1.0 Hadoop 教程

分类 Hadoop 教程

Hadoop 是一个开源的分布式计算和存储框架，由 Apache 基金会开发和维护。

Hadoop 为庞大的计算机集群提供可靠的、可伸缩的应用层计算和存储支持，它允许使用简单的编程模型跨计算机群集分布式处理大型数据集，并且支持在单台计算机到几千台计算机之间进行扩展。

Hadoop 使用 Java 开发，所以可以在多种不同硬件平台的计算机上部署和使用。其核心部件包括分布式文件系统 (Hadoop DFS，HDFS) 和 MapReduce。

Hadoop 历史

2003 年和 2004 年，Google 公司先后发表了两篇著名的论文 GFS 和 MapReduce。

这两篇论文和 2006 年发表的 BigTable 成为了现在著名的"Google 三大论文"。

Doug Cutting 在受到了这些理论的影响后开始了 Hadoop 的开发。

Hadoop 包含了两大核心组件。在 Google 的论文中，GFS 是一个在庞大的计算机集群中运行的分布式文件系统，在 Hadoop 中 HDFS 实现了它的功能。MapReduce 是一个分布式计算的方式，Hadoop 用同名称的 MapReduce 框架实现了它的功能。我们会在之后的 MapReduce 章节中详细介绍它。 从 2008 年开始，Hadoop 作为 Apache 顶级项目存在。它与它的众多子项目广泛应用于包括 Yahoo、阿里巴巴、腾讯等大型网络服务企业，并被 IBM、Intel、Microsoft 等平台公司列为支持对象。

Hadoop 的作用

Hadoop 的作用非常简单，就是在多计算机集群环境中营造一个统一而稳定的存储和计算环境，并能为其他分布式应用服务提供平台支持。

也就是说， Hadoop 在某种程度上将多台计算机组织成了一台计算机（做同一件事），那么 HDFS 就相当于这台计算机的硬盘，而 MapReduce 就是这台计算机的 CPU 控制器。

由于 Hadoop 是为集群设计的软件，所以我们在学习它的使用时难免会遇到在多台计算机上配置 Hadoop 的情况，这对于学习者来说会制造诸多障碍，主要有两个：

- 昂贵的计算机集群。多计算机构成的集群环境需要昂贵的硬件.
- 难以部署和维护。在众多计算机上部署相同的软件环境是一个大量的工作，而且非常不灵活，难以在环境更改后重新部署。

为了解决这些问题，我们有一个非常成熟的方式 **Docker**。

Docker 是一个容器管理系统，它可以向虚拟机一样运行多个"虚拟机"（容器），并构成一个集群。因为虚拟机会完整的虚拟出一个计算机来，所以会消耗大量的硬件资源且效率低下，而 Docker 仅提供一个独立的、可复制的运行环境，实际上容器中所有进程依然在主机上的内核中被执行，因此它的效率几乎和主机上的进程一样（接近100%）。

本教程将会以 Docker 为底层环境来描述 Hadoop 的使用，如果你不会使用 Docker 并且不了解更好的方式，请学习 Docker 教程。

> Windows 上 Docker 安装
>
> **注：**Windows 用户建议使用虚拟机方案安装 Docker。

Docker 部署

进入 Docker 命令行之后，拉取一个 Linux 镜像作为 Hadoop 运行的环境，这里推荐使用 CentOS 镜像（Debian 和其它镜像暂时会出现一些问题）。

```
docker pull centos:8
```

然后通过 docker images 命令可以查看到当前本地的镜像：

```
sobin@vmdebian:~$ docker pull centos:8
8: Pulling from library/centos
7a0437f04f83: Pull complete
Digest: sha256:5528e8b1b1719d34604c87e11dcd1c0a20bedf46e83b5632cdeac91b8c04efc1
Status: Downloaded newer image for centos:8
docker.io/library/centos:8
sobin@vmdebian:~$ docker images
REPOSITORY   TAG       IMAGE ID       CREATED       SIZE
centos       8         300e315adb2f   6 weeks ago   209MB
```

现在，我们创建一个容器：

```
docker run -d centos:8 /usr/sbin/init
```

通过 docker ps 可以查看运行中的容器：

```
sobin@vmdebian:~$ docker run -d centos:8 /usr/sbin/init
8fb5f97253a558ac51492148efd32ea93a366a4cfd2f390c41ebf9fb5fcfea12
sobin@vmdebian:~$ docker ps
CONTAINER ID   IMAGE      COMMAND            CREATED         STATUS         PORTS
   NAMES
8fb5f97253a5   centos:8   "/usr/sbin/init"   4 seconds ago   Up 3 seconds
   jolly_yalow
```

我们可以令容器打印出 Hello World:

```
sobin@vmdebian:~$ docker exec 8fb5f97253a5 echo "Hello World"
Hello World
```

