

# 云计算大作业：基于 Hadoop 平台的 MapReduce 程序

数据科学与计算机学院 计算机科学与技术 2016 级

王凯祺 16337233

2019 年 5 月 31 日

## 1 问题

MapReduce 主要用于处理数据统计问题，而 Hadoop 平台可以分布式地执行 MapReduce 程序。我看到老师给的选题中，有 Web Access Log Statistics，正好我前段时间拿下了一台 Web 服务器，目前有能力获取网页访问记录的资源。因此，我选择用 MapReduce 做网页访问数据统计。

**特别声明** 攻击服务器的用途仅限于学术交流。

## 2 Web 服务器配置

该 Web 服务器使用的是 Apache 2 服务，默认只打开了错误日志，而无访问日志。为了获得访问日志，我们需要修改 Apache 配置，添加自定义记录：

```
1 CustomLog /var/log/apache2/access.log common
2 LogFormat "%h %l %u %t \"%r\" %>s %O" common
```

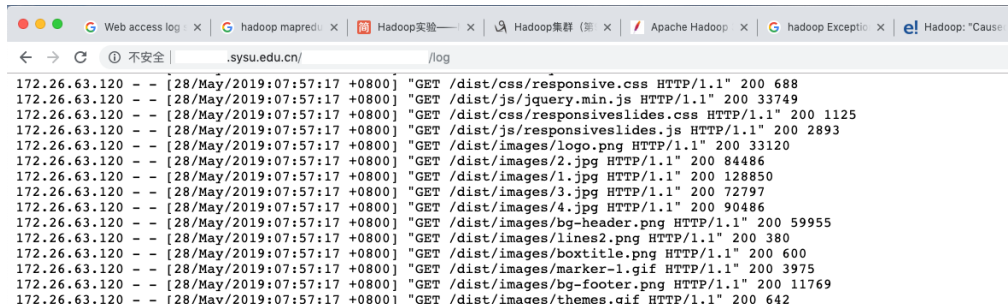
更改配置后，需重启 Apache 2 服务，使配置生效：

```
1 service apache2 restart
```

为了更方便地下载日志，我们将 access.log 创建软链接到网页目录下面，并修改权限。

```
1 ln -s /var/log/apache2/access.log /var/www/html/~/log
2 chmod 755 /var/log/apache2/access.log
3 chown www-data /var/log/apache2/access.log
```

用浏览器访问我刚刚指定的网页目录，获得访问日志。



### 3 统计分析数据

我们对这些数据进行如下统计：

- 统计每天的请求数、数据量。
- 统计每个时段（0 点至 1 点、1 点至 2 点、…、23 点至次日 0 点）的请求数、数据量。
- 统计每个 IP 地址的请求数、数据量。

