO] hadoop-mapreduce-client-shuffle ..... SUCCESS [ 1.956 s]
[INFO] hadoop-mapreduce-client-app ..... SUCCESS [ 7.145 s]
[INFO] hadoop-mapreduce-client-hs ..... SUCCESS [ 4.449 s]
[INFO] hadoop-mapreduce-client-jobclient ..... SUCCESS [ 6.690 s]
[INFO] hadoop-mapreduce-client-hs-plugin ..... SUCCESS [ 1.716 s]
[INFO] Apache Hadoop MapReduce Examples ..... SUCCESS [ 4.989 s]
[INFO] hadoop-mapreduce ..... SUCCESS [ 3.062 s]
[INFO] Apache Hadoop MapReduce Streaming ..... SUCCESS [ 3.623 s]
[INFO] Apache Hadoop Distributed Copy ..... SUCCESS [ 12.446 s]
[INFO] Apache Hadoop Archives ..... SUCCESS [ 1.793 s]
[INFO] Apache Hadoop Rumen ..... SUCCESS [ 4.430 s]
[INFO] Apache Hadoop Gridmix ..... SUCCESS [ 4.198 s]
[INFO] Apache Hadoop Data Join ..... SUCCESS [ 2.107 s]
[INFO] Apache Hadoop Extras ..... SUCCESS [ 2.410 s]
[INFO] Apache Hadoop Pipes ..... SUCCESS [ 9.652 s]
[INFO] Apache Hadoop Tools Dist ..... SUCCESS [ 1.996 s]
[INFO] Apache Hadoop Tools ..... SUCCESS [ 0.031 s]
[INFO] Apache Hadoop Distribution ..... SUCCESS [ 14.857 s]
[INFO] Apache Hadoop Client ..... SUCCESS [ 5.414 s]
[INFO] Apache Hadoop Mini-Cluster ..... SUCCESS [ 0.124 s]
[INFO] -----
[INFO] BUILD SUCCESS
[INFO] -----
[INFO] Total time: 06:49 min

```

然后将编译出的 `native library` 库（路径 `hadoop-2.2.0-src/hadoop-dist/target/hadoop-2.2.0/lib/native`）替换到之前安装的hadoop的库目录中（路径 `hadoop-2.2.0/lib/native`）。ok，编译自己Mac上的库完成。

1.4. 测试hdfs和MapReduce

- `java`路径配置

按照PPT上的指示，建立并写入文件，这时没有报之前 `native library` 的错。但是进一步，要使用MapReduce的wordcount功能时，却报 `exitCode: 127` 错，是某个shell文件的命令在Mac上没有找到。查看该shell的错误日志：

```
1 | /bin/bash: /bin/java: No such file or directory
```

是因为Hadoop默认检查 `/bin/java` 路径下的java，可是Mac的Java不是装这里的，它的路径是 `/usr/bin/java`。类似前面的，建立软链接

```
1 | $ sudo ln -s /usr/bin/java /bin/java
```

却出现问题：`ln: /bin/java: Operation not permitted`。查百度后发现，这是因为苹果在OS X 10.11以后引入了SIP特性（系统完整性保护），使得即使加了`sudo`（也就是具有root权限）也无法修改系统级的目录，其中就包括了`/usr/bin`。要关闭SIP，需要重启并按住`Command+R`进入恢复模式，然后在终端内输入`csrutil disable`，再重启，建立软链接成功。再重启，`csrutil enable`打开SIP。

- 运行结果

1.wordcount

`file.txt`：