到这里说明 Docker 已经安装并部署成功。

创建容器

Hadoop 支持在单个设备上运行，主要有两种模式：单机模式和伪集群模式。

本章讲述 Hadoop 的安装与单机模式。

配置 Java 与 SSH 环境

现在创建一个容器，名为 java_ssh_proto，用于配置一个包含 Java 和 SSH 的环境：

```
docker run -d --name=java_ssh_proto --privileged centos:8 /usr/sbin/init
```

然后进入容器:

```
docker exec -it java_ssh_proto bash
```

```
sobin@vmdebian:~$ docker run -d --name=java_ssh_proto --privileged centos:8 /usr/sb
in/init
197c3fa9b49a0ad02447fab04a471569fadc57ce9191c11e607ed0fbaed3d7c2
sobin@vmdebian:~$ docker exec -it java_ssh_proto bash
[root@197c3fa9b49a /]#
```

配置镜像：

```
sed -e 's|^mirrorlist=|#mirrorlist=|g' \
         -e
's|^#baseurl=http://mirror.centos.org/$contentdir|baseurl=https://mirrors.ustc.edu.
 \
         -i.bak \
         /etc/yum.repos.d/CentOS-Linux-AppStream.repo \
         /etc/yum.repos.d/CentOS-Linux-BaseOS.repo \
         /etc/yum.repos.d/CentOS-Linux-Extras.repo \
         /etc/yum.repos.d/CentOS-Linux-PowerTools.repo \
         /etc/yum.repos.d/CentOS-Linux-Plus.repo
yum makecache
```

安装 OpenJDK 8 和 SSH 服务：

```
yum install -y java-1.8.0-openjdk-devel openssh-clients openssh-server
```

然后启用 SSH 服务：

```
systemctl enable sshd && systemctl start sshd
```

> 如果是 ubuntu 系统，使用以下命令启动 SSH 服务：
>
> ```
> systemctl enable ssh && systemctl start ssh
> ```

到这里为止，如果没有出现任何故障，一个包含 Java 运行环境和 SSH 环境的原型容器就被创建好了。这是一个非常关键的容器，建议大家在这里先在容器中用 exit 命令退出容器，然后运行以下下两条命令停止容器，并保存为一个名为 java_ssh 的镜像：

```
docker stop java_ssh_proto
docker commit java_ssh_proto java_ssh
```

Hadoop 安装

下载 Hadoop

Hadoop 官网地址：http://hadoop.apache.org/

Hadoop 发行版本下载：https://hadoop.apache.org/releases.html

在目前的测试中，3.1.x 与 3.2.x 版本的兼容性较佳，本教程使用 3.1.4 版本作为案例。

Hadoop 3.1.4 镜像地址，下载好 tar.gz 压缩包文件备用。

创建 Hadoop 单机容器

现在以之前保存的 java_ssh 镜像创建容器 hadoop_single：

```
docker run -d --name=hadoop_single --privileged java_ssh /usr/sbin/init
```

将下载好的 hadoop 压缩包拷贝到容器中的 /root 目录下：

```
docker cp <你存放hadoop压缩包的路径> hadoop_single:/root/
```

进入容器：

```
docker exec -it hadoop_single bash
```

进入 /root 目录：

```
cd /root
```

这里应该存放着刚刚拷贝过来的 hadoop-x.x.x.tar.gz 文件，现在解压它：

```
tar -zxf hadoop-3.1.4.tar.gz
```

解压后将得到一个文件夹 hadoop-3.1.4，现在把它拷贝到一个常用的地方：

```
mv hadoop-3.1.4 /usr/local/hadoop
```

然后配置环境变量：

```
echo "export HADOOP_HOME=/usr/local/hadoop" >> /etc/bashrc
echo "export PATH=$PATH:$HADOOP_HOME/bin:$HADOOP_HOME/sbin" >> /etc/bashrc
```

然后退出 docker 容器并重新进入。

这时，echo $HADOOP_HOME 的结果应该是 /usr/local/hadoop

```
echo "export JAVA_HOME=/usr" >> $HADOOP_HOME/etc/hadoop/hadoop-env.sh
echo "export HADOOP_HOME=/usr/local/hadoop" >> $HADOOP_HOME/etc/hadoop/hadoop-
env.sh
```

这两步配置了 hadoop 内置的环境变量，然后执行以下命令判断是否成功：

```
hadoop version
```

```
[root@b051504bf718 ~]# hadoop version
Hadoop 3.1.4
Source code repository https://github.com/apache/hadoop.git -r 1e877761e8dadd71effe
f30e592368f7fe66a61b
Compiled by gabota on 2020-07-21T08:05Z
Compiled with protoc 2.5.0
From source with checksum 38405c63945c88fdf7a6fe391494799b
This command was run using /usr/local/hadoop/share/hadoop/common/hadoop-common-3.1.
4.jar
```