### 4 编写 MapReduce 程序

明确统计对象后，我们通过修改示例程序 WordCount，就能完成以下的统计。

```
1 import java.io.IOException;
2 import java.util.StringTokenizer;
3
4 import org.apache.hadoop.conf.Configuration;
5 import org.apache.hadoop.fs.Path;
6 import org.apache.hadoop.io.LongWritable;
7 import org.apache.hadoop.io.Text;
8 import org.apache.hadoop.mapreduce.Job;
9 import org.apache.hadoop.mapreduce.Mapper;
10 import org.apache.hadoop.mapreduce.Reducer;
11 import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
12 import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;
13 import org.apache.hadoop.util.GenericOptionsParser;
14
15 public class WebAccessLog {
16
17     public static class Map extends Mapper<Object, Text, Text, LongWritable> {
18         // Mapper<Input Key, Input Value, Output Key, Output Value>
19         private final static LongWritable one = new LongWritable(1);
20         private Text ip = new Text(); // IP address
21         private Text date = new Text(); // date (string)
22         private Text time = new Text(); // time (hour only, string)
23         private Text method = new Text(); // HTTP request method (GET/POST/etc)
24         private Text path = new Text(); // URL path
25         private Text http_code = new Text(); // HTTP code (200/etc)
26         private LongWritable size = new LongWritable(0); // Request/Response size (
            in Bytes)
27
28         @Override
29         public void map(Object key, Text value, Context context) throws IOException,
            InterruptedException {
30             StringTokenizer itr = new StringTokenizer(value.toString());
31             if (itr.countTokens() != 7) return;
32             if (itr.hasMoreTokens()) ip.set(itr.nextToken()); // IP address
33             if (itr.hasMoreTokens()) date.set(itr.nextToken()); // date (string)
34             if (itr.hasMoreTokens()) time.set(itr.nextToken().substring(0, 2)); //
                time (hour only, string)
```

```

35         if (itr.hasMoreTokens()) method.set(itr.nextToken()); // HTTP request
36             method (GET/POST/etc)
37         if (itr.hasMoreTokens()) path.set(itr.nextToken()); // URL path
38         if (itr.hasMoreTokens()) http_code.set(itr.nextToken()); // HTTP code
39             (200/etc)
40         if (itr.hasMoreTokens()) size = new LongWritable(Long.parseLong(itr.
41             nextToken())); // Request/Response size (in Bytes)
42         context.write(time, one); // count request amount according to time
43         // context.write(time, size); // count data size according to time
44     }
45 }
46
47 public static class Reduce extends Reducer<Text, LongWritable, Text,
48     LongWritable> {
49     // Reducer<Input Key, Input Value, Output Key, Output Value>
50     private LongWritable result = new LongWritable(0);
51     @Override
52     public void reduce(Text key, Iterable<LongWritable> values, Context context)
53         throws IOException, InterruptedException {
54         // sum up the values
55         long sum = 0;
56         for (LongWritable val : values) {
57             sum += val.get();
58         }
59         result.set(sum);
60         context.write(key, result);
61     }
62 }
63
64 public static void main(String[] args) throws Exception {
65     Configuration conf = new Configuration();
66     String[] otherArgs = new GenericOptionsParser(conf, args).getRemainingArgs()
67         ; // Input folder & Output folder
68     if (otherArgs.length != 2) {
69         System.err.println("Usage: _WebAccessLog_<in>_<out>");
70         System.exit(2);
71     }
72     Job job = Job.getInstance(conf, "WebAccessLog");
73     job.setJarByClass(WebAccessLog.class);
74     job.setMapperClass(Map.class);
75     job.setReducerClass(Reduce.class);
76     job.setOutputKeyClass(Text.class);
77     job.setOutputValueClass(LongWritable.class);
78     FileInputFormat.addInputPath(job, new Path(otherArgs[0]));
79     FileOutputFormat.setOutputPath(job, new Path(otherArgs[1]));
80     System.exit(job.waitForCompletion(true) ? 0 : 1);
81 }

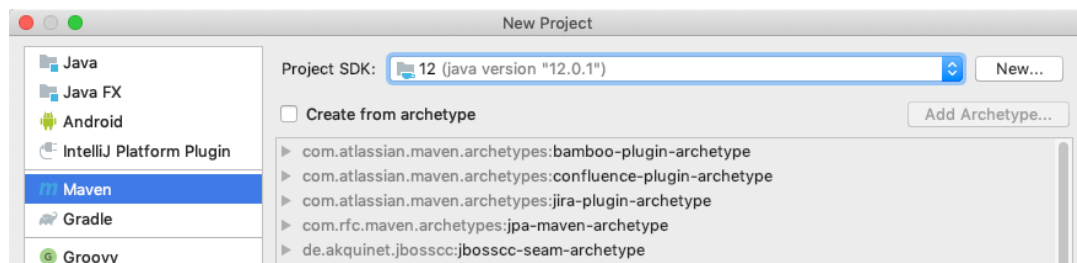
```

## 5 单机运行

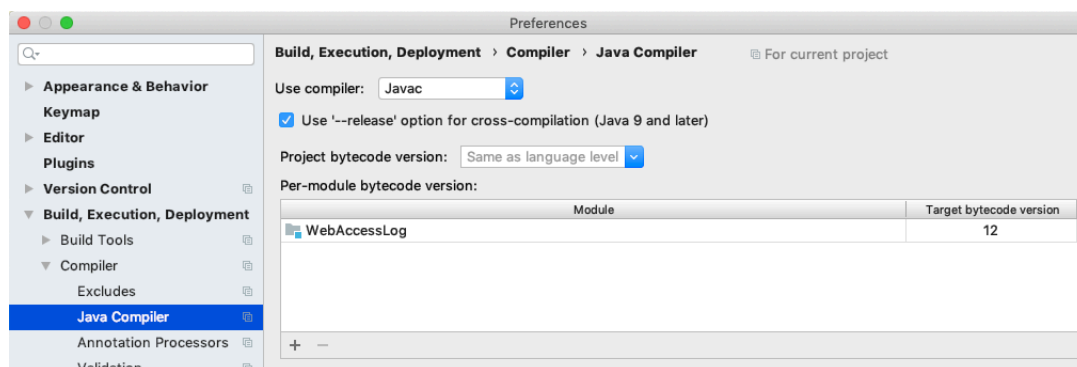
为调试和运行方便，我使用 IntelliJ IDEA 进行本地运行和调试 MapReduce 程序，无需搭建 Hadoop 和 HDFS 环境。这样编写的程序通常不需要经过修改即可在真实的分布式 Hadoop 集群下运行。

### 5.1 新建项目

在 IntelliJ 中点击 File → New → Project，选择 Maven，JDK 版本越高越好。



打开 IntelliJ 的偏好设置，定位到 Java Compiler，将 WebAccessLog 的 Target Bytecode Version 设置为与 JDK 版本号相同的版本。



