最短路径系列
之一

# 从零开始学习 Hadoop

作者三个主站：
csdn 主页：http://blog.csdn.net/lizhe_dashuju
豆瓣主页：http://www.douban.com/people/79049984/
新浪博客：http://blog.sina.com.cn/u/2436144507
敬请关注，谢谢！

# 内容目录

# 前言

Hadoop 是最著名使用最广泛的分布式大数据处理框架，它是用 Java 开发的。

这本书有一个明确的目标：只要有一台能上网的计算机，就可以让读者在最短的时间内，学会 Hadoop 的初级开发。所以，这本书只讲干货，也就是必须要知道的 Hadoop 的最核心知识点，包括如何搭建 Hadoop，如何写代码，如何查 API。对于更多细节问题，书后会列一个书目给大家参考，这些书已经讲得很好了，不需要在这里饶舌浪费时间。

这本书对读者的预要求很少：懂一点点 Linux，懂一点点 Java 编程。“懂一点点 Linux”是指，假如有一台装了 Linux 操作系统的计算机，能做到开机，输入密码，进入图形界面，打开命令终端，就可以了。“懂一点点 Java 编程”是指，在 Linux 上，用记事本写一个 Java 的 HelloWorld 程序，然后把它编译出来，执行一下，就可以了。这个预要求非常低，接近于零，所以这本书叫“从零开始学习 Hadoop”。Hadoop 是用 Java 开发的，它通过 Streaming 方式支持其他语言，诸如 Python，C++，Ruby，但如果想真正理解 Hadoop，必须从 Java 开始，用其他语言以 Streaming 方式学 Hadoop 是一条不归路，这是笔者用一个月时间得来的教训。

“只要有一台能上网的计算机”，Hadoop 的伪分布式运行模式，可以在一台电脑上运行 Hadoop 的全部功能。在伪分布式下编写和运行的代码，不作任何修改便能运行在 Hadoop 集群上，这是学习 Hadoop 的最方便的优点之一。“最短的时间”，这是一个有点长的话题。学习一项技能，有很多种途径。最快的途径找个高手教，高手会说看哪些书从哪里做一下，哪些地方注意不要犯错，哪些地方是不需要的不要在上面浪费时间，哪些地方很关键要多做做将来会很有用。只要没有出现诸如高手数量不多，或者高手很忙，或者高手是同事但职位只比你高一点点—教会你对他其实没什么好处的这些情况之外，这个途径是最好的。

另外一个途径是报一些培训班学习，费用相对高一些。

性价比最高的途径是自学，如果自学能力比较强，从网上找相关的电子书教学视频官方教程学习，笔者个人比较推崇这种方式，这种方式可以很好地锻炼学习能力。能力是一得永得的，将来学习其他技能可以举一反三，缺点是比较耗时，自学的过程其实很大程度上是试错的过程。如果技术比较复杂，而且工作中很快要用到，或者下个星期就要面试笔试，那肯定是妥妥地来不及的。

如何在最短的时间内解决 Hadoop 入门问题呢？理性的方式，是从流程和细节上解决。1898 年，科学管理之父泰勒做了一个铁块搬运实验。当时，工人们每天平均搬运 12 ~ 13 吨铁块。泰勒的实验方式是对搬运过程计时，分析不同搬运方式的影响，分析休息时间和劳动时间的搭配，最终可以让工人在不太累的情况下，每天搬运 47 吨铁块，效率提高到近 4 倍。

事实上，可以用同样的方式解决学习问题。比如说，请一个对 Hadoop 一无所知的小白同学自学 Hadoop，然后要求他记下每天学了哪些内容，遇到什么问题，是怎么解决的，分别花了多少时间，有哪些是必须的，有哪些后来验证是弯路，哪些可以省略。待到他学完了，根据这个学习记录整理出一条快捷之路，避开没用的地方，简化过于复杂的地方，调整到合理的次序，于是就得到一个“最短的时间”学习方案。这本书就是这么产生的，而且是真简化。

“做一遍”是最佳的快速学习方式。比如说，学习 C 语言编程，如果学习方式是看书，读一遍乃至读十遍，不写代码，不会在大脑留下任何痕迹。如果将书中的例题代码原样输入编译执行，理解会好很多。如果不但调通了例题，而且将书里的大部分习题独立做出来，会觉得自己对 C 语言很熟悉。如果写了一个有质量的五千到一万行的项目，就觉得自己真的可以用 C 语言做工作。所以说，做一遍是最好的学习方式。

笔者见过一些高效的人是以反其道而行之的方式学习的。当他们需要用到一种新技术的时候，就找一本相关的书，看一下目录，再看一下每章的简介，大略看看范例。然后，根据范例立刻写代码，需要什么功能，就到书里和 API 里找，如果遇到 Bug，在 Google 上检索一下。这种方式会很快做完功能，同时也就学会这个技术。如果从头开始看书，一点一点做例题做习题，对他们来说太慢了，时间成本太高，很不划算。

还有另外一种方式，笔者将它称之为吉祥物大师法（Master Mascot），注意，不要看成吉祥物大法师。一些有趣的技术公司会给自己设计吉祥物玩具。假如员工在工作中遇到解决不了的问题，按照“潜规则”，要先把这个问题讲给吉祥物玩具听，多数情况下讲完了自己就知道答案了，这就是吉祥物大师法。

这本书的例子，是按照“做一遍”的方式编写的。每个例子都是完整的，从目录和文件结构，到编译打包运行。这种完整很重要，可以大幅度减少试错，查询 API，编译和调试的时间。熟悉这些范例，先手工输入和编译，然后再合上书，自己从零开始做一遍，对着代码找感觉和体会。

尤其非常重要的是，要想象如果自己是 Hadoop 的作者，会怎么设计 Hadoop，用这种眼光审视 Hadoop，会突然之间“开悟”，原来如此。这种“开悟”，要自己走过一遍，想过一遍，做过一遍，才会达到。一旦“开悟”了，会信心满满，应对这个体系架构的诸多问题，会漂亮地解决未知问题，就像厨师做一道好菜，像浪子泡 MM。这种书有点像武侠小说里金世遗练的邪派功夫，简单，见效快，但见效之后要补上一些细节，免得走火入魔。所谓走火入魔就是觉得 Hadoop 如此简单，没什么搞头，其实 Hadoop 真的是很有搞头。

如果你对 Java 很熟悉，对 Linux 很熟悉，这本书对你来说太简单了，只需要一个下午就能搞定。Hadoop 很大，但大部分东西都是在 Linux 出现过。你只需要安装它，然后手写一下例题，编译，运行，然后去看看 Hadoop 的 API，就足够了。

未来将根据这本书发布一系列的视频，更为直观。本书的理念是让零起点的读者根据这本书能在一个星期或者更短的时间内学会 Hadoop 的初步技能，快速让大家安装 Hadoop，快速跑一个 Hadoop 的 Demo，快速了解 Hadoop 的运行流程，写一些初级的 Hadoop 的应用 ，以应对诸如快速开发，面试笔试，大数据课程实践等需求。如果需要掌握更多的 Hadoop 细节，请参考本书的推荐书目，或者关注“最短路径系列”的未来书目。

# 第 1 章 Hadoop 的安装

Hadoop 的安装比较繁琐，有如下几个原因：其一，Hadoop 有非常多的版本；其二，官方文档不尽详细，有时候更新脱节，Hadoop 发展的太快了；其三，网上流传的各种文档，或者是根据某些需求定制，或者加入了不必须要的步骤，或者加入容易令人误解的步骤。其实安装是很重要的步骤，只有安装好了，才能谈及下一步。
在本书撰写的时候，选用 Hadoop 的 stable 版安装。

笔者的登录用户名是 brian，大家可以根据自己的登录名更改命令，后面凡是出现 brian 的地方，都用自己的登录用户名替换掉。

## 1. 操作系统

操作系统是 Ubuntu 10.04 桌面版。

如果操作系统其他版本的 Ubuntu，在图形界面上会略有一点区别，但对安装影响不大。不同发行版的 Linux 的安装 Hadoop 的过程基本类似，没太大的差别。

## 2. Hadoop 的版本

Hadoop 当前的 stable 版是 1.2.1。

## 3. 下载 Hadoop

3.1 在 Hadoop 的主页上提供了多个下载链接。
http://www.apache.org/dyn/closer.cgi/hadoop/common/
3.2 任选一个下载站点如下：
http://mirror.esocc.com/apache/hadoop/common/
3.3 选择 stable 版，其实 stable 版就是 1.2.1 版：
http://mirror.esocc.com/apache/hadoop/common/stable/
在这个目录下有多个文件，是针对不同的 linux 发行版的，不需要全部下载。
3.4 下载 hadoop-1.2.1.tar.gz 和 hadoop-1.2.1.tar.gz.mds
打开命令终端，下文的命令都是在终端里执行，为方便起见，命令都用引号引起。
将 stable 版本的 Hadoop 的两个文件下载到“~/setup/hadoop”目录下，也就是”/home/brian/setup/hadoop”目录，命令如下：

3.4.1 “mkdir -p ~/setup/hadoop”

mkdir 命令是创建新目录。”-p”参数的意思是，假如 hadoop 目录的上级目录不存在，也创建上级目录。在终端里执行“man mkdir”，可以看到对这个命令的更详细的解释，按一下 q 键重新返回终端。

在命令终端里，”~”表示当前登录用户的主目录。比如说，在开机的时候，登录用户是 brian，那么在命令终端里，”~”就表示目录”/home/brian”，如果开机时候，登录用户是 john，那么”~”就表示”/home/john”目录。

3.4.2 "cd ~/setup/hadoop"

cd 就是 change directory 的缩写，切换当前目录。

3.4.3 "wget http://mirror.esocc.com/apache/hadoop/common/stable/hadoop-1.2.1.tar.gz.mds"

wget 是下载文件的命令行工具，”man wget”有详细说明。

3.4.4 "wget http://mirror.esocc.com/apache/hadoop/common/stable/hadoop-1.2.1.tar.gz"

3.4.5 "md5sum hadoop-1.2.1.tar.gz"

md5sum 命令，计算一个文件的 md5 码。开源社区在提供源码下载的时候，会同时提供下载文件的 md5 码。md5 码是根据文件内容生成的 32 位字符串，不同的文件的 md5 码是不同的，如果下载出错，下载文件的 md5 码跟正常文件的 md5 码是不一样的，由此检测下载是否正常，只有在极其罕见的情况下，才会出现不同的文件有相同 md5 码。hadoop-1.2.1.tar.gz 是一个比较大的文件，需要检查下载的文件是否完整，执行这个命令之后，会出现形如"8D 79 04 80 56 17 C1 6C  B2 27 D1 CC BF E9 38 5A  hadoop-1.2.1.tar.gz"的字符串，前面的一串字符串就是 32 位的 md5 校验码。

3.4.6 "cat hadoop-1.2.1.tar.gz.mds"，

cat 命令，cat 是 catenate 的缩写，在标准输出上打印文件内容，通常标准输出就是屏幕。这个命令会在屏幕上打印 hadoop-1.2.1.tar.gz.mds 的内容，也就是一些校验码，在里面找到"md5"这一行，如果跟 md5sum 出来的一致，则表明下载文件完整的，否则需要重新下载。

# 4. 安装 Java JDK

4.1 在这里有 jdk 1.7 的下载

http://www.oracle.com/technetwork/java/javase/downloads/jdk7-downloads-1880260.html

如果是 CPU 是 32 位，选择下载 Linux x86，如果 CPU 是 64 位的，选择 Linux x64。一般来说，如果计算机是双核的，肯定支持 64 位操作系统。或者可以运行”uname -a”命令看一下，在笔者的笔记本上运行这个命令结果如下：

Linux brian-i3 2.6.32-51-generic #113-Ubuntu SMP Wed Aug 21 19:46:35 UTC 2013 x86_64 GNU/Linux

后面的 x86_64 表明系统是 64 位的。

在这个页面，找 ”Java SE Development Kit 7u40”，注意，这里有一个选项，必须选择”Accept License Agreement”，接受 License 才能下载。

下载的 jdk 1.7，存放到 “/home/brian/setup/java-jdk-1.7/”目录。

下载的文件是”java-jdk-7u40-linux-i586.tar.gz”，java jdk 的版本常常有更新，次版本号有可能会比 40 更高一点。

4.2 "sudo su -"

切换到 root 用户，参考”man sudo”。这个命令会切换到 root 用户，也就是最高权限的用户。因为后面要执行的 jdk 安装操作是在/usr/local 目录下进行的，用 root 用户更方便。

4.3 "cd /usr/local/lib"

4.4 "tar -zxvf /home/brian/setup/java-jdk-1.7/java-jdk-7u40-linux-i586.tar.gz"

tar 是 linux 下的打包和解压命令行工具，具体细节可以参考”man tar”。这个命令将 java-jdk-7u40-linux-i586.tar.gz 压缩包解压到当前目录下。解压缩完毕之后，执行"ls"，能看到当前目录下有一个新目录叫"jdk1.7.0_40"

4.5 配置环境变量：

4.5.1 “gedit /etc/profile”

gedit 是 linux 下类似 Windoes 的记事本的编辑器，文件/etc/profile 是 linux 下的配置文件。本命令会打开这个配置文件，以备编辑。

4.5.2 添加配置

在/etc/profile 文件末尾加上如下的三行代码：

```
export JAVA_HOME=/usr/local/lib/jdk1.7.0_40
export CLASSPATH=.:$JAVA_HOME/jre/lib/rt.jar:$JAVA_HOME/lib/dt.jar:$JAVA_HOME/lib/tools.jar
export PATH=$PATH:$JAVA_HOME/bin
```

保存文件，然后退出。

Linux 系统，开机后会自动执行/etc/profile 配置文件。export 命令设置或者显示环境变量。上述三行代码，分别设置了 JAVA_HOME, CLASSPATH, PATH 这三个环境变量。

4.5.3 "chown root:root -R /usr/local/lib/jdk1.7.0_40"

chown 命令，更改目录或者文件的拥有者。这条命令将 jdk1.7.0_40 目录的拥有者改为 root 组的 root 用户。”-R”参数是递归的意思，将 jdk1.7.0_40 目录下连同子目录都进行更改。

4.5.4 "chmod 755 -R /usr/local/lib/jdk1.7.0_40"

chmod 命令，更改目录和文件的模式。本命令将 jdk1.7.0_40 的模式改为拥有者可以读写执行，同组用户和其他用户可读可执行不可写。“-R”参数同上，也是递归的意思。

4.5.5 "source /etc/profile"

如果更改了/etc/profile 配置文件，它只会在新的终端里生效，现在正在使用的终端是不会生效的。如果想让它在正使用的终端也生效，需要用 source 命令运行一下配置文件。这条命令会让 4.4.2 的三个环境变量立即生效。这条命令也可以简写成”. /etc/profile”。

4.5.6 "java -version"

这条命令检查 jdk 安装是否成功。运行这条命令，只要没有报错就表明安装成功了。

# 5. 安装 hadoop

5.1 "su brian"

su 命令，切换用户。安装 jdk 用的是 root 用户。现在切回 brian 用户。

5.2 "mkdir -p ~/usr/hadoop"

创建 Hadoop 的安装目录

5.3 "cd ~/usr/hadoop"

5.4 "tar -xvzf ~/setup/hadoop/hadoop-1.2.1.tar.gz"

解压缩完毕后，就有目录~/usr/hadoop/hadoop-1.2.1，这是 hadoop 的主目录。

5.5 配置 hadoop，参考了 http://hadoop.apache.org/docs/stable/single_node_setup.pdf。

按照伪分布式进行配置，也就是用一个机器同时运行 NameNode, SecondaryNameNode, DataNode, JobTracker, TaskTracker 5 个任务。