到这里，说明你的 Hadoop 单机版已经配置成功了。

3.0 Hadoop 概念

分类 Hadoop 教程

本章着重介绍 Hadoop 中的概念和组成部分，属于理论章节。如果你比较着急可以跳过。但作者不建议跳过，因为它与后面的章节息息相关。

Hadoop 整体设计

Hadoop 框架是用于计算机集群大数据处理的框架，所以它必须是一个可以部署在多台计算机上的软件。部署了 Hadoop 软件的主机之间通过套接字 (网络) 进行通讯。

Hadoop 主要包含 HDFS 和 MapReduce 两大组件，HDFS 负责分布储存数据，MapReduce 负责对数据进行映射、规约处理，并汇总处理结果。

Hadoop 框架最根本的原理就是利用大量的计算机同时运算来加快大量数据的处理速度。例如，一个搜索引擎公司要从上万亿条没有进行规约的数据中筛选和归纳热门词汇就需要组织大量的计算机组成集群来处理这些信息。如果使用传统数据库来处理这些信息的话，那将会花费很长的时间和很大的处理空间来处理数据，这个量级对于任何单计算机来说都变得难以实现，主要难度在于组织大量的硬件并高速地集成为一个计算机，即使成功实现也会产生昂贵的维护成本。

Hadoop 可以在多达几千台廉价的量产计算机上运行，并把它们组织为一个计算机集群。

一个 Hadoop 集群可以高效地储存数据、分配处理任务，这样会有很多好处。首先可以降低计算机的建造和维护成本，其次，一旦任何一个计算机出现了硬件故障，不会对整个计算机系统造成致命的影响，因为面向应用层开发的集群框架本身就必须假定计算机会出故障。

HDFS

Hadoop Distributed File System，Hadoop 分布式文件系统，简称 HDFS。

HDFS 用于在集群中储存文件，它所使用的核心思想是 Google 的 GFS 思想，可以存储很大的文件。

在服务器集群中，文件存储往往被要求高效而稳定，HDFS同时实现了这两个优点。

HDFS 高效的存储是通过计算机集群独立处理请求实现的。因为用户 (一半是后端程序) 在发出数据存储请求时，往往响应服务器正在处理其他请求，这是导致服务效率缓慢的主要原因。但如果响应服务器直接分配一个数据服务器给用户，然后用户直接与数据服务器交互，效率会快很多。

数据存储的稳定性往往通过"多存几份"的方式实现，HDFS 也使用了这种方式。HDFS 的存储单位是块 (Block) ，一个文件可能会被分为多个块储存在物理存储器中。因此 HDFS 往往会按照设定者的要求把数据块复制 n 份并存储在不同的数据节点 (储存数据的服务器) 上，如果一个数据节点发生故障数据也不会丢失。

HDFS 的节点

HDFS 运行在许多不同的计算机上，有的计算机专门用于存储数据，有的计算机专门用于指挥其它计算机储存数据。这里所提到的"计算机"我们可以称之为集群中的节点。

命名节点 (NameNode)

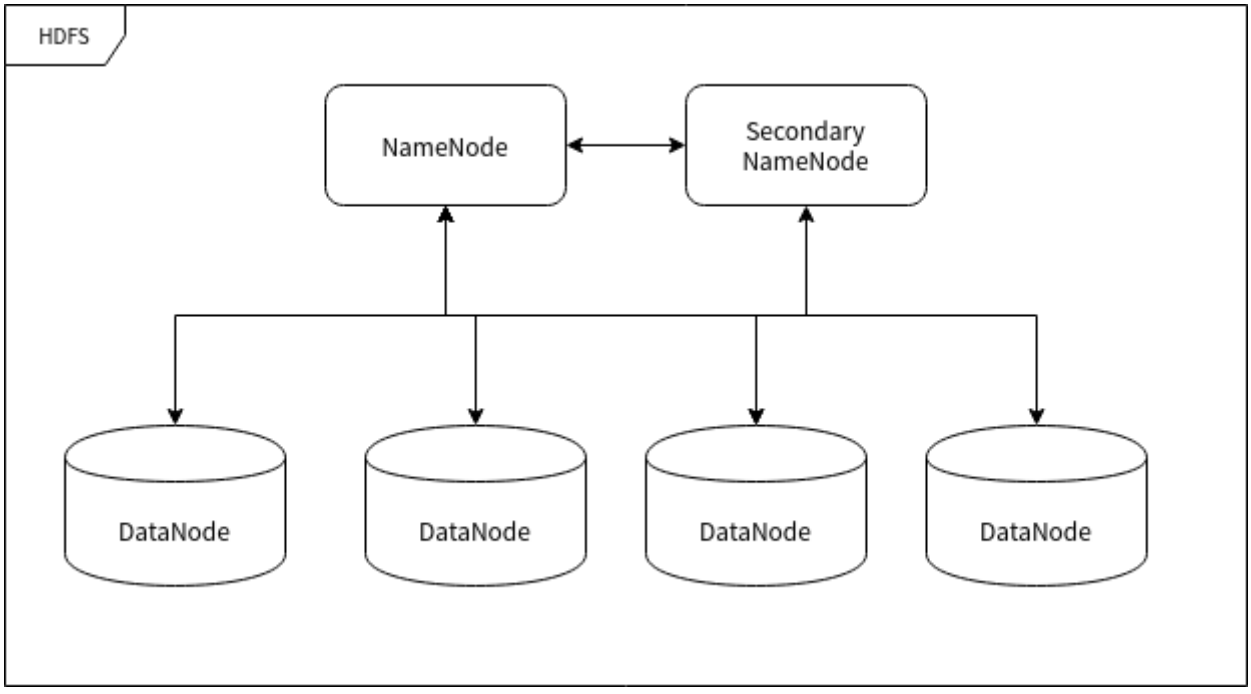
命名节点 (NameNode) 是用于指挥其它节点存储的节点。任何一个"文件系统"(File System, FS) 都需要具备根据文件路径映射到文件的功能，命名节点就是用于储存这些映射信息并提供映射服务的计算机，在整个 HDFS 系统中扮演"管理员"的角色，因此一个 HDFS 集群中只有一个命名节点。