### 5.2 配置依赖

新建项目后，编辑 Maven 配置文件 pom.xml。

#### 5.2.1 添加源

在 project 内尾部添加

```
1 <repositories>
2   <repository>
3     <id>apache</id>
4     <url>http://maven.apache.org</url>
5   </repository>
6 </repositories>
```

### 5.2.2 添加依赖

我们只需要用到基础依赖 `hadoop-core` 和 `hadoop-common` 。  
在 `project` 内尾部添加

```
1 <dependencies>
2   <dependency>
3     <groupId>org.apache.hadoop</groupId>
4     <artifactId>hadoop-core</artifactId>
5     <version>1.2.1</version>
6   </dependency>
7   <dependency>
8     <groupId>org.apache.hadoop</groupId>
9     <artifactId>hadoop-common</artifactId>
10    <version>2.7.2</version>
11  </dependency>
12 </dependencies>
```

对于 `hadoop-core` 和 `hadoop-common` 的版本号，我们建议在 Google 搜一个最新的填上去。

修改 `pom.xml` 完成后，IntelliJ 会提示 `Maven projects need to be Imported`，点击 `Import Changes` 以更新依赖。

### 5.3 主程序

在 `src main java` 下新建一个 `WebAccessLog` 类，在里面写 `MapReduce` 程序。

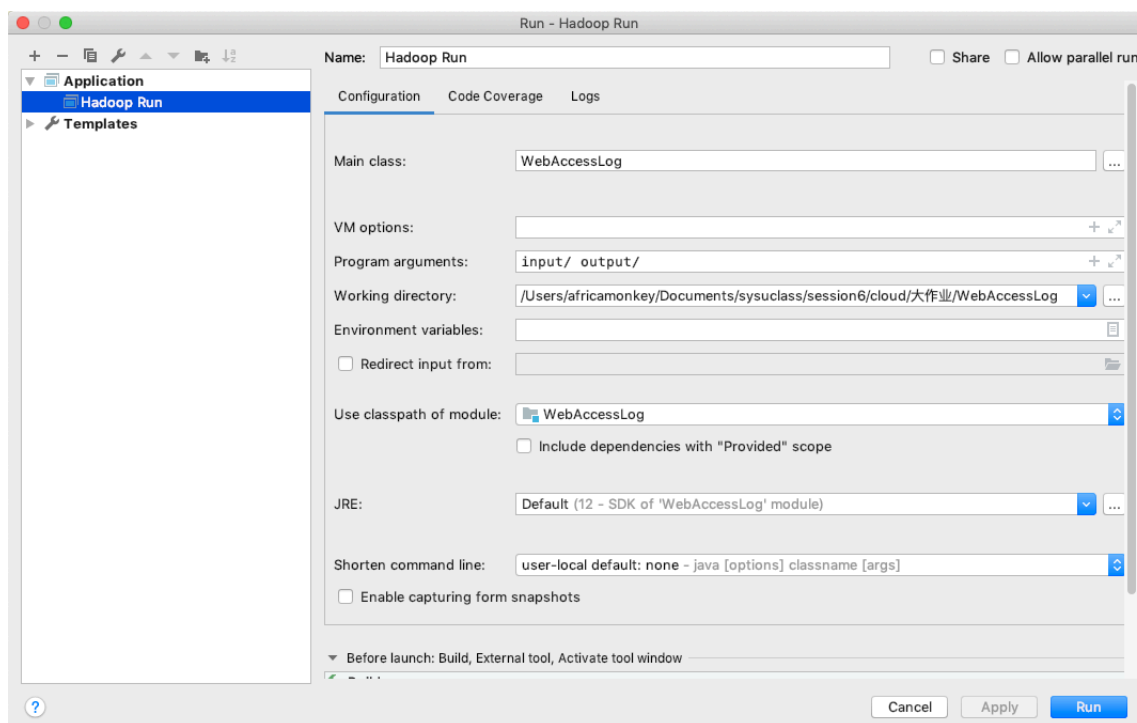
### 5.4 配置输入文件

我的 `WebAccessLog` 程序读取每行访问信息，然后统计相关统计量。首先在 `WebAccessLog` 目录下（`src` 同级目录）新建一个文件夹 `input`，并添加一个或多个文本文件到 `input` 中。

### 5.5 配置运行参数

这里我们需要配置此程序运行时的 `Main class`，以及 `WebAccessLog` 需要的输入输出路径。

在 IntelliJ 菜单栏中选择 `Run` → `Edit Configurations`，在弹出来的对话框中点击 `+`，新建一个 `Application` 配置。配置 `Main class` 为 `WebAccessLog`，`Program arguments` 为 `input/ output/`，即输入路径 `input` 文件夹，输出为 `output`。