```
1 | a hot dog's hot hotdog
```

运行结果：

```
1 | a    1
2 | dog's 1
3 | hot   2
4 | hotdog 1
```

2.计算π

原理是随机生成点，通过大量的实验次数，计算落在圆内的次数/生成总次数得到。

Number of Maps	Number of samples	Result	Time Consumed(s)
2	10	3.8000000000000000000000000000000	16.309
5	10	3.2800000000000000000000000000000	24.507
10	10	3.2000000000000000000000000000000	25.221
2	100	3.1200000000000000000000000000000	16.261
10	100	3.1480000000000000000000000000000	30.627

分析性实验结果可知，number of maps和number of samples共同影响运行时间和计算得出的π值的精度。可以将map个数理解为有多少个机器参与并行计算，将sample个数理解为每台机器生成多少次随机点。发现number of maps对时间的影响更大，（比如(2,100)耗时短于(5,10)的耗时），猜测是由于事实上在我的单台计算机上并不是真的进行了并行计算，而是分次进行了5次，而每次并不是同时进行的。在每次重新计算之际，都要进行初始化、转移、储存等工作，这样就耗时增大。猜测在实际多台机器工作中，影响用时的唯一因素应该是samples个数。

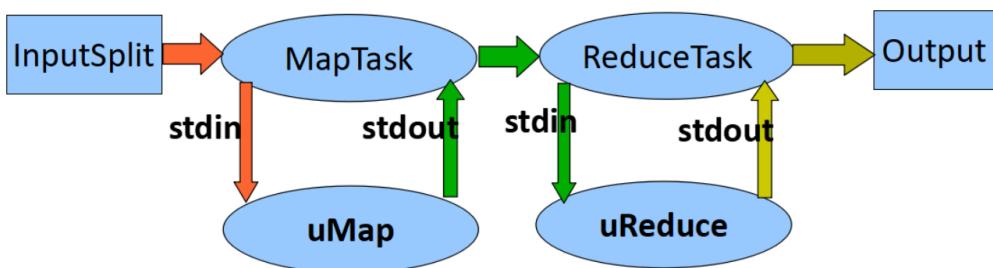
另外考虑精度，计算的总次数为number of maps × number of samples,所以精度和二者乘积相关。要精确到后五位，只需设置number of maps=1（为了耗时短），再递增number of samples直至符合要求。

2.练习MapReduce和HadoopStreaming

2.1.原理简述

正如前文的阐述，Hadoop的两个核心：HDFS为大数据提供储存，MapReduce为大数据提供计算。MapReduce是一种编程模型，用于大规模数据集（大于1TB）的并行运算。概念"Map（映射）"和"Reduce（归约）"，是它们的主要思想，都是从函数式编程语言里借来的，还有从矢量编程语言里借来的特性。它极大地方便了编程人员在不会分布式并行编程的情况下，将自己的程序运行在分布式系统上。当前的软件实现是指定一个Map（映射）函数，用来把一组键值对映射成一组新的键值对，指定并发的Reduce（归约）函数，用来保证所有映射的键值对中的每一个共享相同的键组。[\[1\]](#)

而Hadoop Streaming框架可以让任何语言编写的map, reduce程序能够在hadoop集群上运行；map/reduce程序只要遵循从标准输入stdin读，写出到标准输出stdout即可。并且其容易进行单机调试，通过管道前后相接的方式就可以模拟streaming，在本地完成map/reduce程序的调试。同时，streaming框架还提供了作业提交时的丰富参数控制，直接通过streaming参数，而不需要使用java语言修改；很多mapreduce的高阶功能，都可以通过streaming参数的调整来完成。[\[2\]](#)



2.2.word count 计数

- 实现思路

计算许多单词（essay, 字符串集群）中分别以a,b,..z开头的词的平均长度。首先在 `mapper.py` 中读完整的输入流，将字符串切分为单词后，对每个单词小写预处理，并初步判断是单词，然后返回的键值是开头的字母，值是这个字符串。然后在 `reducer.py` 中，用一个数组记录每个开头字母，对 `mapper.py` 传来的每个键值对处理，这样计算得每个开头字母对应的平均单词长度。

```
1 import sys
2 for line in sys.stdin:
3     line = line.strip()
4     words = line.split()
5     for word in words:
6         word=word.lower()
7         if (word[0]>='A' and word[0]<='Z'):
8             print '%s\t%s' % (word[0], word)
```

```

1 current_word = None
2 Sum=0
3 Total=0
4 word = None
5
6 for line in sys.stdin:
7     line = line.strip()
8     word, count = line.split('\t', 1)
9     try:
10         count = len(count)
11     except ValueError:
12         continue
13     if current_word == word:
14         Sum += count
15         Total+=1
16     else:
17         if current_word:
18             ans=float(Sum)/Total
19             print '%s\t%s' % (current_word, ans)
20         Sum=count
21         Total=1
22         current_word = word
23
24 if current_word == word:
25     ans = float(Sum) / Total

```

• 实验结果

```

markdemacbook-Pro:~ markdana$ hadoop jar /users/markdana/hadoop-2.2.0/share/hadoop/tools/lib/hadoop-streaming-2.2.0.jar tempinput tempoutput
18/11/15 16:30:05 INFO client.PMProxy: Connecting to ResourceManager at /0.0.0.0:8032
18/11/15 16:30:05 INFO client.PMProxy: Connecting to ResourceManager at /0.0.0.0:8032
18/11/15 16:30:05 INFO mapred.FileInputFormat: Total input paths to process : 3
18/11/15 16:30:05 INFO mapreduce.JobSubmitter: number of splits:3