数据节点 (DataNode)

数据节点 (DataNode) 使用来储存数据块的节点。当一个文件被命名节点承认并分块之后将会被储存到被分配的数据节点中去。数据节点具有储存数据、读写数据的功能，其中存储的数据块比较类似于硬盘中的"扇区"概念，是 HDFS 存储的基本单位。

副命名节点 (Secondary NameNode)

副命名节点 (Secondary NameNode) 别名"次命名节点"，是命名节点的"秘书"。这个形容很贴切，因为它并不能代替命名节点的工作，无论命名节点是否有能力继续工作。它主要负责分摊命名节点的压力、备份命名节点的状态并执行一些管理工作，如果命名节点要求它这样做的话。如果命名节点坏掉了，它也可以提供备份数据以恢复命名节点。副命名节点可以有多个。



MapReduce

MapReduce 的含义就像它的名字一样浅显：Map 和 Reduce (映射和规约) 。

大数据处理

大量数据的处理是一个典型的"道理简单，实施复杂"的事情。之所以"实施复杂"，主要是大量的数据使用传统方法处理时会导致硬件资源 (主要是内存) 不足。

现在有一段文字 (真实环境下这个字符串可能长达 1 PB 甚至更多) ，我们执行一个简单的"数字符"统计，即统计出这段文字中所有出现过的字符出现的数量：

```
AABABCABCDABCDE
```

统计之后的结果应该是：

字符	出现次数
A	5
B	4
C	3
D	2
E	1

统计的过程实际上很简单，就是每读取一个字符就要检查表中是否已经有相同的字符，如果没有就添加一条记录并将记录值设置为 1 ，如果有的话就直接将记录值增加 1。

但是如果我们将这里的统计对象由"字符"变成"词"，那么样本容量就瞬间变得非常大，以至于一台计算机可能难以统计数十亿用户一年来用过的"词"。

在这种情况下我们依然有办法完成这项工作——我们先把样本分成一段段能够令单台计算机处理的规模，然后一段段地进行统计，每执行完一次统计就对映射统计结果进行规约处理，即将统计结果合并到一个更庞大的数据结果中去，最终就可以完成大规模的数据规约。

在以上的案例中，第一阶段的整理工作就是"映射"，把数据进行分类和整理，到这里为止，我们可以得到一个相比于源数据小很多的结果。第二阶段的工作往往由集群来完成，整理完数据之后，我们需要将这些数据进行总体的归纳，毕竟有可能多个节点的映射结果出现重叠分类。这个过程中映射的结果将会进一步缩略成可获取的统计结果。

MapReduce 概念

IBM 的网站上找到了一篇 MapReduce 文章，地址：https://www.ibm.com/analytics/hadoop/mapreduce 。现在改编其中的一个 MapReduce 的处理案例来介绍 MapReduce 的原理细节以及相关概念。

这是一个非常简单的 MapReduce 示例。无论需要分析多少数据，关键原则都是相同的。

假设有 5 个文件，每个文件包含两列，分别记录一个城市的名称以及该城市在不同测量日期记录的相应温度。城市名称是键 (Key) ，温度是值 (Value) 。例如：(厦门，20)。现在我们要在所有数据中找到每个城市的最高温度 (请注意，每个文件中可能出现相同的城市)。