## 5.6 运行

点击菜单栏 Run → Run 'WebAccessLog' 即开始运行 MapReduce 程序，IntelliJ 下方会显示 Hadoop 的运行输出。程序运行完毕后，IntelliJ 左上方会出现新的文件夹 output，其中的 part-r-00000 就是运行的结果。

## 5.7 运行结果

从实验开始，到撰写本报告为止，Apache 日志记录了 3 天共 25925 次请求。

### 5.7.1 统计每天的请求数、数据量

| part-r-00000 | WebAccess         | part-r-00000 | WebAccessL           |
|--------------|-------------------|--------------|----------------------|
| 1            | 27/May/2019 16570 | 1            | 27/May/2019 86815718 |
| 2            | 28/May/2019 5179  | 2            | 28/May/2019 30994717 |
| 3            | 29/May/2019 3600  | 3            | 29/May/2019 25705660 |
| 4            | 30/May/2019 1653  | 4            | 30/May/2019 18777560 |
| 5            |                   | 5            |                      |

其中，左图表示的是请求数，右图表示的是数据量（字节）。可以看到请求数和数据量是呈正相关的。

你们可能会觉得很奇怪，访问量为什么会指数级地减少。其原因是 28 号为提交作业的 DDL。过了 DDL，又没有新的作业布置下来，自然访问量就少了。

通过分析这个数据，我们可以很轻松地分析出老师布置作业的时间和频率。

## 5.7.2 统计每个时段的请求数、数据量

| part-r-00000 |    |      | part-r-00000 |    |          |
|--------------|----|------|--------------|----|----------|
| 1            | 00 | 2262 | 1            | 00 | 10626611 |
| 2            | 01 | 217  | 2            | 01 | 1192079  |
| 3            | 02 | 147  | 3            | 02 | 174173   |
| 4            | 03 | 247  | 4            | 03 | 337093   |
| 5            | 07 | 36   | 5            | 07 | 606845   |
| 6            | 08 | 586  | 6            | 08 | 3723233  |
| 7            | 09 | 692  | 7            | 09 | 5321295  |
| 8            | 10 | 598  | 8            | 10 | 6992753  |
| 9            | 11 | 107  | 9            | 11 | 1223836  |
| 10           | 12 | 197  | 10           | 12 | 2365668  |
| 11           | 13 | 281  | 11           | 13 | 974358   |
| 12           | 14 | 515  | 12           | 14 | 4648549  |
| 13           | 15 | 413  | 13           | 15 | 1684065  |
| 14           | 16 | 639  | 14           | 16 | 5844818  |
| 15           | 17 | 1032 | 15           | 17 | 7559685  |
| 16           | 18 | 1441 | 16           | 18 | 7741248  |
| 17           | 19 | 1477 | 17           | 19 | 7536344  |
| 18           | 20 | 2596 | 18           | 20 | 15974354 |
| 19           | 21 | 2212 | 19           | 21 | 12845605 |
| 20           | 22 | 5646 | 20           | 22 | 30426782 |
| 21           | 23 | 4584 | 21           | 23 | 22877826 |

我们能看出来，从 20 点至次日 1 点是网站访问的高峰，在 22 点至 23 点达到峰值。看来我们的同学们喜欢在晚上做题，甚至有同学在凌晨 1 点至 4 点做题……我都快惊呆了！

## 5.7.3 统计每个 IP 地址的请求数、数据量

| part-r-00000 |                |     | part-r-00000 |                |         |
|--------------|----------------|-----|--------------|----------------|---------|
| 1            | 125.217.174.10 | 30  | 1            | 125.217.174.10 | 140625  |
| 2            | 125.217.174.22 | 14  | 2            | 125.217.174.22 | 86125   |
| 3            | 172.16.19.153  | 1   | 3            | 172.16.19.153  | 510     |
| 4            | 172.17.41.23   | 1   | 4            | 172.17.41.23   | 504     |
| 5            | 172.18.109.163 | 185 | 5            | 172.18.109.163 | 1491533 |
| 6            | 172.18.109.9   | 207 | 6            | 172.18.109.9   | 1280044 |
| 7            | 172.18.153.160 | 30  | 7            | 172.18.153.160 | 567549  |
| 8            | 172.18.153.217 | 7   | 8            | 172.18.153.217 | 10737   |
| 9            | 172.18.158.205 | 1   | 9            | 172.18.158.205 | 510     |
| 10           | 172.18.159.150 | 1   | 10           | 172.18.159.150 | 510     |
| 11           | 172.18.160.84  | 1   | 11           | 172.18.160.84  | 510     |
| 12           | 172.18.166.180 | 18  | 12           | 172.18.166.180 | 565136  |
| 13           | 172.18.233.217 | 8   | 13           | 172.18.233.217 | 7752    |
| 14           | 172.18.32.11   | 412 | 14           | 172.18.32.11   | 1884643 |
| 15           | 172.18.32.144  | 272 | 15           | 172.18.32.144  | 1934652 |