18/11/15 16:30:06 INFO mapreduce.Job: The url to track the job: mapreduce.Job: To
he url to track the job: http://markdemacbook-pro.local:8088/proxy/application_1
542271066351_0007/
18/11/15 16:30:06 INFO mapreduce.Job: Running job: job_1542271066351_0007
18/11/15 16:30:08 INFO mapreduce.Job: Job job_1542271066351_0007 running in uber
mode : false
18/11/15 16:30:11 INFO mapreduce.Job: map 0% reduce 0%
18/11/15 16:30:19 INFO mapreduce.Job: map 33% reduce 0%
18/11/15 16:30:30 INFO mapreduce.Job: map 100% reduce 0%
18/11/15 16:30:35 INFO mapreduce.Job: map 100% reduce 100%
18/11/15 16:30:35 INFO mapreduce.Job: Job job_1542271066351_0007 completed succe
ssfully
18/11/15 16:30:35 INFO mapreduce.Job: Counters: 42

```

Goto : </user/markdانا/tempoutput/part-00000> go

[Go back to dir listing](#)

[Advanced view/download options](#)

a	3.489425
b	4.819150
c	6.973598
d	6.090473
e	6.410739
f	5.239738
g	5.862938
h	4.121992
i	3.223825
j	5.106672
k	5.540404
l	5.526044
m	5.343117
n	4.680581
o	2.938513
p	6.931871
q	6.520141
r	6.782817
s	5.804458
t	3.731301
u	5.227670
v	6.174784
w	4.630077
x	4.640103
y	3.917664

[Download this file](#)

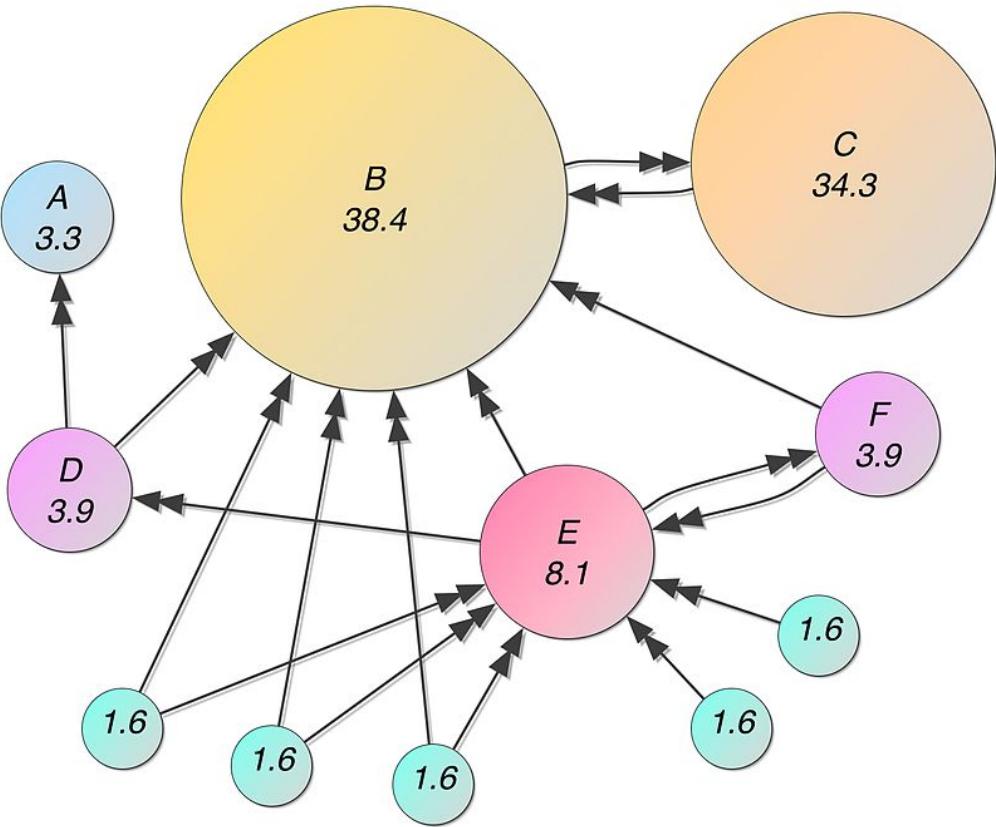
统计结果

2.3. Page Rank

• 原理简述

搜索引擎的两个核心问题，其一是建立资料库，通过爬虫实现；其二是建立一种数据结构，可以根据关键词找到含有这个词的页面，通过倒排索引实现。现代搜索引擎的本质也和这个简单模型一样，Google公司在这两点上也和前辈并无二致，但是创始人佩奇(Larry Page)和布林(Sergey Brin)的研究解决了第三点，也就是对于用户最重要的一点：对查询结果进行排序。在之前用过检索词评价等算法，但是却囿于网页内容的限制，容易造成Term Spam攻击。

PageRank算法是基于这样一种背景思想：被用户访问越多的网页更可能质量越高，而用户在浏览网页时主要通过超链接进行页面跳转，因此我们需要通过分析超链接组成的拓扑结构来推算每个网页被访问频率的高低。将整个web视作一个n个节点的强连通图， $n \times n$ 维转移矩阵M来描述用户在某个页面时访问另一个页面的概率， n 维向量v记录当前的每个页面的Rank，初始设置为。经过几次 $M \cdot v$ 迭代后，可以证明趋于收敛，即 $M \cdot v = v$ ，这就是每个网页最终的PageRank。



至于实时更新、处理Dead Ends、Spider Traps及平滑处理则是另外的细节。比如为了防止Dead Ends（某个网页的rank一直为0），设置系数 a ，一般取0.85，即认为85%的情况下，通过外链去访问下一个网页，但也有15%的可能随机地打开一个网页，则每轮迭代后将 v 乘以 a ，在分别加上 $(1-a)/n$ 。

- 基于MapReduce的迭代实现

计算方法其实较简单，难点在于如何处理mapper和reducer之间输入输出流的传递。在 `mapper.py` 中，先通过 `for line in sys.stdin` 读入所有数据，创建字典格式的矩阵（也可以使用 `numpy` 库，但由于只是演示实验，且考虑到读入读出的方便性，这次暂时只用了字典储存），键是ID，值是LinkID和LinkRank，执行一次矩阵乘法和一次向量加法即完成一次迭代。需要注意的是，`mapper.py` 中全部运算需要在 `for line in sys.stdin` 下完成。然后在 `reducer.py` 中，处理并储存当前的运算结果。

写shell脚本，直接运行之即可。

- 实验结果