使用 MapReduce 框架，我们可以将其分解为 5 个映射任务，其中每个任务负责处理五个文件中的一个。每个映射任务会检查文件中的每条数据并返回该文件中每个城市的最高温度。

例如，对于以下数据：

城市	温度
厦门	12
上海	34
厦门	20
上海	15
北京	14
北京	16
厦门	24

上述数据的一个映射任务产生的结果如下所示：

城市	最高温度
厦门	24
上海	34
北京	16

假设其他四个映射器任务产生以下结果：

城市	最高温度
厦门	17
杭州	25
上海	29
北京	36

城市	最高温度
厦门	30
杭州	17
上海	31
北京	35
厦门	18
杭州	17
上海	17
北京	27
厦门	28
杭州	18
上海	14
北京	27

所有这 5 个结果将被输入到 Reduce 任务中，该任务组合输入结果并输出每个城市的单个值，产生如下的最终结果：

城市	最高温度
厦门	30
上海	34
北京	36
杭州	25

打个比方，你可以把 MapReduce 想象成人口普查，人口普查局会把若干个调查员派到每个城市。每个城市的每个人口普查人员都将统计该市的部分人口数量，然后将结果汇总返回首都。在首都，每个城市的统计结果将被规约到单个计数(各个城市的人口)，然后就可以确定国家的总人口。这种人到城市的映射是并行的，然后合并结果(Reduce)。这比派一个人以连续的方式清点全国中的每一个人效率高得多。

之前提到过的 Hadoop 三种模式：**单机模式**、**伪集群模式**和**集群模式**。

单机模式：Hadoop 仅作为库存在，可以在单计算机上执行 MapReduce 任务，仅用于开发者搭建学习和试验环境。

伪集群模式：此模式 Hadoop 将以守护进程的形式在单机运行，一般用于开发者搭建学习和试验环境。

集群模式：此模式是 Hadoop 的生产环境模式，也就是说这才是 Hadoop 真正使用的模式，用于提供生产级服务。

HDFS 配置和启动

HDFS 和数据库相似，是以守护进程的方式启动的。使用 HDFS 需要用 HDFS 客户端通过网络 (套接字) 连接到 HDFS 服务器实现文件系统的使用。

在Hadoop 运行环境 一章，我们已经配置好了 Hadoop 的基础环境，容器名为 hadoop_single。如果你上次已经关闭了该容器或者关闭了计算机导致容器关闭，请启动并进入该容器。

进入该容器后，我们确认一下 Hadoop 是否存在：

```
hadoop version
```

如果结果显示出 Hadoop 版本号则表示 Hadoop 存在。

接下来我们将进入正式步骤。

新建 hadoop 用户

新建用户，名为 hadoop：

```
adduser hadoop
```

安装一个小工具用于修改用户密码和权限管理：

```
yum install -y passwd sudo
```

设置 hadoop 用户密码：

```
passwd hadoop
```

接下来两次输入密码，一定要记住！

修改 hadoop 安装目录所有人为 hadoop 用户：

```
chown -R hadoop /usr/local/hadoop
```

然后用文本编辑器修改 /etc/sudoers 文件，在

```
root    ALL=(ALL)       ALL
```

之后添加一行

```
hadoop  ALL=(ALL)       ALL
```

然后退出容器。

关闭并提交容器 hadoop_single 到镜像 hadoop_proto：

```
docker stop hadoop_single
docker commit hadoop_single hadoop_proto
```

创建新容器 hdfs_single ：

```
docker run -d --name=hdfs_single --privileged hadoop_proto /usr/sbin/init
```

这样新用户就被创建了。

启动 HDFS

现在进入刚建立的容器：

```
docker exec -it hdfs_single su hadoop
```

现在应该是 hadoop 用户：

```
whoami
```

应该显示 "hadoop"

生成 SSH 密钥：

```
ssh-keygen -t rsa
```

这里可以一直按回车直到生成结束。

然后将生成的密钥添加到信任列表：

```
ssh-copy-id hadoop@172.17.0.2
```

查看容器 IP 地址：

```
ip addr | grep 172
```

从而得知容器的 IP 地址是 172.17.0.2，你们的 IP 可能会与此不同。

在启动 HDFS 以前我们对其进行一些简单配置，Hadoop 配置文件全部储存在安装目录下的 etc/hadoop 子目录下，所以我们可以进入此目录：

```
cd $HADOOP_HOME/etc/hadoop
```

这里我们修改两个文件：core-site.xml 和 hdfs-site.xml

在 core-site.xml 中，我们在 标签下添加属性：

```
<property>
    <name>fs.defaultFS</name>
    <value>hdfs://<你的IP>:9000</value>
</property>
```