我们可以对每个 IP 地址统计它的请求数和数据量，若某个 IP 地址产生的数据量远超过上传下载作业所需的作业量，我们就能把这样的 IP 地址找出来，并对这个 IP 所产生的数据做进一步的调查与分析。

# 6 分布式运行

## 6.1 Master 服务器的搭建

为实现分布式运行，我先在阿里云部署 1 台服务器作为 Master，待安装好全套 Hadoop 环境和 JDK 环境后，用 Master 制作镜像，然后使用 Master 镜像再在阿里云申请 3 台服务器作为 Slave。

硬件配置

- CPU: Intel(R) Xeon(R) Platinum 8163 CPU @ 2.50GHz；核心数: 1；平均基准 CPU 计算性能: 10%
- RAM: 1 GB

- 内网带宽: 200 Mbps

服务器操作系统: Ubuntu 18.04 LTS

为方便起见, 在本实验中使用的用户均为 root 。

## 6.2 Hadoop 环境配置

### 6.2.1 下载 Hadoop

访问 Apache 网站获取下载链接, 在 Master 上直连 Apache 官网下载。下载完成后解压到当前目录下。

在 root 的用户目录 ‘/root/’ 下执行:

```
1 wget https://www-us.apache.org/dist/hadoop/common/hadoop-3.1.2/hadoop-3.1.2.tar.gz
2 tar -zxvf hadoop-3.1.2.tar.gz
```

### 6.2.2 下载和安装 JDK

在 root 的用户目录 ‘/root/’ 下执行:

```
1 wget --no-check-certificate -c --header "Cookie:_oraclelicense=accept-securebackup-
   cookie" https://download.oracle.com/otn-pub/java/jdk/12.0.1+12/69
   cfe15208a647278a19ef0990eea691/jdk-12.0.1_linux-x64_bin.deb
2 dpkg -i jdk-12.0.1_linux-x64_bin.deb
```

### 6.2.3 设置 HDFS 服务器地址

在 ‘/root/hadoop-3.1.2/etc/hadoop/core-site.xml’ 中: 将 localhost 改为 Master 服务器的 IP 地址。

```
1 <configuration>
2   <property>
3     <name>fs.defaultFS</name>
4     <value>hdfs://localhost:9000</value>
5   </property>
6 </configuration>
```

### 6.2.4 设置环境变量

在 ‘/root/.bashrc’ 文件尾追加:

```
1 export ROOT="root"
2 export HDFS_NAMENODE_USER=$ROOT
3 export HDFS_DATANODE_USER=$ROOT
4 export HDFS_SECONDARYNAMENODE_USER=$ROOT
5 export YARN_RESOURCEMANAGER_USER=$ROOT
6 export YARN_NODEMANAGER_USER=$ROOT
7 export JAVA_HOME="/usr/lib/jvm/jdk-12.0.1"
8 export JRE_HOME="${JAVA_HOME}/jre"
9 export HADOOP_HOME="/root/hadoop-3.1.2"
```



```

10 export CLASSPATH=".:${JAVA_HOME}/lib:${JRE_HOME}/lib:${HADOOP_HOME}/bin/hadoop_
    classpath) "
11 export PATH="${JAVA_HOME}/bin:$PATH"
12 export HADOOP_COMMON_LIB_NATIVE_DIR=$HADOOP_HOME/lib/native
13 export HADOOP_OPTS="-Djava.library.path=$HADOOP_HOME/lib"

```

### 6.2.5 单机试运行示例程序

在 ‘/root/hadoop-3.1.2’ 中运行:

```

1 bin/hadoop jar share/hadoop/mapreduce/hadoop-mapreduce-examples-3.1.2.jar grep input
    output 'dfs[a-z.]+'

```

### 6.2.6 配置 SSH 免密登录

首先生成一组公钥和私钥

```

1 ssh-keygen -t rsa -P '' -f ~/.ssh/id_rsa

```

将该公钥列入本机白名单，凭该公钥对应的私钥可登录本机。

```

1 cat ~/.ssh/id_rsa.pub >> ~/.ssh/authorized_keys

```

## 6.3 创建及配置 Slave 服务器

我们将 Master 服务器创建镜像，然后使用该镜像创建 3 台 Slave 服务器。由于我们已经在镜像中配置好 Hadoop、JDK、环境变量和免密登录，我们在 Slave 机器上无需作任何配置，只需保证服务器正常运行即可。