5.5.1 配置文件在~/usr/hadoop/hadoop-1.2.1/conf/目录下

5.5.2 将 core-site.xml 文件内容修改成如下：

```
<configuration>
    <property>
```

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

5.5.3 将 mapred-site.xml 文件内容修改如下：

```
<configuration>
    <property>
        <name>mapred.job.tracker</name>
        <value>localhost:9001</value>
    </property>
</configuration>
```

5.5.4 将 hdfs-site.xml 文件内容修改如下：

```
<configuration>
    <property>
        <name>dfs.replication</name>
        <value>1</value>
    </property>
</configuration>
```

5.5.5 在 hadoop-env.sh 文件里添加如下一条语句：

```
export JAVA_HOME=/usr/local/lib/jdk1.7.0_40
```

# 6. 安装 rsync 和 ssh

6.1 "sudo apt-get install ssh rsync"

这条命令安装 ssh 和 rsync。ssh 是一个很著名的安全外壳协议 Secure Shell Protocol。rsync 是文件同步命令行工具。

6.2 配置 ssh 免登录

6.2.1 "ssh-keygen -t dsa -f ~/.ssh/id_dsa"

执行这条命令生成 ssh 的公钥/私钥，执行过程中，会一些提示让输入字符，直接一路回车就可以。

6.2.2 "cat ~/.ssh/id_dsa.pub >> ~/.ssh/authorized_keys"

ssh 进行远程登录的时候需要输入密码，如果用公钥/私钥方式，就不需要输入密码了。上述方式就是设置公钥/私钥登录。

6.2.3 “ssh localhost”

第一次执行本命令，会出现一个提示，输入”yes”然后回车即可。

# 7. 启动 hadoop

7.1 "cd ~/usr/hadoop/hadoop-1.2.1"

7.2 "./bin/hadoop namenode -format"

格式化 NameNode。

7.3 "./bin/start-all.sh"

启动所有节点，包括 NameNode, SecondaryNameNode, JobTracker, TaskTracker, DataNode。

7.4 “jps”

检查各进程是否运行，这时，应该看到有 6 个 java 虚拟机的进程，分别是 Jps, NameNode, SecondaryNameNode, DataNode, JobTracker, TaskTracker，看到 6 个是对的，表明启动成功。如果提示”jps”没安装或者找不到，执行一次”source /etc/profile”即可。

# 8. 测试 hadoop

8.1 "cd ~/usr/hadoop/hadoop-1.2.1"

8.2 "./bin/hadoop fs -put README.txt readme.txt"

将当前目录下的 README.txt 放到 hadoop 进行测试，这个 README.txt 是 Hadoop 的介绍文件，这里用它做测试。这条命令将 README.txt 文件复制到 Hadoop 的分布式文件系统 HDFS，重命名为 readme.txt。

8.3 "./bin/hadoop jar hadoop-examples-1.2.1.jar wordcount readme.txt output"

运行 hadoop 的 examples 的 wordcount，测试 hadoop 的执行。这条语句用 Hadoop 自带的 examples 里的 wordcount 程序，对 readme.txt 进行处理，处理后的结果放到 HDFS 的 output 目录。

8.4 "./bin/hadoop fs -cat output/part-r-00000"

这条命令查看处理结果，part-r-00000 文件存放 wordcount 的运行结果，cat 命令将文件内容输出到屏幕，显示字符的统计结果。这是一个简单的字符统计，wordcount 只是做了简单的处理，所以会看到单词后面有标点符号。

# 9. 练习

笔者做一次完整的安装是 50 分钟左右，其中下载 Hadoop 安装包和 Java JDK 安装包是半小时，操作部分用时 20 分钟。新手第一次安装，2~5 个小时内完成都是正常的。建议将 Hadoop 的安装过程按照上述流程走上三遍，熟悉每个步骤，然后不看流程凭记忆做出来，重复练习多次次，以加深印象。如果再有时间的话，可以逐个研究里面涉及到的各种命令，诸如 wget, ssh, rsync 等等。

# 第 2 章 第一个 MapReduce 程序

## 1. Hadoop 从头说

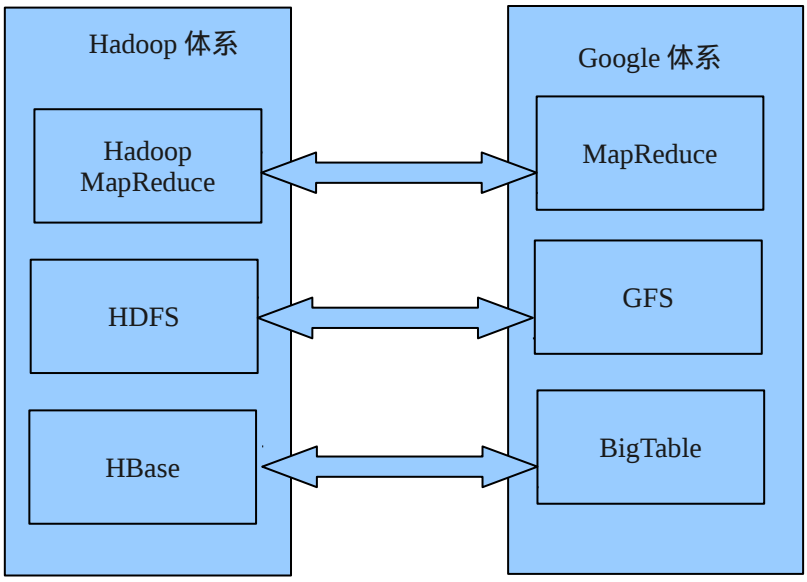
### 1.1 Google 是一家做搜索的公司

做搜索是技术难度很高的活。首先要存储很多的数据，要把全球的大部分网页都抓下来，可想而知存储量有多大。然后，要能快速检索网页，用户输入几个关键词找资料，越快越好，最好在一秒之内出结果。如果全球每秒有上亿个用户在检索，只有一两秒的检索时间，要在全球的网页里找到最合适的检索结果，难度很大。

Google 用三个最重要的核心技术解决上述问题，它们分别是 GFS, MapReduce 和 BigTable。Google 发表了它们的设计论文，但没有将它们开源，核心竞争力不可能开源的。论文在这里，有兴趣的同学可以去看看：GFS ， http://labs.google.com/papers/gfs-sosp2003.pdf ； MapReduce ， http://labs.google.com/papers/mapreduce-osdi04.pdf
；Bigtable，http://labs.google.com/papers/bigtable-osdi06.pdf。

Google 的论文发表之后，Doug Cutting 等人根据论文的思想，在开源项目 Nutch 的基础上实现了 Hadoop。后来，Doug Cutting 去了 Yahoo，继续做 Hadoop。后来，Hadoop 的开发和应用开始爆发了。

在对应关系上看，Hadoop MapReduce 对应 MapReduce，Hadoop Distributed File System (HDFS)对应 GFS，HBase 对应 BigTable。一般我们所说的 Hadoop 其实是指 Hadoop 体系，它包括 Hadoop MapReduce，HDFS，HBase，还有其他更多的技术。



## 1.2 MapReduce 和 HDFS 是如何工作的

先用一种有助于理解的方式描述 MapReduce 和 HDFS 是如何工作的。假如有 1000G 的多个文本文件，内容是英文网页，需要统计词频，也就是哪些单词出现过，各出现过多少次，有 1000 台计算机可供使用，要求速度越快越好。最直接的想法是，把 1000G 的文件分成 1000 份，每台机器处理 1G 数据。处理完之后，其他 999 台机器将处理结果发送到一台固定的机器上，由这台机器进行合并然后输出结果。

Hadoop 将这个过程进行自动化的处理。首先看如何存储这 1000G 的文本文件。HDFS 在这 1000 台机器上创建分布式文件系统，将 1000G 的文件切分成若干个固定大小的文件块，每个块一般是 64M 大小，分散存储在这 1000 台机器上。这么多机器，在运行的时候难免会出现有几台突然死机或者挂掉的情况，这导致上面存储的文件块丢失，会导致计算出错。为避免这种情况，HDFS 对每个文件块都做复制，复制成 3 ~ 5 个相同的块，放到不同的机器上，这样死机的文件块在其他机器上仍然可以找得到，不影响计算。

MapReduce 其实是两部分，先是 Map 过程，然后是 Reduce 过程。从词频计算来说，假设某个文件块里的一行文字是”This is a small cat. That is a small dog.”，那么，Map 过程会对这一行进行处理，将每个单词从句子解析出来，依次生成形如<“this”, 1>, <”is”, 1>, <”a”, 1>, <”small”, 1>, <”cat”, 1>, <”that”, 1>, <”is”, 1>, <”a”, 1>, <”small”, 1>, <”dog”, 1>的键值对，<”this”, 1>表示“this”这个单词出现了 1 次，在每个键值对里，单词出现的次数都是 1 次，允许有相同的键值对多次出现，比如<”is”,1>这个键值对出现了 2 次。Reduce 过程就是合并同类项，将上述产生的相同的键值对合并起来，将这些单词出现的次数累加起来，计算结果就是<“this”, 1>, <”is”, 2>, <”a”, 2>, <”small”, 2>, <”cat”, 1>, <”that”, 1>, <”dog”, 1>。这种方式很简洁，并且可以进行多种形式的优化。比如说，在一个机器上，对本地存储的 1G 的文件块先 Map，然后再 Reduce，那么就得到了这 1G 的词频统计结果，然后再将这个结果传送到远程机器，跟其他 999 台机器的统计结果再次进行 Reduce，就得到 1000G 文件的全部词频统计结果。如果文件没有那么大，只有三四个 G，就不需要在本地进行 Reduce 了，每次 Map 之后直接将结果传送到远程机器做 Reduce。

具体地，如果用 Hadoop 来做词频统计，流程是这样的：

1)先用 HDFS 的命令行工具，将 1000G 的文件复制到 HDFS 上；

2)用 Java 写 MapReduce 代码，写完后调试编译，然后打包成 Jar 包；

3)执行 Hadoop 命令，用这个 Jar 包在 Hadoop 集群上处理 1000G 的文件，然后将结果文件存放到指定的目录。

4)用 HDFS 的命令行工具查看处理结果文件。

## 1.3 API 参考

开发过程需要的 API 全部在 Java API 和 Hadoop API，在下面两个地方找：

Hadoop 1.2.1 的 API 文档：http://hadoop.apache.org/docs/r1.2.1/api/index.html

Java JDK1.7 的 API 文档：http://docs.oracle.com/javase/7/docs/api/

# 2. 词频统计

在这里，我们开始实现 Word Count 的 MapReduce。这里的 Word Count 程序是从 Hadoop 的例子代码改编来的。

# 3. 标准形式的 MapReduce 程序

所谓标准形式的 MapReduce，就说需要写 MapReduce 的时候，脑海里立刻跳出的就是这个形式，一个 Map 的 Java 文件，一个 Reduce 的 Java 文件，一个负责调用的主程序 Java 文件。这个标准形式已经是最简了，没有多余的东东可以删除，没有肥肉，是干货。写 MapReduce 和主程序的时候，分别引用哪些包哪些类，每个包每个类是什么作用，这些要很清晰。如果记不住的话，将这些代码写几遍，编译调试运行，然后不看代码，自己从头写出来，编译调试运行，重复多次应该可以记住了。

## 3.1 目录和文件结构

首先创建一个目录 wordcount_01 存放源代码、编译和打包结果，比如将这个目录放在/home/brian/wordcount_01。

wordcount_01 目录下，有两个子目录，分别是 src 目录和 classes 目录。src 目录存放 Java 的源代码，classes 目录存放编译结果。在 src 目录下，创建三个文件，分别是 IntSumReducer.java，TokenizerMapper.java，WordCount.java。从 MapReduce 的角度看， TokenizerMapper.java 文件是做 Map 的代码，IntSumReducer.java 是做 Reduce 的代码，WordCount.java 是主程序，负责执行整个流程。这三个 Java 文件内容在下面给出。

## 3.2 TokenizerMapper .java 文件的源代码

```
package com.brianchen.hadoop;

import java.io.IOException;
import java.util.StringTokenizer;

import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Mapper;

public class TokenizerMapper
      extends Mapper<Object, Text, Text, IntWritable>{
  IntWritable one = new IntWritable(1);
  Text word = new Text();

  public void map(Object key, Text value, Context context)
      throws IOException, InterruptedException {
    StringTokenizer itr = new StringTokenizer(value.toString());
    while (itr.hasMoreTokens()){
      word.set(itr.nextToken());
      context.write(word, one);
    }
  }
}
```

下面逐行解释代码，所有的类更详细的资料其实都可以在 1.3 节的两个 API 地址里找到：

1）“package com.brianchen.hadoop”

Java 提供包机制管理代码，关键词就是 package，可以随意指定一个包的名字，诸如笔者的就是”com.brianchen.hadoop”，只要不跟其他的包重复就可以。为了保证包的唯一性，Sun 公司推荐用公司的域名的逆序作为包名，于是大家就在代码里看到诸如”org.apache.hadoop”之类的包名。

2)”import java.io.IOException”

凡是以 java 开头的包，在 JDK1.7 的 API 里找类的资料。这一句从 java 的 io 包里导入 IOException。IOException，输入输出异常类。所谓异常，就是 Exception，就是程序出错了，异常机制是 Java 的错误捕获机制。那么，IOException 就是处理输入输出错误时候的异常，I 是 Input，O 是 Output。

3) “import java.util.StringTokenizer”

从 java 的 util 包引入 StringTokenizer 类。StringTokenizer 将符合一定格式的字符串拆分开。比如说，”This is a cat” 是一个字符串，这四个单词是用空格符隔开的，那么 StringTokenizer 可以将它们拆成四个单词”This”，“is”，”a”，“cat”。如果是用其他符号隔开，也能处理，比如”14;229;37”这个字符串，这三个数字是分号隔开的，StringTokenizer 将它们拆成”14”，“229”，“37”。只要指定了分隔符，StringTokenizer 就可以将字符串拆开。“拆开”的术语叫“解析”。

4)”import org.apache.hadoop.io.IntWritable”

凡是以 org.apache.hadoop 开头的包，在 Hadoop 1.2.1 的 API 找类的详细信息。从 hadoop 的 io 包里引入 IntWritable 类。IntWritable 类表示的是一个整数，是一个以类表示的整数，是一个以类表示的可序列化的整数。在 Java 里，要表示一个整数，假如是 15，可以用 int 类型，int 类型是 Java 的基本类型，占 4 个字节，也可以用 Integer 类，Integer 类封装了一个 int 类型，让整数成为类。Integer 类是可以序列化的。但 Hadoop 觉得 Java 的序列化不适合自己，于是实现了 IntWritable 类。至于什么是序列化，这个问题比较长，这个问题会在后面章节详细讲。

5) “import org.apache.hadoop.io.Text”

从 hadoop 的 io 包里引入 Text 类。Text 类是存储字符串的可比较可序列化类。

6) “import org.apache.hadoop.mapreduce.Mapper”

Mapper 类很重要，它将输入键值对映射到输出键值对，也就是 MapReduce 里的 Map 过程。

7)”public class TokenizerMapper extends Mapper<Object, Text, Text, IntWritable>”

定义一个自己的 Map 过程，类名是 TokenizerMapper，它继承了 Hadoop 的 Mapper 类。“<Object, Text, Text, IntWritable>”，这里，第一个参数类型是 Object，表示输入键 key 的参数类型，第二个参数参数类型是 Text，表示输入值的类型，第三个参数类型也是 Text，表示输出键类型，第四个参数类型是 IntWritable，表示输出值类型。

在这个例子里，第一个参数 Object 是 Hadoop 根据默认值生成的，一般是文件块里的一行文字的行偏移数，这些偏移数不重要，在处理时候一般用不上，第二个参数类型是要处理的字符串，形如”This is a cat.”。经过 Map 处理之后，输出的就是诸如<”This”, 1>的键值对，这个”This”就是第三个参数类型，是 Text 类型，而 1 就是第四个参数类型，是 IntWritable。