Contents of directory [/user/markdana](#)

Goto : go

[Go to parent directory](#)

Name	Type	Size	Replication	Block Size	Modification
QuasiMonteCarlo_1541846348716_255459791	dir				2018-11-
tempinput	dir				2018-11-
tempinput_	dir				2018-11-
tempoutput_1	dir				2018-11-
tempoutput_10	dir				2018-11-
tempoutput_11	dir				2018-11-
tempoutput_12	dir				2018-11-
tempoutput_13	dir				2018-11-
tempoutput_14	dir				2018-11-
tempoutput_15	dir				2018-11-
tempoutput_16	dir				2018-11-
tempoutput_17	dir				2018-11-
tempoutput_18	dir				2018-11-
tempoutput_19	dir				2018-11-
tempoutput_2	dir				2018-11-
tempoutput_20	dir				2018-11-
tempoutput_21	dir				2018-11-
tempoutput_22	dir				2018-11-
tempoutput_23	dir				2018-11-
..

每次迭代生成的输出文件

A	B	C	D
[0.0375	0.32083333	0.21458333	0.42708333]
[0.0375	0.41114583	0.18447917	0.366875]
[0.0375	0.35996875	0.22286198	0.37966927]
[0.0375	0.37084388	0.20111172	0.3905444]
[0.0375	0.38008774	0.20573365	0.37667861]
[0.0375	0.36830182	0.20966229	0.38453589]
[0.0375	0.37498051	0.20465327	0.38286622]
[0.0375	0.37356129	0.20749172	0.381447]
[0.0375	0.37235495	0.20688855	0.38325651]
[0.0375	0.37389303	0.20637585	0.38223112]
[0.0375	0.37302145	0.20702954	0.38244901]
[0.0375	0.37320666	0.20665912	0.38263422]
[0.0375	0.37336409	0.20673783	0.38239808]
[0.0375	0.37316337	0.20680474	0.38253189]
[0.0375	0.37327711	0.20671943	0.38250346]
[0.0375	0.37325294	0.20676777	0.38247929]
[0.0375	0.3732324	0.2067575	0.38251011]
[0.0375	0.37325859	0.20674877	0.38249264]
[0.0375	0.37324375	0.2067599	0.38249635]
[0.0375	0.3732469	0.20675359	0.38249951]
[0.0375	0.37324958	0.20675493	0.38249549]
[0.0375	0.37324616	0.20675607	0.38249776]
[0.0375	0.3732481	0.20675462	0.38249728]
[0.0375	0.37324769	0.20675544	0.38249687]
[0.0375	0.37324734	0.20675527	0.38249739]
[0.0375	0.37324778	0.20675512	0.3824971]
[0.0375	0.37324753	0.20675531	0.38249716]
[0.0375	0.37324759	0.2067552	0.38249721]
[0.0375	0.37324763	0.20675522	0.38249714]
[0.0375	0.37324757	0.20675524	0.38249718]

迭代30次，基本已稳定

File: /user/markdana/tempoutput_30/part-00000

Goto : /markdana/tempoutput_30

[Go back to dir listing](#)
[Advanced view/download options](#)

A	0.037500
B	0.320833
C	0.214583
D	0.427083

收敛结果