在 hdfs-site.xml 中的 标签下添加属性：

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

格式化文件结构：

```
hdfs namenode -format
```

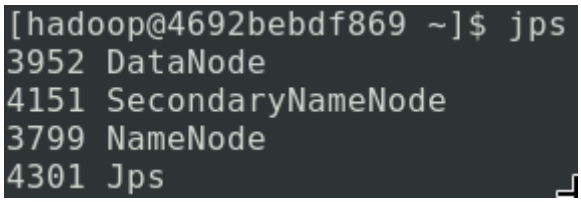
然后启动 HDFS：

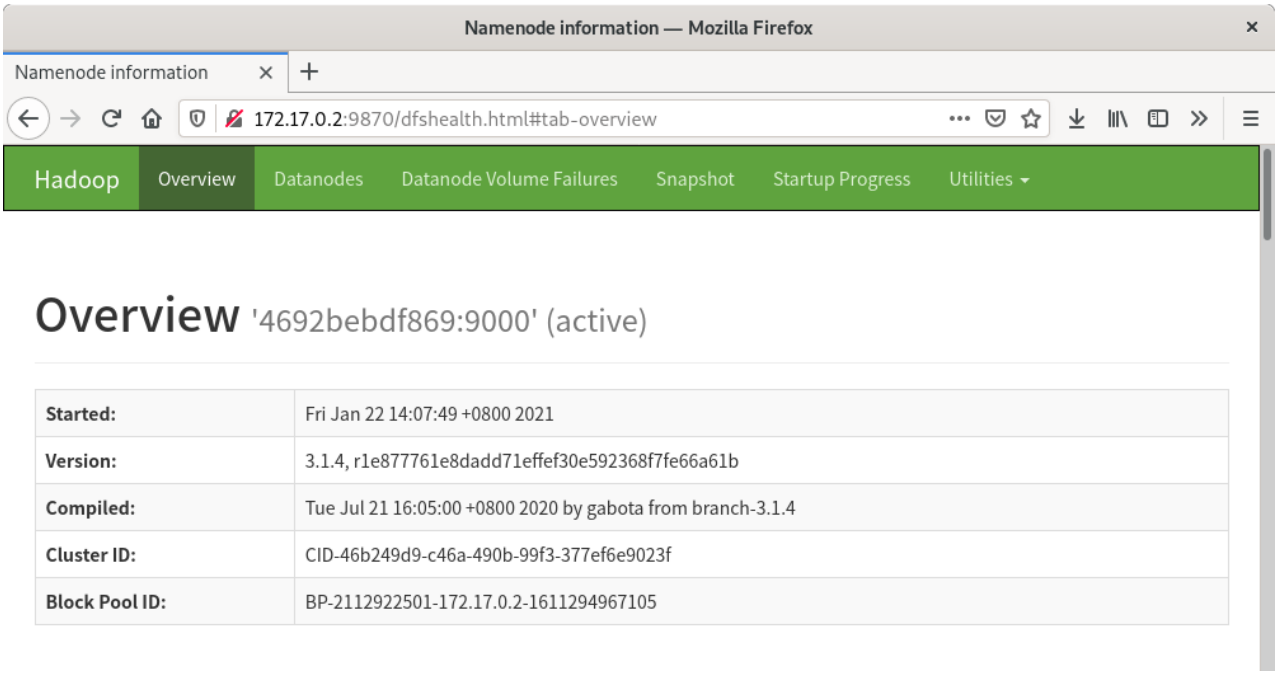
```
start-dfs.sh
```

启动分三个步骤，分别启动 NameNode、DataNode 和 Secondary NameNode。

我们可以运行 jps 来查看 Java 进程：

到此为止，HDFS 守护进程已经建立，由于 HDFS 本身具备 HTTP 面板，我们可以通过浏览器访问http://你的容器IP:9870/来查看 HDFS 面板以及详细信息：





如果出现这个页面，说明 HDFS 配置并启动成功。

注意：如果你使用的不是含有桌面环境的 Linux 系统，没有浏览器，可以跳过这个步骤。如果你使用的是 Windows 系统但是没有使用 Docker Desktop ，那么这个步骤对你来说将难以实现。

HDFS 使用

HDFS Shell

回到 hdfs_single 容器，以下命令将用于操作 HDFS：

```
# 显示根目录 / 下的文件和子目录，绝对路径
hadoop fs -ls /
# 新建文件夹，绝对路径
hadoop fs -mkdir /hello
# 上传文件
hadoop fs -put hello.txt /hello/
# 下载文件
hadoop fs -get /hello/hello.txt
# 输出文件内容
hadoop fs -cat /hello/hello.txt
```

HDFS 最基础的命令如上所述，除此之外还有许多其他传统文件系统所支持的操作。

HDFS API

HDFS 已经被很多的后端平台所支持，目前官方在发行版中包含了 C/C++ 和 Java 的编程接口。此外，node.js 和 Python 语言的包管理器也支持导入 HDFS 的客户端。