8)“IntWritable one = new IntWritable(1)”

定义输出值，始终是 1。

9) “Text word = new Text()”

定义输出键。

10) “public void map(Object key, Text value, Context context) throws IOException, InterruptedException ”

定义 map 函数，函数有三个参数，key 是输入键，它是什么无所谓，实际上用不到它的，value 是输入值。在 map 函数中，出错的时候会抛出异常，所以有“throws IOException, InterruptedException ”。至于 Context 类，这个类的定义是在 TokenizerMapper 的祖先类 Mapper 的内部，不需要引入，如果去查看 Mapper 类的源代码的话，能看到 Context 类是继承 MapContext 类的。

11) “StringTokenizer itr = new StringTokenizer(value.toString())”

定义 StringTokenizer 对象 itr， StringTokenizer 的构造函数只接受 Java 的 String 类，而 value 是 Text 类，所以要进行转化，将 value 转成 String 类，也就是“value.toString()”。

12)Map 过程

```
while (itr.hasMoreTokens()){
  word.set(itr.nextToken());
  context.write(word, one);
}
```

在默认的情况下， StringTokenizer 以空格符作为分隔符对字符串进行解析，每次解析会先调用 hasMoreTokens 看看是不是需要做解析，如果需要做，就用 nextToken()函数获取解析结果，然后用这个结果给 word 赋值，然后，再将 word 和 one 作为一个键值对写到 context 里，context 会存储键值留待 Reduce 过程处理。

## 3.3 IntSumReducer.java 文件的源代码

```
package com.brianchen.hadoop;

import java.io.IOException;

import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Reducer;

public class IntSumReducer extends
      Reducer<Text,IntWritable,Text,IntWritable> {
  IntWritable result = new IntWritable();

  public void reduce(Text key, Iterable<IntWritable> values, Context context)
      throws IOException, InterruptedException {
    int sum = 0;
    for (IntWritable val : values) {
      sum += val.get();
    }
    result.set(sum);
    context.write(key, result);
  }
}
```

跟上节相同的地方就不解释了，只解释上节没有的东东。

1)”import org.apache.hadoop.mapreduce.Reducer”

引入 hadoop 的 Reducer 类，这个类负责 MapReduce 的 Reduce 过程。

2) “public class IntSumReducer extends Reducer<Text,IntWritable,Text,IntWritable> “

定义 Reduce 过程，也就是 IntSumReducer 类，这个类继承 Hadoop 的 Reducer 类。这里的”<Text,IntWritable,Text,IntWritable>”，含义跟上一节一样，依次分别是输入键类型，输入值类型，输出键类型，输出值类型。

3)“IntWritable result = new IntWritable()”

定义输出结果，这是一个整数。

4) “public void reduce(Text key, Iterable<IntWritable> values, Context context) throws IOException, InterruptedException ”

定义 reduce 函数。key 是输入键类型，values 是一个实现了 Iterable 接口的变量，可以把它理解成 values 里包含若干个 IntWritable 整数，可以通过迭代的方式遍历所有的值，至于 Context 类型，跟 Mapper 里的 Context 类似的方式，是在 Redurer 类内部实现的。

举例来说，假如处理一个字符串 ”This is a That is a“，那么，经过 Map 过程之后，到达 reduce 函数的时候，依次传递给 reduce 函数的是： key=”This”， values=<1>； key = “is”， values=<1, 1>； key = “a”, values=<1, 1>； key=”That”, values=<1>。注意，在 key = “is”和 key=”a”的时候，values 里有两个 1。

5) Reduce 过程

```
int sum = 0;
for (IntWritable val : values) {
    sum += val.get();
}
```

```
    result.set(sum);
    context.write(key, result);
```

这个过程，就是用一个循环，不断从 values 里取值，然后累加计算和，循环结束后，将累加和赋值给 result 变量，然后，将键值和累加和作为一个键值对写入 context。继续以上一步的例子来说，写入 context 的键值对依次就是<”This”，1>，<“is”，2>，<“a”, 2>，<”That”, 1>。

## 3.4 WordCount.java 文件的源代码

```
package com.brianchen.hadoop;

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

public class WordCount {
  public static void main(String[] args) throws Exception {
    Configuration conf = new Configuration();
    String[] otherArgs = new GenericOptionsParser(conf, args).getRemainingArgs();
    if (otherArgs.length != 2) {
      System.err.println("Usage: wordcount <in> <out>");
      System.exit(2);
    }

    Job job = new Job(conf, "wordcount");
    job.setJarByClass(WordCount.class);
    job.setMapperClass(TokenizerMapper.class);
    job.setReducerClass(IntSumReducer.class);
    job.setOutputKeyClass(Text.class);
    job.setOutputValueClass(IntWritable.class);
    FileInputFormat.addInputPath(job, new Path(otherArgs[0]));
    FileOutputFormat.setOutputPath(job, new Path(otherArgs[1]));
    System.exit(job.waitForCompletion(true) ? 0 : 1);
  }
}
```

1)”import org.apache.hadoop.conf.Configuration”

Configuration 类，顾名思义，读写和保存各种配置资源。

2) “import org.apache.hadoop.fs.Path”

引入 Path 类，Path 类保存文件或者目录的路径字符串。

3) “import org.apache.hadoop.mapreduce.Job”

引入 Job 类。在 hadoop 里，每个需要执行的任务是一个 Job，这个 Job 负责很多事情，包括参数配置，设置

MapReduce 细节，提交到 Hadoop 集群，执行控制，查询执行状态，等等。

4)”import org.apache.hadoop.mapreduce.lib.input.FileInputFormat”

引入 FileInputFormat 类。这个类的很重要的作用就是将文件进行切分 split，因为只有切分才可以并行处理。这个会在后面章节有详细解释。

5) “import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat”

引入 FileOutputFormat 类，处理结果写入输出文件。

6) “import org.apache.hadoop.util.GenericOptionsParser”

引入 GenericOptionsParser 类，这个类负责解析 hadoop 的命令行参数。

7)”public class WordCount ”

这是 word count 主类，它负责读取命令行参数，配置 Job，调用 Mapper 和 Reducer，返回结果等等工作。

8)“Configuration conf = new Configuration()”

默认情况下，Configuration 开始实例化的时候，会从 Hadoop 的配置文件里读取参数。

9)”String[] otherArgs = new GenericOptionsParser(conf, args).getRemainingArgs()”

读取参数分两步，上一步是从 Hadoop 的配置文件读取参数，这一步是从命令行参数读取参数，args 是存放命令行参数的字符串数组。

10) “if (otherArgs.length != 2) ”

如果命令行参数不是 2 个，就出错了，退出。因为程序需要知道处理的是哪个输入文件，处理结果放到哪个目录，必须是两个参数。

11)” Job job = new Job(conf, "wordcount")”

每个运行的处理任务就是一个 Job，”worodcount”是 Job 的名字。

12) “ job.setJarByClass(WordCount.class)”

Jar 文件是 Java 语言的一个功能，可以将所有的类文件打包成一个 Jar 文件，setJarByClass 的意思是，根据 WordCount 类的位置设置 Jar 文件。

13) “job.setMapperClass(TokenizerMapper.class)”

设置 Mapper。

14) “job.setReducerClass(IntSumReducer.class)”

设置 Reducer。

15) “job.setOutputKeyClass(Text.class)”

设置输出键的类型。

16)“job.setOutputValueClass(IntWritable.class)”

设置输出值的类型。

17) “FileInputFormat.addInputPath(job, new Path(otherArgs[0]))”

设置要处理的文件，也就是输入文件，它是 otherArgs 的第一个参数。

18) “FileOutputFormat.setOutputPath(job, new Path(otherArgs[1]))”

设置输出文件，将处理结果写到这个文件里，它是 otherArgs 的第二个参数。

19) “System.exit(job.waitForCompletion(true) ? 0 : 1)”

最后一步，job 开始执行，等待执行结束。

## 3.5 编译

用 javac 编译项目。javac 即 Java programming language compiler，是 Java JDK 的命令行编译器。如前所说，wordcount_01 目录存放源代码和编译结果，要在这个目录下进行编译。

3.5.1 “cd ~/wordcount_01”

先执行这个命令，切换目录到 wordcount_01 下。

3.5.2 “javac -classpath /home/brian/usr/hadoop/hadoop-1.2.1/hadoop-core-1.2.1.jar:/home/brian/usr/hadoop/hadoop-1.2.1/lib/commons-cli-1.2.jar -d ./classes/ ./src/*.java”

执行这条命令，编译源代码。-classpath，设置源代码里使用的各种类的库文件路径，路径之间用":"隔开，-d 参数，设置编译后的 class 文件存在路径。

## 3.6 打包

3.6.1 "jar -cvf wordcount.jar -C ./classes/ ."

将编译好的 class 文件打包成 Jar 包，jar 命令是 JDK 的打包命令行工具，跟 tar 非常像。在命令里，-C 是值在执行 jar 的时候将目录切换到当前目录下的 classes 目录，这个目录包含编译好的 class 文件。打包结果是 wordcount.jar 文件，放在当前目录下。

## 3.7 执行

3.7.1 首先要确实一下 Hadoop 已经运行起来了。启动方式就是第 1 章的第 7 节。然后，执行

3.7.2 "cd ~/usr/hadoop/hadoop-1.2.1"

切换目录到 Hadoop 的安装目录下。

3.7.3 "./bin/hadoop fs -put README.txt readme.txt"

仍然用 README.txt 做测试，将它复制到 HDFS 上，更名为 readme.txt

3.7.4 "./bin/hadoop fs -rmr output"

处理结果要放在 HDFS 的 output 目录里，如果这个目录已经存在了，Hadoop 是不会运行的，会报错，先删除它。

3.7.5 "./bin/hadoop jar /home/brian/wordcount_01/wordcount.jar com.brianchen.hadoop.WordCount readme.txt output"

运行程序，处理 readme.txt 文件，将结果写入 output 目录，其中"jar"参数是指定 jar 包的位置，而"com.brianchen.hadoop.WordCount"，这里"com.brianchen.hadoop"是包的名字，"WordCount"是主类，注意，如果不写包名字会报错的，必须有包名。

## 3.8 查看结果

3.8.1 "./bin/hadoop fs -cat output/part-r-00000"

处理结果 output 目录的 part-r-00000 文件里，用 cat 命令可以输出到屏幕显示。

# 4. 最简形式的 MapReduce

最简单形式的 Word Count 的 MapReduce 代码是 Hadoop 自带的例子，略作改动放在这里。这个例子只有一个 Java 文件，Mapper 和 Reducer 都写在 WordCount 类的内部。

## 4.1 目录和文件结构

代码放在~/wordcount_02 目录，它有两个子目录，分别是 classes 和 src，classes 目录存放编译结果，src 目录存放源代码，src 目录下只有一个 java 文件，即”WordCount.java”，所有的代码都在里面。

## 4.2 WordCount.java 文件的源代码

```
package com.brianchen.hadoop;

import java.io.IOException;
import java.util.StringTokenizer;

import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.Text;
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 WordCount {
  public static class TokenizerMapper
       extends Mapper<Object, Text, Text, IntWritable>{
    private final static IntWritable one = new IntWritable(1);
    private Text word = new Text();

    public void map(Object key, Text value, Context context)
       throws IOException, InterruptedException {
      StringTokenizer itr = new StringTokenizer(value.toString());
      while (itr.hasMoreTokens()){
        word.set(itr.nextToken());
        context.write(word, one);
      }
    }
  }

 public static class IntSumReducer extends
       Reducer<Text,IntWritable,Text,IntWritable> {
    private IntWritable result = new IntWritable();
    public void reduce(Text key, Iterable<IntWritable> values, Context context)
```

```
      throws IOException, InterruptedException {
    int sum = 0;
    for (IntWritable val : values) {
      sum += val.get();
    }
    result.set(sum);
    context.write(key, result);
  }
 }

 public static void main(String[] args) throws Exception {
  Configuration conf = new Configuration();
  String[] otherArgs = new GenericOptionsParser(conf, args).getRemainingArgs();
  if (otherArgs.length != 2) {
    System.err.println("Usage: wordcount <in> <out>");
    System.exit(2);
  }

  Job job = new Job(conf, "word count");
  job.setJarByClass(WordCount.class);
  job.setMapperClass(TokenizerMapper.class);
  job.setCombinerClass(IntSumReducer.class);
  job.setReducerClass(IntSumReducer.class);
  job.setOutputKeyClass(Text.class);
  job.setOutputValueClass(IntWritable.class);
  FileInputFormat.addInputPath(job, new Path(otherArgs[0]));
  FileOutputFormat.setOutputPath(job, new Path(otherArgs[1]));
  System.exit(job.waitForCompletion(true) ? 0 : 1);
 }
}
```

这里的代码，跟前一节有点不太一样。

1)”public static class TokenizerMapper”

这表示 TokenizerMapper 类是 WordCount 类的内部静态类，这种方式可以将 TokenizerMapper 隐藏在 WordCount 类内部，且 TokenizerMapper 类不引用 WordCount 类的任何变量和函数。

2) “private final static IntWritable one”

跟上一节的定义相比，这里多了”private final static”，”private”表示这个变量是类的私有变量，“final”表示这变量只能在定义的时候被赋值一次，以后不可更改，”static”表示这是一个静态变量，独立于对象，被该类的所有实例共享，这种做法的好处是，one 这个值是私有的不可更改的仅仅只有一个，代码更可靠，更节省内存空间。

## 4.3 编译

4.3.1 “cd ~/wordcount_02”

4.3.2 “javac -classpath /home/brian/usr/hadoop/hadoop-1.2.1/hadoop-core-1.2.1.jar:/home/brian/usr/hadoop/hadoop-

1.2.1/lib/commons-cli-1.2.jar -d ./classes/ ./src/WordCount.java ”

## 4.4 打包

4.4.1 “jar -cvf wordcount.jar -C ./classes/ . ”

## 4.5 运行

4.5.1 “cd ~/usr/bin/hadoop/hadoop-1.2.1”
4.5.2 “./bin/hadoop fs -rmr output”
4.5.3 “./bin/hadoop jar /home/brian/wordcount_02/wordcount.jar com.brianchen.hadoop.WordCount readme.txt output”

## 4.6 查看结果

4.6.1 “./bin/hadoop fs -cat output/part-r-00000”

# 第 3 章 HDFS 分布式文件系统

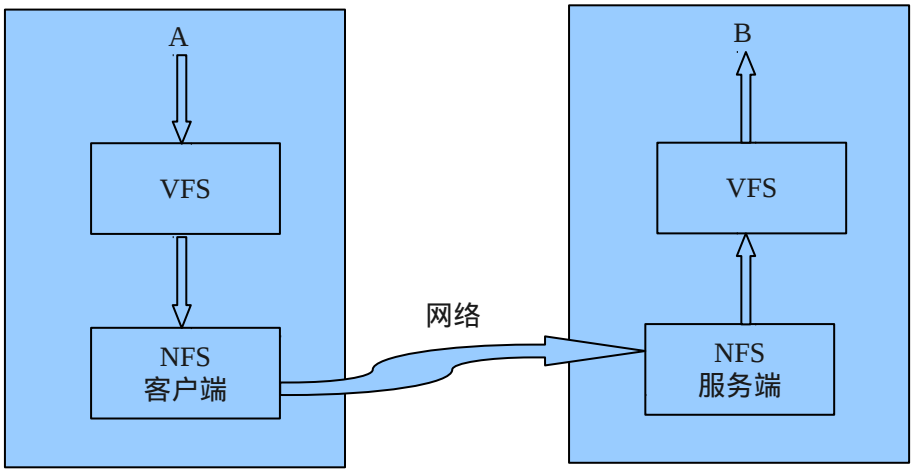
## 1. 文件系统从头说

文件系统的作用就是永久存储数据。计算机可以存储数据的地方是内存，硬盘，优盘，SD 卡等等。如果计算机断电关机，存放在内存里的数据就没有了，而存放在硬盘优盘 SD 卡这些上的数据会仍然存在。硬盘优盘 SD 卡上的数据是以文件的形式存在，文件系统就是文件的组织和处理。总之，凡是断电之后不会消失的数据，就必须由文件系统存储和管理。