我们需记录下阿里云为 Slave 服务器分配的 IP 地址（如不喜欢可更改）。

| <input type="checkbox"/> 实例ID/名称                                   | 标签 | 监控 | 可用区 ▼      | IP地址              |
|--|----|----|------------|-------------------|
| <input type="checkbox"/> i-wz920at8x7uw4ve82o2v<br>hadoop-slave003 |    |    | 华南 1 可用区 A | 172.18.88.248(私有) |
| <input type="checkbox"/> i-wz920at8x7uw4ve82o2u<br>hadoop-slave002 |    |    | 华南 1 可用区 A | 172.18.88.246(私有) |
| <input type="checkbox"/> i-wz920at8x7uw4ve82o2t<br>hadoop-slave001 |    |    | 华南 1 可用区 A | 172.18.88.247(私有) |
| <input type="checkbox"/> i-wz95o45s4xlmqvs2xhy<br>hadoop-test      |    |    | 华南 1 可用区 D | 172.18.57.97(私有)  |

为方便管理，我们在 Master 服务器上的 /etc/hosts 文件中添加 3 台 Slave 服务器的别名。

## 6.4 在 Master 服务器上加入 Slave 服务器

在 ‘/root/hadoop-3.1.2/etc/hadoop/workers’ 中，加入 Slave 服务器的 IP 地址、别名或域名。

```
3. root@hadoop-test: ~ (ssh)
1 127.0.0.1 localhost
2
3
4 # The following lines are desirable for IPv6 capable hosts
5 ::1 localhost ip6-localhost ip6-loopback
6 ff02::1 ip6-allnodes
7 ff02::2 ip6-allrouters
8
9 172.18.57.97 hadoop-test hadoop-test
10 172.18.88.247 slave1
11 172.18.88.246 slave2
12 172.18.88.248 slave3
```

```
1 slave1
2 slave2
3 slave3
```

## 6.5 自动初始化和自动运行脚本

每次初始化集群都要敲一大堆命令，由于我非常懒，不想次次都敲，就写了个脚本。代码的意思是，先停止 Hadoop 的全部服务，把该清的清了，创建并格式化 HDFS，再启动所有服务，最后在 HDFS 下为 root 用户创建一个用户目录。

初始化脚本 ‘/root/hadoop-3.1.2/init.sh’：

```
1 sbin/stop-all.sh
2 ssh localhost "rm -rf /tmp/*"
3 ssh slave1 "rm -rf /tmp/*"
4 ssh slave2 "rm -rf /tmp/*"
5 ssh slave3 "rm -rf /tmp/*"
6 bin/hdfs namenode -format
7 sbin/start-all.sh
8 bin/hdfs dfs -mkdir /user
9 bin/hdfs dfs -mkdir /user/root
```

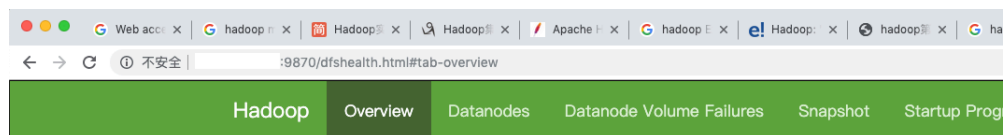
每次启动之前要将 HDFS 上的 input、output 文件夹删除，然后将本地的 input 文件夹上传至 HDFS。为此，我也写了个启动脚本。

启动脚本 ‘/root/hadoop-3.1.2/start\_task.sh’：

```
1 bin/hdfs dfs -rm -r output/
2 bin/hdfs dfs -rm -r input/
3 bin/hdfs dfs -put input/
4 bin/hadoop jar ~/WebAccessLog/WebAccessLog.jar WebAccessLog input output
```

## 6.6 运行

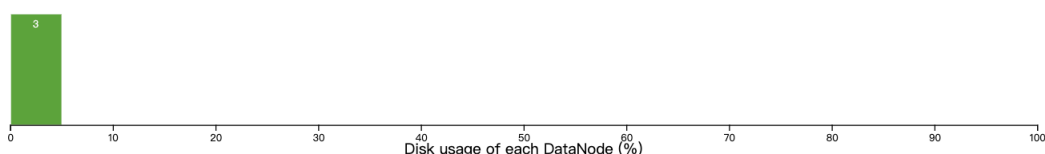
执行初始化脚本，打开浏览器，访问 <http://Master-ip:9870>，可以看到 Hadoop 集群的运行状态



## Overview 'hadoop-test:9000' (active)

|                |  |
|----------------|--|
| Started:       | Thu May 30 22:04:24 +0800 2019                             |
| Version:       | 3.1.2, r1019dde65bcf12e05ef48ac71e84550d589e5d9a           |
| Compiled:      | Tue Jan 29 09:39:00 +0800 2019 by sunilg from branch-3.1.2 |
| Cluster ID:    | CID-de7c4205-b571-432a-932f-976fe5df080b                   |
| Block Pool ID: | BP-2047905374-172.18.57.97-1559225056899                   |