以下是包管理器的依赖项列表：

Maven：

```
    <dependency>
        <groupId>org.apache.hadoop</groupId>
        <artifactId>hadoop-client</artifactId>
        <version>3.1.4</version>
    </dependency>
```

Gradle：

```
providedCompile group: 'org.apache.hadoop', name: 'hadoop-hdfs-client', version:
'3.1.4'
```

NPM：

```
npm i webhdfs
```

pip：

```
pip install hdfs
```

这里提供一个 Java 连接 HDFS 的例子（别忘了修改 IP 地址）：

实例

```
package com.runoob;
import java.io.IOException;
import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.*;
public class Application {
   public static void main(String[] args) {
      try {
         // 配置连接地址
         Configuration conf = new Configuration();
         conf.set("fs.defaultFS", "hdfs://172.17.0.2:9000");
         FileSystem fs = FileSystem.get(conf);
         // 打开文件并读取输出
         Path hello = new Path("/hello/hello.txt");
         FSDataInputStream ins = fs.open(hello);
         int ch = ins.read();
         while (ch != -1) {
            System.out.print((char)ch);
            ch = ins.read();
         }
         System.out.println();
      } catch (IOException ioe) {
         ioe.printStackTrace();
      }
   }
}
```

HDFS 集群是建立在 Hadoop 集群之上的，由于 HDFS 是 Hadoop 最主要的守护进程，所以 HDFS 集群的配置过程是 Hadoop 集群配置过程的代表。

使用 Docker 可以更加方便地、高效地构建出一个集群环境。

每台计算机中的配置

Hadoop 如何配置集群、不同的计算机里又应该有怎样的配置，这些问题是在学习中产生的。本章的配置中将会提供一个典型的示例，但 Hadoop 复杂多样的配置项远超于此。

HDFS 命名节点对数据节点的远程控制是通过 SSH 来实现的，因此关键的配置项应该在命名节点被配置，非关键的节点配置要在各个数据节点配置。也就是说，数据节点与命名节点的配置可以不同，不同数据节点之间的配置也可以有所不同。

但是本章为了方便建立集群，将使用相同的配置文件通过 Docker 镜像的形式同步到所有的集群节点，特做解释。

具体步骤

总体思路是这样的，我们先用一个包含 Hadoop 的镜像进行配置，配置成集群中所有节点都可以共用的样子，然后再以它为原型生成若干个容器，构成一个集群。

配置原型

首先，我们将使用之前准备的 hadoop_proto 镜像启动为容器：

```
docker run -d --name=hadoop_temp --privileged hadoop_proto /usr/sbin/init
```

进入 Hadoop 的配置文件目录：

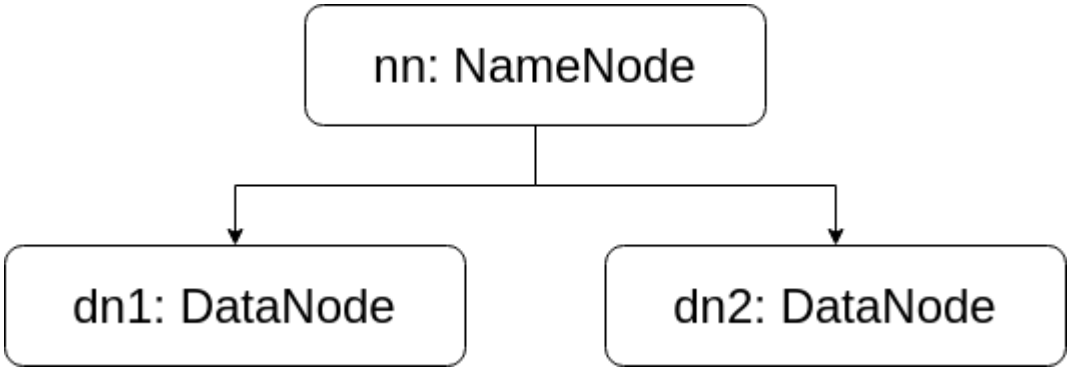
```
cd $HADOOP_HOME/etc/hadoop
```

现在对这里的文件的作用做简单的描述：

文件	作用
workers	记录所有的数据节点的主机名或 IP 地址
core-site.xml	Hadoop 核心配置
hdfs-site.xml	HDFS 配置项
mapred-site.xml	MapReduce 配置项
yarn-site.xml	YARN 配置项

注：YARN 的作用是为 MapReduce 提供资源管理服务，此处暂时用不着。
我们现在设计这样一个简单的集群：

- 1 个命名节点 nn
- 2 个数据节点 dn1, dn2



首先编辑 workers ，更改文件内容为：

```
dn1
dn2
```

然后编辑 core-site.xml，在 中添加以下配置项：

```
<!-- 配置 HDFS 主机地址与端口号 -->
<property>
    <name>fs.defaultFS</name>
    <value>hdfs://nn:9000</value>
</property>
<!-- 配置 Hadoop 的临时文件目录 -->
<property>
    <name>hadoop.tmp.dir</name>
    <value>file:///home/hadoop/tmp</value>
</property>
```