从用户的角度来说，文件系统需要提供文件的创建，删除，读，写，追加，重命名，查看属性，更改属性等各种功能。文件夹，也叫目录，它的作用类似容器，保存其他文件夹和文件。于是，各级文件夹和各级文件就共同组成了文件系统的层次，看起来象一棵倒放的树，最上层是最大的目录，也叫根目录，然后这个目录包含子目录和文件，子目录又包含更多的子目录和文件，这棵树的术语叫目录树。

起初，Linux 使用的文件系统是 Minix 文件系统。但 Minix 系统有不少限制，诸如最大文件尺寸只有 64M，文件名最多是 14 个字符长度。后来，Linux 内核加入了 VFS，也就是虚拟文件系统 Virtual File System。VFS 是 Linux 内核和真正文件系统之间的抽象层，它提供统一的接口，真正的文件系统和 Linxu 内核必须通过 VFS 的接口进行沟通。随后，Linux 逐步使用基于 VFS 的 ext 文件系统，ext2 文件系统，ext3 文件系统等等。基于 VFS，Linux 对 Windows 的 FAT 和 NTFS 格式也提供支持。

通常情况下，Linux 的文件系统是单机的，也就说，从物理的角度看，文件系统只存储单台计算机的数据。分布式文件系统在物理上分散的计算机上存储数据。比如，NFS（NetWork File System）是一种非常经典的分布式文件系统，它基于 VFS，由 Sun 公司开发的。本质上，NFS 是在物理上分散的计算机之间增加了一个客户-服务器层。对 NFS，可以这么理解：计算机 A 有自己的 VFS，计算机 B 也有自己的 VFS，那么，如果 A 想操作 B 上的文件，A 的数据和命令依次通过的路线是：A 的 VFS-->A 的 NFS 客户端-->网络-->B 的 NFS 服务器端-->B 的 VFS-->B 的文件系统。



## 2. Hadoop 的文件系统

Hadoop 借鉴了 VFS，也引入了虚拟文件系统机制。HDFS 是 Hadoop 虚拟文件系统的一个具体实现。除了 HDFS 文件系统之外，Hadoop 还实现很多其他文件系统，诸如本地文件系统，支持 HTTP 的 HFTP 文件系统，支持 Amazon 的 S3 文件系统等等。

HDFS 从设计上来说，主要考虑以下的特征：超大文件，最大能支持 PB 级别的数据；流式数据访问，一次写入，多次读取；在不可靠的文件，故障率高的商用硬件上能运行。Hadoop 的不利之处，是不适应低时间延迟的数据访问，不适应大量的小文件，也不适应多用户写入任意修改文件的情况。

假设有一个 HDFS 集群，那么这个集群有且仅有一台计算机做名字节点 NameNode，有且仅有一台计算机做第二名字节点 SecondaryNameNode，其他机器都是数据节点 DataNode。在伪分布式的运行方式下，NameNode，SecodaryNameNode，DataNode 都由同一台机器担任。

NameNode 是 HDFS 的管理者。SecondaryNameNode 是 NameNode 的辅助者，帮助 NameNode 处理一些合并事宜，注意，它不是 NameNode 的热备份，它的功能跟 NameNode 是不同的。DataNode 以数据块的方式分散存储 HDFS 的文件。HDFS 将大文件分割成数据块，每个数据块是 64M，也可以设置成 128M 或者 256M，然后将这些数据块以普通文件的形式存放到数据节点上，为了防止 DataNode 意外失效，HDFS 会将每个数据块复制若干份放到不同的数据节点。

执行”hadoop fs -help”可以看到 HDFS 的命令行工具和用法。

如前所说，文件系统主要作用是提供文件的创建，删除，读，写，追加，重命名，查看属性，更改属性等各种功能。在随后部分，本章选取若干功能，给出了 HDFS 的文件操作示例代码。熟悉这些之后会对 HDFS 的操作有一个形象了解。这样将来参考 Hadoop API 的 FileSystem 类及其相关子类，就可以写出更多的文件系统操作。

## 3. 如何将文件复制到 HDFS

### 3.1 目录和文件结构

这个例子的功能跟”hadoop fs -put”是一样的。创建目录~/filecopy 存放源代码、编译和打包结果。在 filecopy 目录下，有两个子目录，分别是 src 目录和 classes 目录，src 目录存放 Java 源代码，class 存放编译结果。在 src 目录下，只有一个源代码文件 FileCopy.java。

### 3.2 FileCopy.java 文件的源代码

```
package com.brianchen.hadoop;

import java.net.URI;
import java.io.InputStream;
import java.io.OutputStream;
import java.io.BufferedInputStream;
import java.io.FileInputStream;

import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.FileSystem;
import org.apache.hadoop.fs.FSDataInputStream;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.IOUtils;

public class FileCopy{
  public static void main(String[] args) throws Exception{
    if (args.length != 2){
      System.err.println("Usage: filecopy <source> <target>");
      System.exit(2);
    }

    Configuration conf = new Configuration();
    InputStream in = new BufferedInputStream(new FileInputStream(args[0]));
    FileSystem fs = FileSystem.get(URI.create(args[1]), conf);
    OutputStream out = fs.create(new Path(args[1]));
    IOUtils.copyBytes(in, out, 4096, true);
  }
}
```

## 3.3 编译

“cd ~/filecopy”
“javac -cp /home/brian/usr/hadoop/hadoop-1.2.1/hadoop-core-1.2.1.jar -d ./classes ./src/*.java”

## 3.4 打包

“jar -cvf filecopy.jar -C ./classes/ .”

## 3.5 运行

“cd /home/brian/usr/hadoop/hadoop-1.2.1”

“./bin/hadoop jar ~/filecopy/filecopy.jar com.brianchen.hadoop.FileCopy README.txt readme.txt”

首先确认 Hadoop 已经是运行的，然后切换到 Hadoop 的安装目录，仍然用 README.txt 做测试，将这个文件复制到 HDFS，另存为 readme.txt 文件。

## 3.6 检查结果

“./bin/hadoop fs -ls”

执行这个命令可以看到 readme.txt 是否存在。

“./bin/hadoop fs -ls cat readme.txt”

输出 readme.txt 文件到屏幕查看其内容。

# 4. 从 HDFS 删除文件

## 4.1 目录和文件结构

这个例子的功能跟”hadoop fs -rm”是一样的。创建目录~/filedelete 存放源代码、编译和打包结果。在 filedelete 目录下，有两个子目录，分别是 src 目录和 classes 目录，src 目录存放 Java 源代码，class 存放编译结果。在 src 目录下，只有一个源代码文件 FileDelete.java。

## 4.2 FileDelete.java 文件的源代码

```
package com.brianchen.hadoop;

import java.net.URI;

import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.FileSystem;
import org.apache.hadoop.fs.Path;

public class FileDelete{
  public static void main(String[] args) throws Exception{
    if (args.length != 1){
      System.err.println("Usage: filedelete <target>");
      System.exit(2);
    }

    Configuration conf = new Configuration();
    FileSystem fs = FileSystem.get(URI.create(args[0]), conf);
    fs.delete(new Path(args[0]), false);
  }
}
```

## 4.3 编译

“cd ~/filedelete”
“javac -cp /home/brian/usr/hadoop/hadoop-1.2.1/hadoop-core-1.2.1.jar -d ./classes ./src/*.java”

## 4.4 打包

“jar -cvf filedelete.jar -C ./classes/ .”

## 4.5 运行

“cd /home/brian/usr/hadoop/hadoop-1.2.1”
“./bin/hadoop jar ~/filedelete/filedelete.jar com.brianchen.hadoop.FileDelete readme.txt”
首先确认 Hadoop 已经是运行的，然后切换到 Hadoop 的安装目录，删除 readme.txt 文件。

## 4.6 检查结果

“./bin/hadoop fs -ls”
执行这个命令可以看到 readme.txt 是否存在，如果执行正常，readme.txt 是不存在的。

# 5. 读取 HDFS 的文件

## 5.1 目录和文件结构

这个例子的功能跟”hadoop fs -cat”是一样的。创建目录~/filecat 存放源代码、编译和打包结果。在 filecat 目录下，有两个子目录，分别是 src 目录和 classes 目录，src 目录存放 Java 源代码，class 存放编译结果。在 src 目录下，只有一个源代码文件 FileCat.java。

## 5.2 FileCat.java 文件的源代码

```
package com.brianchen.hadoop;

import java.net.URI;
import java.io.InputStream;

import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.FileSystem;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.IOUtils;

public class FileCat{
  public static void main(String[] args) throws Exception{
    if (args.length != 1){
      System.err.println("Usage: filecat <source>");
      System.exit(2);
    }

    Configuration conf = new Configuration();
    FileSystem fs = FileSystem.get(URI.create(args[0]), conf);
    InputStream in = null;
    try{
      in = fs.open(new Path(args[0]));
      IOUtils.copyBytes(in, System.out, 4096, false);
    }finally{
      IOUtils.closeStream(in);
    }
  }
}
```

## 5.3 编译

“cd ~/filecat”
“javac -cp /home/brian/usr/hadoop/hadoop-1.2.1/hadoop-core-1.2.1.jar -d ./classes ./src/*.java”

## 5.4 打包

“jar -cvf filecat.jar -C ./classes/ .”

## 5.5 运行

“cd /home/brian/usr/hadoop/hadoop-1.2.1”
“./bin/hadoop fs -put README.txt readme.txt”
“./bin/hadoop jar ~/filecat/filecat.jar com.brianchen.hadoop.FileCat readme.txt”
首先确认 Hadoop 已经是运行的，然后切换到 Hadoop 的安装目录，将 readme.txt 文件内容输出到屏幕。

# 6. 获取文件的属性信息

## 6.1 目录和文件结构

这个例子的功能跟”hadoop fs -ls”类似。创建目录~/fileinfo 存放源代码、编译和打包结果。在 fileinfo 目录下，有两个子目录，分别是 src 目录和 classes 目录，src 目录存放 Java 源代码，class 存放编译结果。在 src 目录下，只有一个源代码文件 FileInfo.java。它获取和输出文件长度，块大小，备份，修改时间，所有者，权限等信息。

## 6.2 FileInfo.java 文件的源代码

```
package com.brianchen.hadoop;

import java.net.URI;

import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.FileSystem;
import org.apache.hadoop.fs.FileStatus;
import org.apache.hadoop.fs.Path;

public class FileInfo{
  public static void main(String[] args) throws Exception{
    if (args.length != 1){
      System.err.println("Usage: fileinfo <source>");
      System.exit(2);
    }

    Configuration conf = new Configuration();
    FileSystem fs = FileSystem.get(URI.create(args[0]),conf);
    FileStatus stat = fs.getFileStatus(new Path(args[0]));

    System.out.println(stat.getPath());
    System.out.println(stat.getLen());
    System.out.println(stat.getModificationTime());
    System.out.println(stat.getOwner());
    System.out.println(stat.getReplication());
    System.out.println(stat.getBlockSize());
    System.out.println(stat.getGroup());
    System.out.println(stat.getPermission().toString());
  }
}
```

## 6.3 编译

“cd ~/fileinfo”
“javac -cp /home/brian/usr/hadoop/hadoop-1.2.1/hadoop-core-1.2.1.jar -d ./classes ./src/*.java”

## 6.4 打包

“jar -cvf fileinfo.jar -C ./classes/ .”

## 6.5 运行

“cd /home/brian/usr/hadoop/hadoop-1.2.1”
“./bin/hadoop fs -put README.txt readme.txt”
“./bin/hadoop jar ~/fileinfo/fileinfo.jar com.brianchen.hadoop.FileInfo readme.txt”
首先确认 Hadoop 已经是运行的，然后切换到 Hadoop 的安装目录，将 readme.txt 文件的属性信息输出到屏幕。

# 7. 列出目录下所有文件

## 7.1 目录和文件结构

创建目录~/filelist 存放源代码、编译和打包结果。在 fileinfo 目录下，有两个子目录，分别是 src 目录和 classes 目录，src 目录存放 Java 源代码，class 存放编译结果。在 src 目录下，只有一个源代码文件 FileList.java。

## 7.2 FileList.java 文件的源代码

```
package com.brianchen.hadoop;

import java.net.URI;

import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.FileSystem;
import org.apache.hadoop.fs.FileStatus;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.fs.FileUtil;

public class FileList{
  public static void main(String[] args) throws Exception{
    if (args.length != 1){
      System.err.println("Usage: filelist <source>");
      System.exit(2);
    }

    Configuration conf = new Configuration();
    FileSystem fs = FileSystem.get(URI.create(args[0]),conf);
    FileStatus[] status = fs.listStatus(new Path(args[0]));
    Path[] listedPaths = FileUtil.stat2Paths(status);

    for(Path p : listedPaths){
      System.out.println(p);
    }
  }
}
```

## 7.3 编译

“cd ~/filelist”
“javac -cp /home/brian/usr/hadoop/hadoop-1.2.1/hadoop-core-1.2.1.jar -d ./classes ./src/*.java”

## 7.4 打包

“jar -cvf filelist.jar -C ./classes/ .”