### Datanode usage histogram



### In operation

Show  entries Search:

| Node  | Http Address  | Last contact | Last Block Report | Capacity                        | Blocks | Block pool used | Version |
|---|---|--------------|-------------------|---------------------------------|--------|-----------------|---------|
| ✓hadoop-slave001:9866<br>(172.18.88.247:9866) | <a href="http://hadoop-slave001:9866">http://hadoop-slave001:9866</a> | 0s           | 0m                | 39.25 GB <div><div></div></div> | 0      | 24 KB (0%)      | 3.1.2   |
| ✓hadoop-slave002:9866<br>(172.18.88.246:9866) | <a href="http://hadoop-slave002:9866">http://hadoop-slave002:9866</a> | 2s           | 0m                | 39.25 GB <div><div></div></div> | 0      | 24 KB (0%)      | 3.1.2   |
| ✓hadoop-slave003:9866<br>(172.18.88.248:9866) | <a href="http://hadoop-slave003:9866">http://hadoop-slave003:9866</a> | 0s           | 0m                | 39.25 GB <div><div></div></div> | 0      | 24 KB (0%)      | 3.1.2   |

Showing 1 to 3 of 3 entries Previous **1** Next

除了在网页能查看，我们还可以通过在 Master 上执行 `bin/hdfs dfsadmin -report` 在命令行查看 Hadoop 集群的运行状态。

上传 `WebAccessLog.java` 到 Master 服务器上，放在 `/root/WebAccessLog/` 下。

编译 `WebAccessLog.java` :

```
1 javac WebAccessLog.java
```

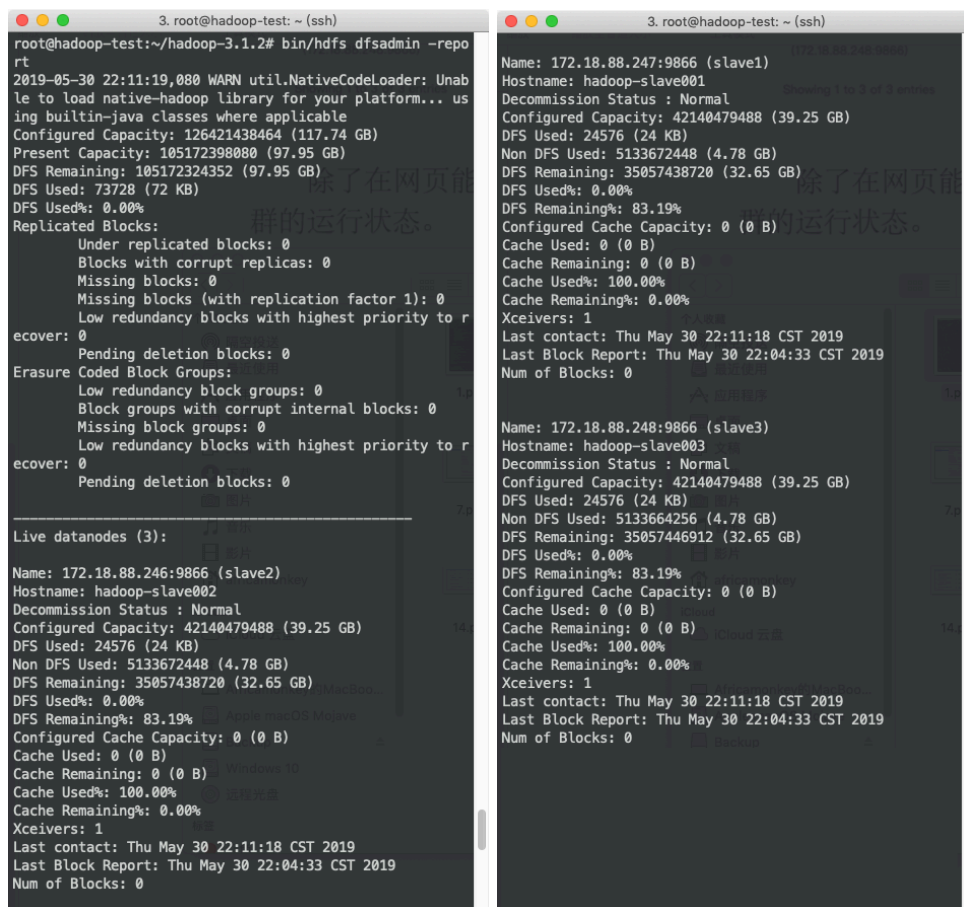
将生成的 class 打包:

```
1 jar -cvf WebAccessLog.jar *.class
```