## 7.5 运行

“cd /home/brian/usr/hadoop/hadoop-1.2.1”

“./bin/hadoop fs -put README.txt readme.txt”

“./bin/hadoop fs -put README.txt readme2.txt”

“./bin/hadoop jar ~/filelist/filelist.jar com.brianchen.hadoop.FileList hdfs://localhost:9000/user/brian”

首先确认 Hadoop 已经是运行的，然后切换到 Hadoop 的安装目录，然后列出/user/brian 下的文件，有两个，分别是 readme.txt 和 readme2.txt。

# 第 4 章 序列化

## 1. 序列化从头说

在面向对象程序设计中，类是个很重要的概念。所谓“类”，可以将它想像成建筑图纸，而对象就是根据图纸盖的大楼。类，规定了对象的一切。根据建筑图纸造房子，盖出来的就是大楼，等同于将类进行实例化，得到的就是对象。

一开始，在源代码里，类的定义是明确的，但对象的行为有些地方是明确的，有些地方是不明确的。对象里不明确地方，是因为对象在运行的时候，需要处理无法预测的事情，诸如用户点了下屏幕，用户点了下按钮，输入点东西，或者需要从网络发送接收数据之类的。后来，引入了泛型的概念之后，类也开始不明确了，如果使用了泛型，直到程序运行的时候，才知道究竟是哪种对象需要处理。

对象可以很复杂，也可以跟时序相关。一般来说，“活的”对象只生存在内存里，关机断电就没有了。一般来说，“活的”对象只能由本地的进程使用，不能被发送到网络上的另外一台计算机。

序列化，可以存储“活的”对象，可以将“活的”对象发送到远程计算机。

把“活的”对象序列化，就是把“活的”对象转化成一串字节，而“反序列化”，就是从一串字节里解析出“活的”对象。于是，如果想把“活的”对象存储到文件，存储这串字节即可，如果想把“活的”对象发送到远程主机，发送这串字节即可，需要对象的时候，做一下反序列化，就能将对象“复活”了。

将对象序列化存储到文件，术语又叫“持久化”。将对象序列化发送到远程计算机，术语又叫“数据通信”。

Java 对序列化提供了非常方便的支持，在定义类的时候，如果想让对象可以被序列化，只要在类的定义上加上了”implements Serializable”即可，比如说，可以这么定义”public class Building implements Serializable”，其他什么都不要做，Java 会自动的处理相关一切。Java 的序列化机制相当复杂，能处理各种对象关系。

Java 的序列化机制的缺点就是计算量开销大，且序列化的结果体积大太，有时能达到对象大小的数倍乃至十倍。它的引用机制也会导致大文件不能分割的问题。这些缺点使得 Java 的序列化机制对 Hadoop 来说是不合适的。于是 Hadoop 设计了自己的序列化机制。

为什么序列化对 Hadoop 很重要？因为 Hadoop 在集群之间进行通讯或者 RPC 调用的时候，需要序列化，而且要求序列化要快，且体积要小，占用带宽要小。所以必须理解 Hadoop 的序列化机制。

## 2. Hadoop 的序列化接口

什么是接口？简答来说，接口就是规定，它规定类必须实现的方法。一个接口可以包含多干个方法。如果一个类说

自己实现了某个接口，那么它必须实现这个接口里的所有方法。特殊情况下，接口也可以没有任何方法。

Writable 接口，也就是 org.apache.hadoop.io.Writable 接口。Hadoop 的所有可序列化对象都必须实现这个接口 。Writable 接口里有两个方法，一个是 write 方法，将对象写入字节流，另一个是 readFields 方法，从字节流解析出对象。

Java 的 API 提供了 Comparable 接口，也就是 java.lang.Comparable 接口。这个接口只有一个方法，就是 compareTo，用于比较两个对象。

WritableComparable 接口同时继承了 Writable 和 Comparable 这两个接口。

Hadoop 里的三个类 IntWritable、DoubleWritable 和 ByteWritable，都继承了 WritableComparable 接口。注意，IntWritable、DoubleWritable 和 ByteWritable，尽管后缀是“Writable”，但它们不是接口，是类！！

Hadoop 的序列化接口还有更多的类型，在这里不一一列举。

# 3. IntWritable 如何序列化

## 3.1 目录和文件结构

这个例子演示 IntWritable 如何序列化。首先，创建一个 IntWritable，然后，将它序列化，输出到一个字节流中。然后，创建一个新的 IntWritable，从字节流中读取值，这是反序列化。

创建目录~/intser 存放源代码、编译和打包结果。在 intser 目录下，有两个子目录，分别是 src 目录和 classes 目录，src 目录存放 Java 源代码，class 存放编译结果。在 src 目录下，只有一个源代码文件 IntSer.java。

## 3.2 IntSer.jav 文件的源代码

```
package com.brianchen.hadoop;

import java.io.DataOutputStream;
import java.io.DataInputStream;
import java.io.ByteArrayOutputStream;
import java.io.ByteArrayInputStream;
import java.io.IOException;

import org.apache.hadoop.io.Writable;
import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.util.StringUtils;

public class IntSer{
  public byte[] serialize(Writable w)throws IOException{
    ByteArrayOutputStream out = new ByteArrayOutputStream();
    DataOutputStream dataout = new DataOutputStream(out);
    w.write(dataout);
    dataout.close();
    return out.toByteArray();
  }

  public byte[] deserialize(Writable w, byte[] bytes) throws IOException{
    ByteArrayInputStream in = new ByteArrayInputStream(bytes);
    DataInputStream datain = new DataInputStream(in);
    w.readFields(datain);
    datain.close();
    return bytes;
  }

  public static void main(String[] args) throws Exception{
    IntWritable intw = new IntWritable(7);
    byte[] bytes = serialize(intw);
    String bytes_str = StringUtils.byteToHexString(bytes);
    System.out.println(bytes_str);

    IntWritable intw2 = new IntWritable(0);
    deserialize(intw2, bytes);
    System.out.println(intw2);
  }
}
```

## 3.3 编译

“cd ~/intser”
“javac -cp /home/brian/usr/hadoop/hadoop-1.2.1/hadoop-core-1.2.1.jar -d ./classes src/*.java”

## 3.4 打包

“jar -cvf intser.jar -C ./classes .”

## 3.5 运行

“cd ~/usr/hadoop/hadoop-1.2.1”
“./bin/hadoop jar /home/brian/intser/intser.jar com.brianchen.hadoop.IntSer”
首先确认 Hadoop 已经是运行的，然后切换到 Hadoop 的安装目录，然后运行，输出结果是两行，第一行是”00000007”，第二行是”7”。

# 第 5 章 压缩

## 1. 压缩从头说

一般来说，数据存在冗余度。数据包括图像文本视频音频。减少数据的冗余度，让数据的体积更小一点，这叫压缩从压缩后的数据，重新解析出原始数据，叫解压缩。

压缩无处不在。压缩的算法非常多。

对 Hadoop 来说，有两个地方需要用到压缩：其一，在 HDFS 上存储数据文件，压缩之后数据体积更小，有利存储；其二，集群间的通讯需要压缩数据，这样可以提高网络带宽的利用率。如果用 MapReduce 处理压缩文件，那么要求压缩算法能支持文件分割，因为 MapReduce 的过程需要将文件分割成多个切片，如果压缩算法不支持分割，就不能做切片了。

在 Java 里，一切输入输出都用流的方式进行。一个可以读取字节序列的对象叫输入流。文件，网络连接，内存区域，都可以是输入流。一个可以写入字节序列的对象叫输出流。文件，网络连接，内存区域，都可以是输出流。

Hadoop 如何压缩？假设，输入流是 A，输出流是 B。A 和 B 有很多种可能，可以是文件，网络连接，内存区域，标准输入，标准输出的两两组合。做压缩的话，先选择压缩算法，然后根据压缩算法创建相应的压缩器，然后用压缩器和输出流 B 创建压缩输出流 C，最后，将数据从输入流 A 复制到压缩输出流 C 即可进行压缩并输出结果。

如果是解压缩，先选择解压缩算法，然后根据解压缩算法创建相应的解压缩器，然后用解压缩器和输入流 A 创建压缩输入流 C，最后，将数据从输入流 C 复制到输出流 B 即可进行解压缩并输出结果。

## 2. 从文件到文件的压缩

## 2.1 文件和目录结构

这个程序将 HDFS 上的一个文本文件压缩到另外一个文件。

创建目录~/cprsf2f 存放源代码、编译和打包结果。在 cprsf2f 目录下，有两个子目录，分别是 src 目录和 classes 目录，src 目录存放 Java 源代码，class 存放编译结果。在 src 目录下，只有一个源代码文件 CprsF2F.java。

## 2.2 CprsF2F.java 源文件的代码

```
package com.brianchen.hadoop;

import java.net.URI;
import java.io.InputStream;
import java.io.OutputStream;

import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.io.compress.CompressionCodec;
import org.apache.hadoop.io.IOUtils;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.fs.FileSystem;
import org.apache.hadoop.util.ReflectionUtils;

public class CprsF2F{
  public static void main(String[] args) throws Exception{
    if (args.length != 3){
      System.err.println("Usage: CprsF2F cmps_name src target");
      System.exit(2);
    }

    Class<?> codecClass = Class.forName(args[0]);
    Configuration conf = new Configuration();
        CompressionCodec codec = (CompressionCodec)ReflectionUtils.newInstance(codecClass,
conf);
    InputStream in = null;
    OutputStream out = null;
    FileSystem fs = FileSystem.get(URI.create(args[1]), conf);

    try{
      in = fs.open(new Path(args[1]));
      out = codec.createOutputStream(fs.create(new Path(args[2])));
      IOUtils.copyBytes(in, out, conf);
    }finally{
      IOUtils.closeStream(in);
      IOUtils.closeStream(out);
    }
  }
}
```

## 2.3 编译

“cd ~/cprsf2f”
“jvac -cp /home/brian/usr/hadoop/hadoop-1.2.1/hadoop-core-1.2.1.jar -d ./class/ src/*.java”

## 2.4 打包

“jar -cvf cprsf2f.jar -C ./class/ .”

## 2.5 执行

“cd ~/usr/hadoop/hadoop-1.2.1”
“echo "hellowrod" >> a.txt”
“./bin/hadoop fs -put a.txt a.txt”
“ ./bin/hadoop jar /home/brian/cprsf2f/cprsf2f.jar com.brianchen.hadoop.CprsF2F org.apache.hadoop.io.compress.GzipCodec a.txt b.txt”
“./bin/hadoop fs -cat b.txt”
“./bin/hadoop fs -cat b.txt|gunzip”
首先需要确认 Hadoop 已经启动。用 echo 命令创建一个内容是”hello world”的文件 a.txt。将 a.txt 文件复制到 HDFS。执行 jar，将文件 a.txt 压缩成 b.txt。压缩完毕之后，执行 cat，检查 b.txt 内容，这时候显示的是乱码，因为原始内容已经被压缩了。然后再执行“./bin/hadoop fs -cat b.txt | gunzip”，这次会显示出”hello world”，因为管道命令 gunzip 会将压缩文件的内容进行解压缩然后输出。

# 3. 从标准输入到文件的压缩

## 3.1 文件和目录结构

这个程序从标准输入读取字符串，然后讲它压缩到 HDFS 的文件存储。创建目录~/cprsin2f 存放源代码、编译和打包结果。在 cprsin2f 目录下，有两个子目录，分别是 src 目录和 classes 目录，src 目录存放 Java 源代码，class 存放编译结果。在 src 目录下，只有一个源代码文件 CprsIn2F.java。

## 3.2 CprsIn2F.java 源文件的代码

```
package com.brianchen.hadoop;

import java.net.URI;
import java.io.OutputStream;

import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.io.compress.CompressionCodec;
import org.apache.hadoop.io.IOUtils;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.fs.FileSystem;
import org.apache.hadoop.util.ReflectionUtils;

public class CprsIn2F{
  public static void main(String[] args) throws Exception{
    if (args.length != 2){
      System.err.println("Usage: CprsIn2F cmps_name target");
      System.exit(2);
    }

    Class<?> codecClass = Class.forName(args[0]);
    Configuration conf = new Configuration();
        CompressionCodec  codec  =  (CompressionCodec)ReflectionUtils.newInstance(codecClass,
conf);
    OutputStream out = null;
    FileSystem fs = FileSystem.get(URI.create(args[1]), conf);
    try{
      out = codec.createOutputStream(fs.create(new Path(args[1])));
      IOUtils.copyBytes(System.in, out, 4096, false);
    }finally{
      IOUtils.closeStream(out);
    }
  }
}
```

## 3.3 编译

“cd ~/cprsin2f”
“jvac -cp /home/brian/usr/hadoop/hadoop-1.2.1/hadoop-core-1.2.1.jar -d ./class/ src/*.java”

## 3.4 打包

“jar -cvf cprsf2f.jar -C ./class/ .”

## 3.5 执行

“cd ~/usr/hadoop/hadoop-1.2.1”

“ echo “hello, world!” | ./bin/hadoop jar /home/brian/cprsf2f/cprsf2f.jar com.brianchen.hadoop.CprsF2F org.apache.hadoop.io.compress.GzipCodec hello.txt”

“./bin/hadoop fs -cat hello.txt”

“./bin/hadoop fs -cat hello.txt|gunzip”

首先需要确认 Hadoop 已经启动。注意，第二行的命令的双引号需要写上去，这是表示输出字符串。用 echo 命令将”hello world”输出到标准输入，然后用管道的方式执行 jar，将标准输入的字符串压缩到 HDFS 的 hello.txt。“org.apache.hadoop.io.compress.GzipCodec”，这个是 Hadoop 的 Gzip 压缩器类的类名。压缩完毕之后，执行 cat，检查 hello.txt 内容，这时候显示的是乱码，因为原始内容已经被压缩了。然后再执行“./bin/hadoop fs -cat hello.txt | gunzip”，这次会显示出”hello, world!”，因为管道命令 gunzip 会将压缩文件的内容进行解压缩然后输出。

# 4 从文件到文件的解压缩

## 4.1 文件和目录结构

这个程序把压缩文件压解缩到另外一个文件。创建目录~/dcprsf2f 存放源代码、编译和打包结果。在 dcprsf2f 目录下，有两个子目录，分别是 src 目录和 classes 目录，src 目录存放 Java 源代码，class 存放编译结果。在 src 目录下，只有一个源代码文件 DcprsF2F.java。

## 4.2 DcprsF2F.java 源文件的代码

```
package com.brianchen.hadoop;

import java.net.URI;
import java.io.InputStream;
import java.io.OutputStream;

import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.io.compress.CompressionCodec;
import org.apache.hadoop.io.IOUtils;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.fs.FileSystem;
import org.apache.hadoop.util.ReflectionUtils;

public class DcprsF2F{
  public static void main(String[] args) throws Exception{
    if (args.length != 3){
      System.err.println("Usage: CprsF2F cmps_name src target");
      System.exit(2);
    }

    Class<?> codecClass = Class.forName(args[0]);
    Configuration conf = new Configuration();
    CompressionCodec codec = (CompressionCodec)ReflectionUtils.newInstance(codecClass, conf);
    InputStream in = null;
    OutputStream out = null;
    FileSystem fs = FileSystem.get(URI.create(args[1]), conf);
    try{
      in = codec.createInputStream(fs.open(new Path(args[1])), codec.createDecompressor());
      out = fs.create(new Path(args[2]));
      IOUtils.copyBytes(in, out, conf);
    }finally{
      IOUtils.closeStream(in);
      IOUtils.closeStream(out);
    }
  }
}
```

## 4.3 编译

“cd ~/dcprsf2f”

“jvac -cp /home/brian/usr/hadoop/hadoop-1.2.1/hadoop-core-1.2.1.jar -d ./class/ src/*.java”

## 4.4 打包

“jar -cvf dcprsf2f.jar -C ./class/ .”

## 4.5 执行

“cd ~/usr/hadoop/hadoop-1.2.1”

“ ./bin/hadoop jar /home/brian/cprsf2f/cprsf2f.jar com.brianchen.hadoop.DcprsF2F org.apache.hadoop.io.compress.GzipCodec hello.txt hello_dec.txt”

“./bin/hadoop fs -cat hello.txt”

“./bin/hadoop fs -cat hello_dec.txt”

首先需要确认 Hadoop 已经启动。压缩文件 hello.txt 是 5.3 节创建的，在这里直接使用。将压缩文件 hello.txt 解压缩到文件 hello_c.txt。“org.apache.hadoop.io.compress.GzipCodec”，这个是 Hadoop 的 Gzip 压缩器类的类名。压缩完毕之后，执行 cat，检查 hello.txt 内容，这时候显示的是乱码。然后再执行“./bin/hadoop fs -cat hello_dec.txt”，这次会显示出”hello, world!”。

# 第 6 章　MapReduce 的输入输出

这一章都是文字叙述，不需要写源代码了。一般情况下，只需要记住这些东西就可以了。

Hadoop 处理大数据。

大数据以文件的形式存储在 HDFS。

大文件被划分成文件块存贮，每个文件块有固定的大小，通常是 64M，或者 128M，或者 255M。

我们在第 2 章写了一个 Word Count 的 MapReduce 程序，最关键部分是 Mapper 和 Reducer。在做 MapReuce 时，先做 Map，再做 Reduce。Hadoop 的框架让我们不需要关注 Mapper 和 Reducer 之外的地方，但我们肯定会想，数据是怎么从 HDFS 传给 Mapper 的？Reducer 处理完数据之后，又是怎么将数据存储到 HDFS 的？

将数据从 HDFS 传到 Mapper 是由 InputFormat 类实现的。

将数据从 Reducer 存储到 HDFS 是由 OutputFormat 类实现的。

先解释一下 Java 的抽象类。抽象类就是声明了抽象函数的类。假如 A 是一个抽象类，它有一个抽象函数 do_it，那么它只是声明有 do_it 这个函数，但并没有实现它的功能。抽象类只能作为类的模板，也就是说抽象类不能实例化。假设，类 B 继承了类 A，B 也就有了 dot_it 函数，如果 B 是一个具体的类，它就必须在自己的内部实现 do_it 函数的功能。

## 1. InputFormat 类

InputFormat 类是一个抽象类。InputFormat 类定义了两个抽象函数。这两个抽象函数是：

“abstract  List<InputSplit>  getSplits(JobContext context) ”

“abstract  RecordReader<K,V>  createRecordReader(InputSplit split, TaskAttemptContext context) ”

如果检查一下源代码， InputFormat 抽象类的定义是在 InputFormat.java，代码非常短，注释之外只有这两行抽象函数声明。

函数 getSplits 的功能，是将输入的 HDFS 文件切分成若干个 spit。在 Hadoop 集群里的每个节点做 MapReduce 时候处理的时候，每次只处理一个 split，所以 split 是 MapReduce 处理的最小单元。

函数 createRecordReader 的功能，是创建 RecordReader 对象，这个 RecordReader 对象根据 split 的内容，将 split 解析成若干个键值对。在做 MapReduce 的时候，Mappper 会不断地调用 RecordReader 的功能，从 RecordReader 里读取键值对，然后用 map 函数进行处理。

InputFormat 类是一个抽象类，它并不具体负责这两个功能的实现。它有 3 个继承类，DBInputFormat 类，DelegatingInputFormat 类和 FileInputFormat 类。其中，DBInputFormat 类是处理从数据库输入，DelegatingInputFormat 类是用在多个输入处理，FileInputFormat 类是处理基于文件的输入。

以 fileInputFormat 类为例。FileInputFormat 类是一个抽象类，它在 InputFormat 类的基础上，增加一些跟文件操作相关的函数。它实现了 getSplits 函数，但没实现 createRecordReader 函数，它把 createRecordReader 的实现留给继承类去做。在 getSplits 函数里，最重要的是这段：

```
// generate splits
//这是返回值
List<InputSplit> splits = new ArrayList<InputSplit>();
//获取 HDFS 文件的信息，FileStatus 在前面的章节使用过。
List<FileStatus>files = listStatus(job);
//对作业的每个文件都进行处理
for (FileStatus file: files) {
  //获取文件路径
  Path path = file.getPath();
  FileSystem fs = path.getFileSystem(job.getConfiguration());
  //获取文件长度
  long length = file.getLen();
  //获取文件在 HDFS 上存储的文件块的位置信息
  BlockLocation[] blkLocations = fs.getFileBlockLocations(file, 0, length);

  if ((length != 0) && isSplitable(job, path)) {
    //获取文件块的大小
    long blockSize = file.getBlockSize();
    //根据文件块大小，最小尺寸，最大尺寸，计算出 split 的大小
    long splitSize = computeSplitSize(blockSize, minSize, maxSize);

    //这段代码是根据 splitSize，每次计算一个 split 的块位置和所在主机的位置。
    //然后生成 split 对象存储。
    long bytesRemaining = length;
    while (((double) bytesRemaining)/splitSize > SPLIT_SLOP) {
      int blkIndex = getBlockIndex(blkLocations, length-bytesRemaining);
      splits.add(new FileSplit(path, length-bytesRemaining, splitSize,
                               blkLocations[blkIndex].getHosts()));
      bytesRemaining -= splitSize;
    }

    //最后剩下的不够一个 splitSize 的数据单独做一个 split。
    if (bytesRemaining != 0) {
      splits.add(new FileSplit(path, length-bytesRemaining, bytesRemaining,
                 blkLocations[blkLocations.length-1].getHosts()));
    }
  } else if (length != 0) {
    //如果文件很小，就直接做成一个 split
    splits.add(new FileSplit(path, 0, length, blkLocations[0].getHosts()));
  } else {
    //如果文件尺寸是 0，空文件，就创建一个空主机，主要是为了形式上一致。
    splits.add(new FileSplit(path, 0, length, new String[0]));
  }
}
```

FileInputFormat 有 5 个继承类，包括 CombineFileInputFormat 类， KeyValueTextInputFormat 类，

NLineInputFormat 类，SequenceFileInputFormat 类和 TextInputFormat 类。这几个有抽象类，也有具体类。

以 TextInputFormat 类为例，它实现了 createRecordReader 函数，非常简单，函数体只有一个语句，返回一个 LineRecordReader。

LineRecordReader 类继承了抽象类 RecordReader。抽象类 RecordReader 定义的全是抽象函数。LineRecordReader 每次从一个 InputSplit 里读取一行文本，以这行文本在文件中的偏移量为键，以这行文本为值，组成一个键值对，返回给 Mapper 处理。

## 2. OutputFormat 类

OutputFormat 类将键值对写入存储结构。一般来说，Mapper 类和 Reducer 类都会用到 OutputFormat 类。Mapper 类用它存储中间结果，Reducer 类用它存储最终结果。

OutputFormat 是个抽象类，这个类声明了 3 个抽象函数：

```
public abstract RecordWriter<K, V>     getRecordWriter(TaskAttemptContext context)
public abstract void checkOutputSpecs(JobContext context)
public abstract OutputCommitter getOutputCommitter(TaskAttemptContext context)
```

其中，最主要的函数是 getRecordWriter 返回 RecordWriter，它负责将键值对写入存储部件。函数 checkOutputSpecs 检查输出参数是否合理，一般是检查输出目录是否存在，如果已经存在就报错。函数 getOutputCommitter 获取 OutputCommitter， OutputCommitter 类是负责做杂活的，诸如初始化临时文件，作业完成后清理临时目录临时文件，处理作业的临时目录临时文件等等。

OutputFormat 类 4 个继承类，有 DBOutputFormat，FileOutputFormat，FilterOutputFormat，NullOutputFormat。顾名思义，DBOutputFormat 是将键值对写入到数据库， FileOutputFormat 将键值对写到文件系统， FilterOutputFormat 将其实是提供一种将 OutputFormat 进行再次封装，类似 Java 的流的 Filter 方式， NullOutputFormat 将键值对写入/dev/null，相当于舍弃这些值。

以 FileOutputFormat 为例。FileOutputFormat 是一个抽象类。它有两个继承类，SequenceFileOutputFormat 和 TextOutputFormat。SequenceFileOutputFormat 将键值对写入 HDFS 的顺序文件。TextOutputFormat 将数据写入 HDFS 的文本文件。写入过程类似第 3 章 HDFS 文件操作。

# 第 7 章　Hadoop MapReduce 的运行机制

这一章不会有源代码，主要是分析 Hadoop 的运行机制。

1. 一个 Hadoop 集群，有 5 种节点。
   1.1 NameNode，有且仅有一个，负责管理 HDFS 文件系统。
   1.2 DataNode，至少有一个，通常有很多个。具体地说，每台负责做集群计算的的计算机都是一个 DataNode。
   1.3 SecondaryNameNode，有且仅有一个，只辅助处理 NameNode，不做其他任务。
   1.4 JobTracker，有且仅有一个，负责 Hadoop 的作业管理，所有的 MapReduce 的执行由它处理。
   1.5 TaskTracker，至少有一个，通常有很多个。具体地说，每台负责做集群计算的的计算机都是一个 TaskTracker。

2. 伪分布式 Hadoop 集群，这个集群只有一台计算机，这台计算机上 5 种节点都有，启动 Hadoop 之后，用”jps”命令可以看到这 5 个节点进程。

3. 分布式 Hadoop 集群，如果集群内计算机足够多，会有一台机器做 NameNode，一台机器做 SecondaryNameNode，一台机器做 JobTracker。其他机器负责做集群计算，这些机器每台上面各有一个 DataNode 和 TaskNode 进程。

4. Hadoop 配置好之后，第一个命令是执行”./bin/hadoop namenode -format”。这个命令会对 NameNode 进行格式化，然后再关闭 NameNode。

5. 执行完第 4 步之后，用”./bin/start-all.sh”启动 Hadoop 集群。

6. 前处理
   6.1 写 MapReduce 代码，编译，将代码打成 tar 包。
   6.2 在 Linux 命令终端，运行 HDFS 的命令，将待处理的数据文件放到 HDFS 上。这时候，NameNode 节点会负责将文件分割成块，分散存储在各个 DataNode 节点上。
   6.3 在 Hadoop 集群运行的时候，每隔一定时间，SecondaryNameNode 会将 NameNode 上的数据进行合并处理。

7. 在一个 Linux 终端执行 Hadoop 命令，如"./bin/hadoop jar hadoop-examples-1.2.1.jar wordcount readme.txt output"。在 Hadoop 运行这个命令的时候，这个 Linux 终端就是一个”客户端”，它发的这个命令就是在“提交 MapReduce 作业”。

8. 客户端在做什么
   8.1 客户端向 JobTracker 节点申请一个作业 ID。
   8.2 检查作业的输出目录。如果目录没指定或者已存在，客户端会打印错误信息，然后退出。
   8.3 根据 InputFormat，将输入的数据格式进行分割，也就是分割成若干个 InputSplit。如果出错会退出。
   8.4 将 Mapreduce 程序的 jar 包，配置文件，InputSplit 信息都复制到 HDFS 的一个目录，这个目录名是作业 ID。
   8.5 通知 JobTracker 执行作业。至此，提交作业完毕。

8.6 客户端提交了作业之后，每秒会查询一次作业执行到了什么状态，如果有变化，就在冲端上打印出来，以便大家看到进度。

9. JobTracker 在做什么

9.1 客户端告诉 JobTracker 去执行作业。

9.2 JobTracker 将作业放到一个队列。

9.3 作业调度器对作业进行初始化，它取出 InputSplit 信息，为每个 InputSplit 创建一个 Map 任务，根据 map.reduce.reduce 参数创建若干个 Reduce 任务。

9.4 Hadoop 集群启动后，JobTracker 从始至终一直接收所有 TaskTracker 发送过来的心跳。如果某个心跳附带的信息说这个 TaskTracker 是空闲的，就发一个新任务给这个 TaskTracker 去做。

9.5 JobTracker 根据心跳，将作业和各种任务的执行进度计算出来。如果客户端查询，就将这些信息返回给它。

9.6 JobTracker 根据心跳，处理各种执行失败问题。

10. TaskTracker 在做什么

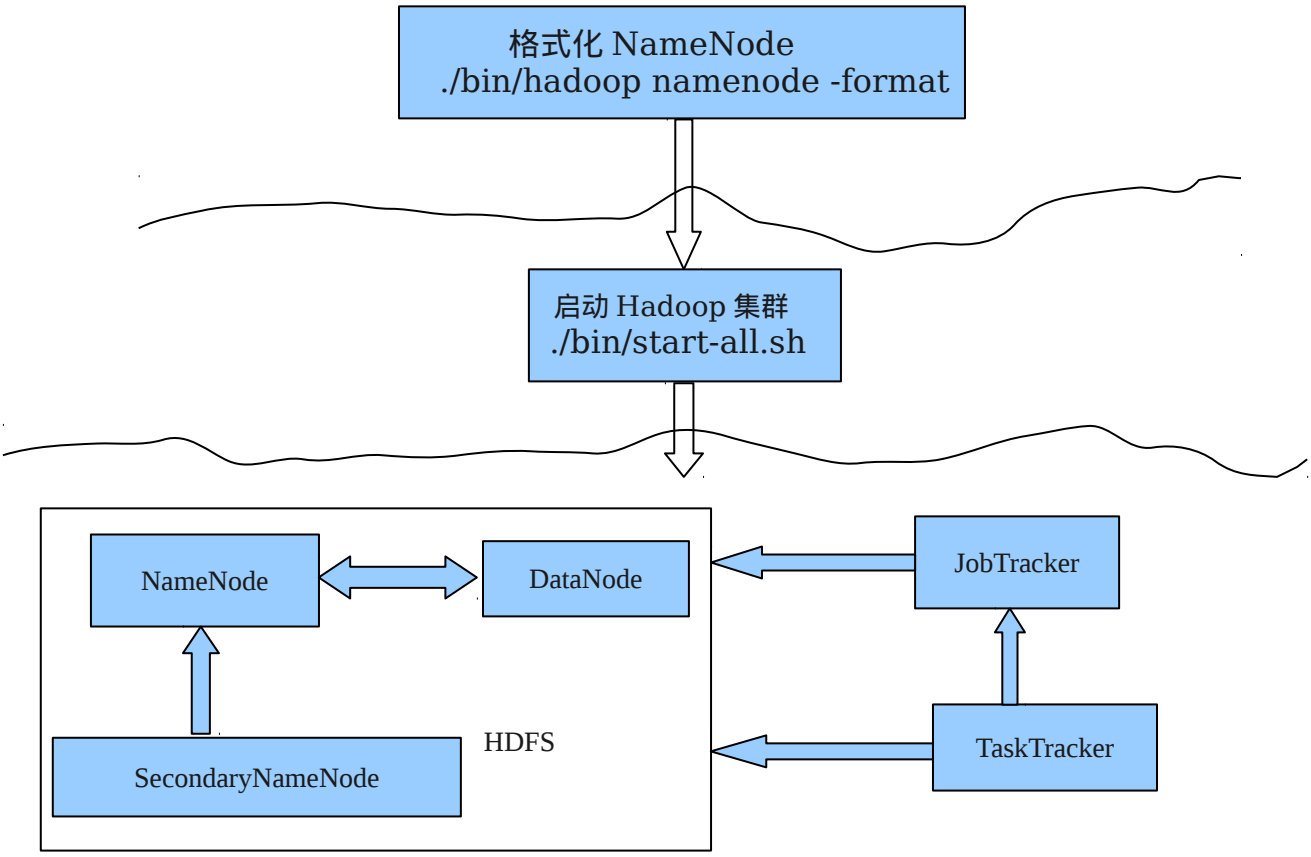
10.1 在 Hadoop 集群启动后，TaskTracker 定期向 JobTracker 发送“心跳”。心跳附带的信息会告诉 JobTracer，现在 TaskTracker 的状态是什么，是否可以运行新任务，是否在忙碌状态。

10.2 如果 TaskTracker 可以运行新任务，那么，JobTracer 会在心跳的返回值里给 TaskTracker 分配一个新任务。

10.3 TaskTracker 接到新任务，将这个任务所在的作业的 jar 文件以及所有需要处理的数据文件从 JobTracker 和 HDFS 复制到本地磁盘上。

10.4 TaskTracker 创建一个 TasckRunner 实例运行这个任务。每隔一段是时间，将任务执行的进度通过心跳发送给 JobTracker。

10.5 任务运行的时候，处理各种失败问题。


格式化 NameNode
./bin/hadoop namenode -format
启动 Hadoop 集群
./bin/start-all.sh
NameNode
DataNode
JobTracker
HDFS
SecondaryNameNode
TaskTracker

# 第 8 章 一个实际的例子

## 1. Pi 值估算原理

Hadoop 自带的例子中，有一个计算 Pi 值的例子。这例子比较全面，它用的 API 是旧版的。本章先分析一下这个例子，然后再用新版的 API 重新实现一下。

这个程序的原理是这样的。假如有一个边长为1 的正方形。以正方形的一个端点为圆心，以 1 为半径，画一个圆弧，于是在正方形内就有了一个直角扇形。在正方形里随机生成若干的点，则有些点是在扇形内，有些点是在扇形外。正方形的面积是 1，扇形的面积是 0.25*Pi。设点的数量一共是 n，扇形内的点数量是 nc，在点足够多足够密集的情况下，会近似有 nc/n 的比值约等于扇形面积与正方形面积的比值，也就是 nc/n = 0.25*Pi/1，即 Pi = 4*nc/n。

如何生成随机点？最简单的方式是在[0, 1]的区间内每次生成两个随机小数作为随机点的 x 和 y 坐标。可惜这种生成方式效果不够好，随机点之间有间隙过大和重叠的可能，会让计算精度不够高。 Halton 序列算法生成样本点的效果要好得多，更均匀，计算精度比随机生成的点更高，因此这个例子用 Halton 序列算法生成点集。关于 Halton 序列可以参考这里 http://orion.math.iastate.edu/reu/2001/voronoi/halton_sequence.html 和这里 http://www.aae.wisc.edu/dphaneuf/AAE%20875/Halton%20sequences.pdf，在这里就不详细说了。