上传 `access.log` 日志到 `/root/hadoop-3.1.2/input/` 下。

然后执行启动脚本，期间会有 `mapreduce` 的过程日志刷屏。

```
1 ./start_task.sh
```



查看结果：

```
1 bin/hdfs dfs -cat output/*
```

下载结果：

```
1 bin/hdfs dfs -get output
```

## 6.7 性能测试

由于我们没有这么多数据，我们将这 27002 条（与前面的数字不同，原因是撰写报告的时候又多了一些访问量）的数据复制 200 份，这样大约产生 540 万条数据。我们就用这些数据来测试 Hadoop 性能。

我们以“统计每个 IP 地址的数据量”的程序为例，分别测试在 1 个、2 个和 3 个 Slave 的条件下，完成 MapReduce 所需的时间。

| Slave 数量 | 用时      |
|----------|---------|
| 1        | 90.037s |
| 2        | 91.052s |
| 3        | 90.552s |

咦这就奇怪了，为什么无论是几台 Slave，完成 MapReduce 的时间都是 90s 左右呢？

我发现我选服务器选的是最便宜的入门级（共享）……很大可能它们 3 个 Slave 都是共享同一个核心的 CPU 跑的。

然后我开了 3 台配置好一点的机器，测得耗时分别是 95.319s、94.669s、93.411s，原因应该不是我之前的猜测。

我在服务器跑的过程中监控了 CPU 使用率（如下图），发现只有 Master 的 CPU 使用率是比较高的，而 Slave 的使用率都比较低。我认为 Slave 只运行了 HDFS，而未实际参与 Map/Reduce 的计算，那这个 Slave 就很没用了。

| PID  | USER | PR | NI  | VIRT    | RES    | SHR   | S | %CPU | %MEM | TIME+   | COMMAND |
|------|------|----|-----|---------|--------|-------|---|------|------|---------|---------|
| 8    | root | 20 | 0   | 0       | 0      | 0     | I | 0.3  | 0.0  | 0:00.14 | rcu_sc+ |
| 2267 | root | 20 | 0   | 41868   | 3828   | 3328  | R | 0.3  | 0.0  | 0:00.65 | top     |
| 3387 | root | 20 | 0   | 3982576 | 212256 | 29724 | S | 0.3  | 2.6  | 0:09.51 | java    |
| 1    | root | 20 | 0   | 77520   | 8608   | 6668  | S | 0.0  | 0.1  | 0:01.40 | systemd |
| 2    | root | 20 | 0   | 0       | 0      | 0     | S | 0.0  | 0.0  | 0:00.00 | kthrea+ |
| 4    | root | 0  | -20 | 0       | 0      | 0     | I | 0.0  | 0.0  | 0:00.00 | kworke+ |
| 6    | root | 0  | -20 | 0       | 0      | 0     | I | 0.0  | 0.0  | 0:00.00 | mm_per+ |
| 7    | root | 20 | 0   | 0       | 0      | 0     | S | 0.0  | 0.0  | 0:00.04 | ksofti+ |
| 9    | root | 20 | 0   | 0       | 0      | 0     | I | 0.0  | 0.0  | 0:00.00 | rcu_bh  |
| 10   | root | rt | 0   | 0       | 0      | 0     | S | 0.0  | 0.0  | 0:00.00 | migrat+ |
| 11   | root | rt | 0   | 0       | 0      | 0     | S | 0.0  | 0.0  | 0:00.00 | watchd+ |
| 12   | root | 20 | 0   | 0       | 0      | 0     | S | 0.0  | 0.0  | 0:00.00 | cpuhp/0 |

## 7 总结

MapReduce 还是处理大数据的很有用的工具，它能帮助我们对访问记录等做统计数据的工作。在本问题中，MapReduce 切切实实地帮助我们分析了日访问量、用户活跃时间、可疑 IP 地址行为，并可根据实际需要添加 HTTP 状态码分析、扩展名分析等功能。加入这些功能后，可以通过 HTTP 状态码查找出遇到问题的网页，也可以帮助我们统计静态资源使用量，从而更精确地把某些静态资源移到 CDN 上。Hadoop 使用 Java 语言，使得 MapReduce 还能跨平台工作，甚至是将源代码编译后可直接在别的服务器上运行。Hadoop 的配置不复杂，但没能充分利用 Slave 服务器的 CPU 资源。

用 Hadoop 还有个好处就是文档丰富，用户也比较多，所以在搜索引擎中搜索一个问题总是大概率能找到答案。用 Java 写陌生的继承类必须要有 IDE。我在 Android 中使用了 Android Studio，觉得非常棒，可以很清晰地看到继承类的定义、方法的定义等等；另外，它在程序运行前、运行后都可以添加脚本，方便自动化配置输入输出文件运行。我查看“关于”知 Android Studio 是在 IntelliJ 平台开发的。于是我在 JetBrains 官网找是否有 Java 的 IDE，那当然是有的啦——IntelliJ IDEA。IntelliJ IDEA 的界面、布局、快捷键跟 Android Studio 都极为相似，基本上是零学习成本上手。