在正方形内生成的样本点越多，计算 Pi 值越精确，这样，这个问题就很适合用 Hadoop 来处理啦。假设要在正方形内生成 1000 万个点，可以设置 10 个 Map 任务，每个 Map 任务处理 100 万个点，也可以设置 100 个 Map 任务，每个 Map 任务处理 10 万个点。

## 2. 旧版 API 的 Pi 值估算 MapReduce 程序

此处带来来自 Hadoop 的示例程序。

为了计算，设置 10 个 Map 任务，每个任务处理 1000 个点，具体流程是这样的：

1) 运行 PiEstimator 的 MapReduce 程序，输入参数是 10，1000，意思是设置 10 个 Map 任务，每个 Map 任务处理 1000 个点。

2) PiEstimator 进行初始化。初始化时，有一个步骤是在 HDFS 上生成一个目录，也就是输入目录。这个目录下有 10 个序列文件。Map 任务的数量的数量决定序列文件的数量，PiEstimator 就生成有 10 个序列文件。每个序列文件保存两个整数，分别是要处理的样本点在 Halton 序列的序号和生成样本点的数量。也就是说，第一个文件的内容是”0, 1000”，第二个文件的内容是”1000, 1000”，第三个文件的内容是“2000, 1000”，第四个文件的内容是“3000, 1000”，以此类推。如果用 Halton 序列算法生成一万个样本点，那么，第一个 Map 任务生成的点的序号是从 0 到 999，第二个 Map 任务生成的点的序号是从 1000 到 1999，第三个 Map 任务生成的点的序号是从 2000 到 2999，以此类推 。Halton 序列算法生成随机点的的唯一参数是序号。

3) PiEstimator 运行 MapReduce 任务。

4) PiEstimator 从 MapReduce 的输出目录读取两个整数，它们分别是直角扇形内的点的数量和直角扇形外的点的数量。

5) 根据 4)的结果数值，计算 Pi 值，然后返回。

PiEstimator.java 文件的对应 PiEstimator 类。PiEstimator 类有三个内部类，分别是 HalthonSequence 类，PiMapper 类，PiReducer 类。HalthonSequence 类负责产生样本点，PiMapper 类是 Map 过程，PiReducer 类是 Reduce 过程。

PiRefuce.java 的代码如下：

```
package org.apache.hadoop.examples;

import java.io.IOException;
import java.math.BigDecimal;
import java.util.Iterator;

import org.apache.hadoop.conf.Configured;
import org.apache.hadoop.fs.FileSystem;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.BooleanWritable;
import org.apache.hadoop.io.LongWritable;
import org.apache.hadoop.io.SequenceFile;
import org.apache.hadoop.io.Writable;
import org.apache.hadoop.io.WritableComparable;
import org.apache.hadoop.io.SequenceFile.CompressionType;
import org.apache.hadoop.mapred.FileInputFormat;
import org.apache.hadoop.mapred.FileOutputFormat;
import org.apache.hadoop.mapred.JobClient;
import org.apache.hadoop.mapred.JobConf;
import org.apache.hadoop.mapred.MapReduceBase;
import org.apache.hadoop.mapred.Mapper;
import org.apache.hadoop.mapred.OutputCollector;
import org.apache.hadoop.mapred.Reducer;
import org.apache.hadoop.mapred.Reporter;
import org.apache.hadoop.mapred.SequenceFileInputFormat;
import org.apache.hadoop.mapred.SequenceFileOutputFormat;
import org.apache.hadoop.util.Tool;
import org.apache.hadoop.util.ToolRunner;

public class PiEstimator extends Configured implements Tool {
  //临时目录的路径，保存运行中的文件数据。
  static private final Path TMP_DIR = new Path(
      PiEstimator.class.getSimpleName() + "_TMP_3_141592654");

  //Halton 序列类，产生随机点。
```

```
private static class HaltonSequence {
  static final int[] P = {2, 3};
  static final int[] K = {63, 40};

  private long index;
  private double[] x;
  private double[][] q;
  private int[][] d;

   //构造函数
  HaltonSequence(long startindex) {
    index = startindex;
    x = new double[K.length];
    q = new double[K.length][];
    d = new int[K.length][];
    for(int i = 0; i < K.length; i++) {
      q[i] = new double[K[i]];
      d[i] = new int[K[i]];
    }

    for(int i = 0; i < K.length; i++) {
      long k = index;
      x[i] = 0;
      for(int j = 0; j < K[i]; j++) {
        q[i][j] = (j == 0? 1.0: q[i][j-1])/P[i];
        d[i][j] = (int)(k % P[i]);
        k = (k - d[i][j])/P[i];
        x[i] += d[i][j] * q[i][j];
      }
    }
  }

  //生成点
  double[] nextPoint() {
    index++;
    for(int i = 0; i < K.length; i++) {
      for(int j = 0; j < K[i]; j++) {
        d[i][j]++;
        x[i] += q[i][j];
        if (d[i][j] < P[i]) {
          break;
        }
        d[i][j] = 0;
        x[i] -= (j == 0? 1.0: q[i][j-1]);
      }
    }
    return x;
  }
```

```
  }

  //PiMapper 类，定义 Map 过程
  public static class PiMapper extends MapReduceBase
    implements Mapper<LongWritable, LongWritable, BooleanWritable, LongWritable> {
    public void map(LongWritable offset,
                    LongWritable size,
                    OutputCollector<BooleanWritable, LongWritable> out,
                    Reporter reporter) throws IOException {
      /*
     函数 map 从序列文件里得到 offset 和 size，offset 是要产生的样本点在 Halton 序列里的序
号，size 是这个 Map 任务需要产生的样本点数量。
      */
      final HaltonSequence haltonsequence = new HaltonSequence(offset.get());
      long numInside = 0;
      long numOutside = 0;

      for(long i = 0; i < size.get(); ) {
        //生成随机点
        final double[] point = haltonsequence.nextPoint();
        //判断随机点是否在直角扇形内。
        final double x = point[0] - 0.5;
        final double y = point[1] - 0.5;
        if (x*x + y*y > 0.25) {
          numOutside++;
        } else {
          numInside++;
        }
        i++;
        //每产生 1000 个点，就更新一下状态，提示作用。
        if (i % 1000 == 0) {
          reporter.setStatus("Generated " + i + " samples.");
        }
      }

      //Map 任务运行完毕，输出结果。
      out.collect(new BooleanWritable(true), new LongWritable(numInside));
      out.collect(new BooleanWritable(false), new LongWritable(numOutside));
    }
  }

  //PiReduce 类，定义 Reduce 过程
  public static class PiReducer extends MapReduceBase
    implements Reducer<BooleanWritable, LongWritable, WritableComparable<?>, Writable> {
    private long numInside = 0;
    private long numOutside = 0;
    private JobConf conf;
```

```
  @Override
  public void configure(JobConf job) {
    conf = job;
  }

  public void reduce(BooleanWritable isInside,
                     Iterator<LongWritable> values,
                     OutputCollector<WritableComparable<?>, Writable> output,
                     Reporter reporter) throws IOException {
    /*
   isInside 是布尔类型，或者是 true 或者是 false。
   values 包含 10 个数值，分别来自 10 个 Map 任务。
    */

    //在这里，累加计算 10 次 Map 任务所有的直角扇形之外和直角扇形之内的样本点数量。
    if (isInside.get()) {
      for(; values.hasNext(); numInside += values.next().get());
    } else {
      for(; values.hasNext(); numOutside += values.next().get());
    }
  }

  //在 Reduce 过程结束后，将计算结果写到临时目录的文件中，以供 PiEstimator 类读取。
  @Override
  public void close() throws IOException {
    Path outDir = new Path(TMP_DIR, "out");
    Path outFile = new Path(outDir, "reduce-out");
    FileSystem fileSys = FileSystem.get(conf);
    SequenceFile.Writer writer = SequenceFile.createWriter(fileSys, conf,
        outFile, LongWritable.class, LongWritable.class,
        CompressionType.NONE);
    writer.append(new LongWritable(numInside), new LongWritable(numOutside));
    writer.close();
  }
}

//PiEstimator 类的 estimate 函数，运行 MapReduce，然后计算 Pi 值。
public static BigDecimal estimate(int numMaps, long numPoints, JobConf jobConf
    ) throws IOException {
  jobConf.setJobName(PiEstimator.class.getSimpleName());
  jobConf.setInputFormat(SequenceFileInputFormat.class);

  jobConf.setOutputKeyClass(BooleanWritable.class);
  jobConf.setOutputValueClass(LongWritable.class);
  jobConf.setOutputFormat(SequenceFileOutputFormat.class);

  jobConf.setMapperClass(PiMapper.class);
  jobConf.setNumMapTasks(numMaps);
```

```
    jobConf.setReducerClass(PiReducer.class);
    jobConf.setNumReduceTasks(1);

    /*
    关掉 speculative execution 功能。
    speculative execution 功能是指，假如 Hadoop 发现有些任务执行的比较慢，那么，它会在其他的节点上再运行一个同样的任务。这两个任务，哪个先完成就以哪个结果为准。
    但 Reduce 任务需要将数值写入到 HDFS 的文件里，而且这个文件名是固定的，如果同时运行两个以上的 Reduce 任务，会导致写入出错，所以要关闭这个功能。
    */
    jobConf.setSpeculativeExecution(false);

    //设置输入目录和输出目录
    final Path inDir = new Path(TMP_DIR, "in");
    final Path outDir = new Path(TMP_DIR, "out");
    FileInputFormat.setInputPaths(jobConf, inDir);
    FileOutputFormat.setOutputPath(jobConf, outDir);

    //生成输入目录
    final FileSystem fs = FileSystem.get(jobConf);
    if (fs.exists(TMP_DIR)) {
      throw new IOException("Tmp directory " + fs.makeQualified(TMP_DIR)
          + " already exists.  Please remove it first.");
    }

    if (!fs.mkdirs(inDir)) {
      throw new IOException("Cannot create input directory " + inDir);
    }

    try {
      //为每个 Map 任务生成一个序列文件，并写入数值。
      for(int i=0; i < numMaps; ++i) {
        final Path file = new Path(inDir, "part"+i);
        final LongWritable offset = new LongWritable(i * numPoints);
        final LongWritable size = new LongWritable(numPoints);
        final SequenceFile.Writer writer = SequenceFile.createWriter(
            fs, jobConf, file,
            LongWritable.class, LongWritable.class, CompressionType.NONE);
        try {
          writer.append(offset, size);
        } finally {
          writer.close();
        }
        System.out.println("Wrote input for Map #"+i);
      }

      //运行 MapReduce 任务
```

```
    System.out.println("Starting Job");
    final long startTime = System.currentTimeMillis();
    JobClient.runJob(jobConf);
    final double duration = (System.currentTimeMillis() - startTime)/1000.0;
    System.out.println("Job Finished in " + duration + " seconds");

    //读取输出结果
    Path inFile = new Path(outDir, "reduce-out");
    LongWritable numInside = new LongWritable();
    LongWritable numOutside = new LongWritable();
    SequenceFile.Reader reader = new SequenceFile.Reader(fs, inFile, jobConf);
    try {
      reader.next(numInside, numOutside);
    } finally {
      reader.close();
    }

    //计算 Pi 值然后返回
    return BigDecimal.valueOf(4).setScale(20)
        .multiply(BigDecimal.valueOf(numInside.get()))
        .divide(BigDecimal.valueOf(numMaps))
        .divide(BigDecimal.valueOf(numPoints));
  } finally {
    fs.delete(TMP_DIR, true);
  }
}

//实现 Tool 接口的 run 函数
public int run(String[] args) throws Exception {
  if (args.length != 2) {
    System.err.println("Usage: "+getClass().getName()+" <nMaps> <nSamples>");
    ToolRunner.printGenericCommandUsage(System.err);
    return -1;
  }

  final int nMaps = Integer.parseInt(args[0]);
  final long nSamples = Long.parseLong(args[1]);

  System.out.println("Number of Maps  = " + nMaps);
  System.out.println("Samples per Map = " + nSamples);

  final JobConf jobConf = new JobConf(getConf(), getClass());
  System.out.println("Estimated value of Pi is "
      + estimate(nMaps, nSamples, jobConf));
  return 0;
}

//PiEstimator 类的 main 函数
```

```
  public static void main(String[] argv) throws Exception {
    System.exit(ToolRunner.run(null, new PiEstimator(), argv));
  }
}
```

PiEstimator 的运行次序是：1)PiEstimator 类的 main 函数；2)PiEsitmator 类的 run 函数；3)PiEsitmator 类的 estimat 函数。

# 3. 新版 API 的 Pi 值估算的 MapReduce 程序

新版 API 和旧版 API 有较大差异，包括 Mapper，Reducer，Job 都有变化。本章在旧版 API 的代码上进行修改，给出一个基于新版 API 的程序。在多数地方已经做了注释，故不再一一解释。

## 3.1 NewPiEst.java 文件的源代码

```
package com.brianchen.hadoop;

import java.lang.Exception;
import java.lang.Integer;
import java.math.BigDecimal;
import java.io.IOException;
import java.lang.InterruptedException;
import java.lang.Iterable;

import org.apache.hadoop.conf.Configured;
import org.apache.hadoop.util.Tool;
import org.apache.hadoop.util.ToolRunner;
import org.apache.hadoop.mapreduce.Job;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.LongWritable;
import org.apache.hadoop.mapreduce.Mapper;
import org.apache.hadoop.mapreduce.Reducer;
import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.FileSystem;
import org.apache.hadoop.io.SequenceFile;
import org.apache.hadoop.io.SequenceFile.CompressionType;
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;
import org.apache.hadoop.mapreduce.lib.input.SequenceFileInputFormat;
import org.apache.hadoop.mapreduce.lib.output.SequenceFileOutputFormat;
import org.apache.commons.logging.Log;
```

```
import org.apache.commons.logging.LogFactory;
import org.apache.hadoop.util.GenericOptionsParser;

public class NewPiEst extends Configured implements Tool{
  //临时目录，存储用
  static private final Path TMP_DIR = new Path("pitmp");
  //Log
  static final Log LOG = LogFactory.getLog(NewPiEst.class);

  //Halton 序列的类
  private static class HaltonSequence{
    // bases
    static final int[]  P = {2, 3};
    // maximum number of digits allowed
    static final int[] K = {63, 40};

    private long index;
    private double[] x;
    private double[][] q;
    private int[][] d;

    HaltonSequence(long startindex){
      index = startindex;
      x = new double[K.length];
      q = new double[K.length][];
      d = new int[K.length][];
      for(int i = 0; i < K.length; i++){
        q[i] = new double[K[i]];
        d[i] = new int[K[i]];
      }

      for(int i = 0; i < K.length; i++){
        long k = index;
        x[i] = 0;

        for(int j = 0; j < K[i]; j++){
          q[i][j] = (j == 0? 1.0: q[i][j-1])/P[i];
          d[i][j] = (int)(k % P[i]);
          k = (k - d[i][j])/P[i];
          x[i] += d[i][j] * q[i][j];
        }
      }
    }

    double[] nextPoint(){
      index++;
      for(int i = 0; i < K.length;  i++){
       for(int j = 0; j < K[i]; j++){
```

```
          d[i][j]++;
          x[i] += q[i][j];
          if (d[i][j] < P[i]){
         break;
          }
          d[i][j] = 0;
          x[i] -= (j == 0? 1.0: q[i][j-1]);
        }
      }
      return x;
    }
  }

  //新版 API 的 Mapper 类
    public static class PiMapper extends Mapper<LongWritable, LongWritable, LongWritable,
LongWritable>{
    public void map(LongWritable offset, LongWritable size, Context context)
                    throws IOException, InterruptedException{
      final HaltonSequence hs = new HaltonSequence(offset.get());
      long nInside = 0;
      long nOutside = 0;

      for(int i = 0; i < size.get(); i++){
      final double[] point = hs.nextPoint();
        if (point[0]*point[0] + point[1]*point[1] > 1){
        nOutside++;
        }else{
        nInside++;
        }

        context.write(new LongWritable(1), new LongWritable(nOutside));
        context.write(new LongWritable(2), new LongWritable(nInside));
      }
    }
  }

  //新版 API 的 Reducer 类
  public static class PiReducer extends
       Reducer<LongWritable, LongWritable, LongWritable, LongWritable> {
    long nInside = 0;
    long nOutside = 0;

    public void reduce(LongWritable isInside, Iterable<LongWritable> values, Context context)
                        throws IOException, InterruptedException{
      if (isInside.get() == 2 ){
      for (LongWritable val : values) {
        nInside += val.get();
        }
```

```
      }else{
        for (LongWritable val : values) {
        nOutside += val.get();
        }
      }

         LOG.info("reduce-log:" + "isInside = " + isInside.get() + ", nInside = "+ nInside + ", nOutSide = "+nOutside );
    }

   //Reducer 类在结束前执行 cleanup 函数，于是在这里将 reduce 过程计算的 nInside 和 nOutSide 写入文件。
    @Override
    protected void cleanup(Context context) throws IOException, InterruptedException{
      Path OutDir = new Path(TMP_DIR, "out");
      Path outFile = new Path(OutDir, "reduce-out");
      Configuration conf = new Configuration();
      FileSystem fs = FileSystem.get(conf);
      SequenceFile.Writer writer = SequenceFile.createWriter(
            fs, conf, outFile, LongWritable.class, LongWritable.class, CompressionType.NONE);
      writer.append(new LongWritable(nInside), new LongWritable(nOutside));
      writer.close();
    }
  }

  public static BigDecimal estimate(int nMaps, int nSamples, Job job)throws Exception{
    LOG.info("\n\n estimate \n\n");

    //设置 Job 的 Jar，Mapper，Reducer 等等
    job.setJarByClass(NewPiEst.class);
    job.setMapperClass(PiMapper.class);
    job.setReducerClass(PiReducer.class);
    job.setNumReduceTasks(1);

    //设置输入输出格式为序列文件格式
    job.setInputFormatClass(SequenceFileInputFormat.class);
    job.setOutputFormatClass(SequenceFileOutputFormat.class);

    //设置输出键和输出值的类型
    job.setOutputKeyClass(LongWritable.class);
    job.setOutputValueClass(LongWritable.class);

    job.setSpeculativeExecution(false);

    Path inDir = new Path(TMP_DIR, "in");
    Path outDir = new Path(TMP_DIR, "out");

    //设置输入文件所在目录和输出结果所在目录
```

```
    FileInputFormat.addInputPath(job, inDir);
    FileOutputFormat.setOutputPath(job, outDir);

    //检查目录
    FileSystem fs = FileSystem.get(job.getConfiguration());
    if (fs.exists(TMP_DIR)){
       throw new IOException("Tmp directory " + fs.makeQualified(TMP_DIR) + " already exists,
pls remove it.");
    }

    //生成目录
    if (!fs.mkdirs(inDir)){
      throw new IOException("Cannot create input directory " + inDir);
    }

    try{
      //生成若干个序列文件，每个文件放两个整数。每个序列文件将对应一个 Map 任务
      for(int i = 0; i < nMaps; i++){
      final Path file = new Path(inDir, "part"+i);
        final LongWritable offset = new LongWritable(i*nSamples);
        final LongWritable size = new LongWritable(nSamples);
        final SequenceFile.Writer writer = SequenceFile.createWriter(
          fs, job.getConfiguration(), file,
            LongWritable.class, LongWritable.class, CompressionType.NONE);
        writer.append(offset, size);
        writer.close();
        System.out.println("wrote input for Map #" + i);
      }

      //执行 MapReduce 任务
      System.out.println("starting mapreduce job");
      final long startTime = System.currentTimeMillis();
      boolean ret = job.waitForCompletion(true);
      final double duration = (System.currentTimeMillis() - startTime)/1000.0;
      System.out.println("Job finished in " + duration + " seconds.");

      //从 HDFS 将 MapReduce 的结果读取出来
      Path inFile = new Path(outDir, "reduce-out");
      LongWritable nInside = new LongWritable();
      LongWritable nOutside = new LongWritable();
      SequenceFile.Reader reader = new SequenceFile.Reader(fs, inFile, job.getConfiguration());
      reader.next(nInside, nOutside);
      reader.close();
      LOG.info("estimate-log: " + "nInside = "+nInside.get()+", nOutSide = "+nOutside.get());

      //计算 Pi 值然后返回
      return BigDecimal.valueOf(4).multiply(
              BigDecimal.valueOf(nInside.get())
```

```
          ).divide(
                                        BigDecimal.valueOf(nInside.get() + nOutside.get()), 20,
BigDecimal.ROUND_HALF_DOWN
              );
    }finally{
      fs.delete(TMP_DIR, true);
    }
  }

  public int run(String[] args) throws Exception{
    LOG.info("\n\n run \n\n");
    if (args.length != 2){
      System.err.println("Use: NewPieEst 10 10000");
      System.exit(1);
    }

    //解析参数
    int nMaps = Integer.parseInt(args[0]);
    int nSamples = Integer.parseInt(args[1]);
    Configuration conf = new Configuration();
    String[] otherArgs = new GenericOptionsParser(conf, args).getRemainingArgs();
    Job job = new Job(conf, "Pi estimating job");

    System.out.println("Pi = " + estimate(nMaps, nSamples, job));

    return 0;
  }

  public static void main(String[] argv) throws Exception{
    LOG.info("\n\n main \n\n");
    System.exit(ToolRunner.run(null, new NewPiEst(), argv));
  }
}
```

## 3.2 编译和运行

项目的目录和文件结构跟前面章节类似，请作为练习处理，下面给出近似的各命令。

编译：　”javac -cp /home/brian/usr/hadoop/hadoop-1.2.1/hadoop-core-1.2.1.jar:/home/brian/usr/hadoop/hadoop-1.2.1/lib/commons-logging-1.1.1.jar:/home/brian/usr/hadoop/hadoop-1.2.1/lib/commons-cli-1.2.jar -d ./classes/ src/*.java”

打包：”jar -cvf newpiest.jar -C ./classes/ .”

运行：　”./bin/hadoop jar ~/all/work/ifitus/ifitus-dev-svn/hadoop_from_zero/src/chp08/newpiest/newpiest.jar com.brianchen.hadoop.NewPiEst 10 1000”

# 推荐书目

关于 Hadoop 的书非常多，不能一一列举，每本书都有可取之处。我只列一下自己看过的书，略带一点书评。

1.《Java2 核心技术》，这套书有两卷。学习 Java 的话，搞定这两本就差不多了，足以应对 Hadoop 的常规开发。

2.《Hadoop 权威指南》，这本书英文版已经出到第 3 版了，中文版是第 2 版。这本书内容比较全面。缺点是废话太多，书太厚，例子复杂，代码没写全，有些地方的代码是低版本 API。当然，在 Hadoop 快速发展的时代里，出现这些问题都是很正常的。我觉得等 Hadoop 稳定了，这本书继续改版的话，很有可能成为 Hadoop 的权威经典。

3.《Hadoop 技术内幕-深入解析 MapReduce 架构设计与实现原理》和《Hadoop 技术内幕-深入解析 Hadoop Common 和 HDFS 架构设计与实现原理》，这两本书是国人写的，非常之好，语言清晰易懂，叙述细致，校印精确，强烈推荐有志于 Hadoop 的同学人手一套。

4.《Hadoop 实战》，英文书名是《Hadoop in Action》，这本书非常好，是从人类容易理解的方式阐述 Hadoop 如何入门的。如果你搞定了这本书，我的这本书就不用看了。如果你看完我这本书，再看这本书会很轻松，可以补充很多细节问题。

# 后记

## 未来的趋势是什么

在越来越多的任务上，计算机比人类做得更快更强成本更低。麻省理工的两位经济学家写了一本书叫《与机器竞赛》http://www.geekonomics10000.com/639，书中说，以人工智能为代表的计算机技术可以完全代替司机驾驶汽车，可以在商业上代替人类翻译员，IBM 的 Watson 计算机已经在传统电视竞答节目上击败人类选手，苹果公司的 Siri 语言助手对人类语言可以理解得很好。再比如，一种新的软件可以代替高级放射科医生分析医学图像，成本降到原来的百分之一。在公认的人类才能做的律师行业，在计算机软件的帮助下，一个律师可以可以完成以前 500 个律师的工作，而且准确率更高一大截，成本低到不可思议的程度。

因此，社会将不断地用计算机取代人类的工作，计算机能做的工作会越来越广泛。掌握最新的计算机技术的人在未来有更多的机会得到好工作，为自己和公司创造更大的价值。在美国硅谷，顶级的程序员们已经象体育明星电影明星一样，有了自己的经纪人 http://www.36kr.com/p/202520.html。

另一方面，如果不掌握最新的技术，会面临巨大的失业风险。《与机器竞赛》说，技术进步导致美国的就业机会更少，同时越来越明显地看出，财富是由少数人操纵资本、机器设备和软件创造出来。FaceBook 公司曾经在市值是 1000 亿美元时只有 3000 多员工。Twitter 公司估值 100 亿美元，只有 900 多员工。更早一点的公司，Google 市值 2500 亿美元，员工 3 万人。比 Google 更早的公司，微软市值也是 2500 亿美元左右，员工有 8 万人。越新的公司，单位市值对应的员工数越少，趋势非常明显！

这种趋势不但在美国发生，在中国也开始了，2013 年 8 月 10 号和 11 号，中央电视台在晚上黄金时间报道了美的电器，同方热水器，沈阳新松机器人公司等生产和使用机器人替代人力的新闻分析，甚至河南平顶山这种传统的煤炭城市也开始生产智能机器人了。华晨宝马汽车在沈阳的新工厂里，95%的工位是机器人做的，只有返修工位是工人处理。我们相信，中国未来会在更多的行业使用机器人，智能度更高，效率更高，跟美国的差距不会太远。我们无法预测未来的行业是是怎么变化的，但很明显的是，那些拥有普通的技能，做着普通的工作，拿着普通的工资，过着普通的生活的人，会被自动化淘汰。

根据上述的事实，未来好的职业只能是不用担心这种未来机器取代人的趋势，必然是肯定不会被智能软件和机器人取代的技能—这样就不用担心失业问题，这种技能必须是本质上有难度的，需要经过精深的练习才可以掌握 --这就意味着稀缺性和好的薪资，这种技能如果遇到机遇可以创造巨大的财富 --看看数量众多的美国创业公司的发展就知道了。

## 为什么大数据很重要

那么，哪种技术最有可能符合上述条件呢？个性化服务，或者高度专业的技能。我认为，大数据技术是高度专业化的技能，理由如下：其一，未来几乎所有行业都会逐步扩展到应对全球客户，要快速处理海量数据，由此维持和

增强竞争力，既然几乎所有行业都要逐步使用大数据技术，因此它会提供大量的职位和创新机会；其二，大数据是有技术难度的，如果要做好大数据，必须对 Linux 操作系统，网络技术，编程语言，人工智能与模式识别都有一定程度的了解和掌握，它们都会跟大数据相关，并且要有实际项目经验，因此技术的全面性和技术的深度是大数据行业的门槛和护城河；其三，智能软件和机器人可以代替人，那么最终不可代替的就是开发智能软件和机器人的人。

众所周知，信息技术行业有三个显著的特点： 1 ) 技术更新非常快，从技术更新的角度说，几个月发生一次小变化，一两年发生一次大变化，新的语言，新的方案，新的技术趋势层出不穷，仅仅研究这些变化都需要花很多时间 2 ) 工作压力非常大，加班到晚上九点十点是常有的事情。几年前华为的工程师曾经猝死和跳楼。在北京，下班后通用汽车的大楼和微软研究院的大楼截然不同，前者的灯是关掉的，而后者常常灯火通明。3 ) 爆发迅猛，快速改变人生和世界，最简单地说，从创造财富的角度看，几乎没有什么行业比 IT 行业更快，比如美国不断爆发出几十个人做出一个产品被数亿数十亿美元收购的新闻。如果想快速地赚一大笔钱，这个行业是最合适的。如果不想创业，在这个行业工作也是一种很好的选择，在北上广起薪可观，如果有几年的工作经验，薪水还可以更高。

# 如何学习大数据最快

那么，我们怎样解决大数据的学习问题呢？根据一万小时天才理论，任何人只要通过持续的精深练习，就可以学会任意一种技能，天赋只是精深练习的结果。如果能持续一万小时的精深练习，任何人都能成为天才式人物。在通常情况下，一般人只要能很好地掌握一种稀缺的技能，就足以获得一份不错的工作，这只需要几个月的精深训练就足够了。

如果想学习大数据，必须要学会 Hadoop。Hadoop 是大数据行业使用最多最广的开源工具，围绕 Hadoop 已经形成了一个巨大的生态环境。那么，我们应该如何学习 Hadoop 呢？

1. 试错成本

如果自学一种快速发展中的新技术，最大的成本是试错。比较成熟的技术，都有若干本经典书籍可供学习，这些经典书非常好，经过多次印刷出版，几乎没有错误，讲述完善，习题全面，语言流畅，按照它们的路子走可以很顺利搞定这些技术，诸如《Unix 高级环境编程》，《算法导论》，《深入理解计算机系统》，《C 程序设计语言》，《Java 设计模式》，《JavaScript 权威指南》，太多了，不一一列出。但对 Hadoop 这种本身仍在快速发展中的技术，大部分已出版书籍的内容往往落后于技术本身，中文版的比英文版的更慢一些。那么，用这些书籍学习的时候，会遇到各种不一致的问题，代码的，编译的，API 变动，有时候甚至从安装和配置都不一致。一般来说，书里一定会有错误，示例代码原样敲上也会有编译问题。根据我的估算，学习过程的三分之一的时间都花费在处理这些麻烦上头。

2. 冗余成本

一个新技术不全是新的，通常是用成熟的技术实现几个核心理念　　Hadoop 的核心理念是两个：MapReduce 和 HDFS。在不同的层次上，对 MapReduce 和 HDFS 的理解的深度也是不同的。Hadoop 体系非常复杂，有诸多的细节，试图在短时间内完全理解 Hadoop 是不现实的。因此，我们需要根据学习的目标，将学习定位在不同的层次上，如应用层，架构层，然后有选择地进行学习。如果方向走偏了，会导致比较大的冗余成本---把时间花费在不该花费的地方。

3. 精深练习

做一遍是最好的学习方式。读书千遍，不如做题一遍，做题千万，不如一个 Project 实现。第一阶段，要熟悉书里的例子，能做到不看例子的源代码，可以查 java 和 hadoop 的 API 的情况下，将它们写出来，能运行和调试成功，

能解释清楚每一行代码的作用。第二个阶段，遇到一个大数据的问题，比如说某些经典书的章尾习题，某些创业公司的业务需求，或者跟其他行业朋友的闲谈，能从熟悉的例子里找到相近的“模式”，然后作一些修改满足需求，这个阶段是需要一些灵感的，有时候需要回头重新看书看基础知识，有时候需要多查一点资料，但基本上来说只要你做过一次，这块知识就融入你的灵魂，你会象熟悉自己一样熟悉它。第三个阶段，当解决了足够多的有价值实际问题，你会在它们之中找到共性，然后实现一个通用解决方案，或者一个有价值的开源软件，到这个地步，就是非常好的开发人员了。