配置 hdfs-site.xml，在 中添加以下配置项：

```
<!-- 每个数据块复制 2 份存储 -->
<property>
    <name>dfs.replication</name>
    <value>2</value>
</property>

<!-- 设置储存命名信息的目录 -->
<property>
    <name>dfs.namenode.name.dir</name>
    <value>file:///home/hadoop/hdfs/name</value>
</property>
```

最后需要配置一下 SSH ：

```
ssh-keygen -t rsa -P "" -f ~/.ssh/id_rsa
ssh-copy-id -i ~/.ssh/id_rsa hadoop@localhost
```

到此为止，集群的原型就配置完毕了，可以退出容器并上传容器到新镜像 cluster_proto ：

```
docker stop hadoop_temp
docker commit hadoop_temp cluster_proto
```

此处如果有必要可以删除临时镜像 hadoop_temp 。

部署集群

接下来部署集群。

首先，要为 Hadoop 集群建立专用网络 hnet ：

```
docker network create --subnet=172.20.0.0/16 hnet
```

接下来创建集群容器：

```
docker run -d --name=nn --hostname=nn --network=hnet --ip=172.20.1.0 --add-
host=dn1:172.20.1.1 --add-host=dn2:172.20.1.2 --privileged cluster_proto
/usr/sbin/init
docker run -d --name=dn1 --hostname=dn1 --network=hnet --ip=172.20.1.1 --add-
host=nn:172.20.1.0 --add-host=dn2:172.20.1.2 --privileged cluster_proto
/usr/sbin/init
docker run -d --name=dn2 --hostname=dn2 --network=hnet --ip=172.20.1.2 --add-
host=nn:172.20.1.0 --add-host=dn1:172.20.1.1 --privileged cluster_proto
/usr/sbin/init
```

进入命名节点：

```
docker exec -it nn su hadoop
```

格式化 HDFS：

```
hdfs namenode -format
```

如果没有出错，那么下一步就可以启动 HDFS：

```
start-dfs.sh
```

成功启动之后，jps 命令应该能查到 NameNode 和 SecondaryNameNode 的存在。命名节点不存在 DataNode 进程，因为这个进程在 dn1 和 dn2 中运行。

至此，可以像上一章中讲述伪集群模式时所说的方法检测 HDFS 的运行，使用 HDFS 的方式也没有差别（命名节点代表整个集群）。

在学习了之前的 MapReduce 概念之后，我们应该已经知道什么是 Map 和 Reduce，并了解了他们的工作方式。

本章将学习如何使用 MapReduce。

Word Count

Word Count 就是"词语统计"，这是 MapReduce 工作程序中最经典的一种。它的主要任务是对一个文本文件中的词语作归纳统计，统计出每个出现过的词语一共出现的次数。

Hadoop 中包含了许多经典的 MapReduce 示例程序，其中就包含 Word Count。

注意：这个案例在 HDFS 不运行的状态下依然可以运行，所以我们先在单机模式下测试

首先，启动一个之前制作的 hadoop_proto 镜像的新容器：

```
docker run -d --name=word_count hadoop_proto
```

进入容器：

```
docker exec -it word_count bash
```

进入 HOME 目录：

```
cd ~
```

现在我们准备一份文本文件 input.txt：

```
I love runoob
I like runoob
I love hadoop
I like hadoop
```

将以上内容用文本编辑器保存。

执行 MapReduce：

```
hadoop jar $HADOOP_HOME/share/hadoop/mapreduce/hadoop-mapreduce-examples-3.1.4.jar
wordcount input.txt output
```

解释一下含义：

hadoop jar从 jar 文件执行 MapReduce 任务，之后跟着的是示例程序包的路径。

wordcount表示执行示例程序包中的 Word Count 程序，之后跟这两个参数，第一个是输入文件，第二个是输出结果的目录名（因为输出结果是多个文件）。

执行之后，应该会输出一个文件夹 output，在这个文件夹里有两个文件：_SUCCESS 和 part-r-ooooo。

其中 _SUCCESS 只是用于表达执行成功的空文件，part-r-ooooo 则是处理结果，当我们显示一下它的内容：

```
cat ~/output/part-r-00000
```

你应该可以看到如下信息：

```
I       4
hadoop  2
like    2
love    2
runoob  2
```

集群模式

现在我们在集群模式下运行 MapReduce。

启动在上一章配置好的集群容器：

```
docker start nn dn1 dn2
```

进入 NameNode 容器：

```
docker exec -it nn su hadoop
```

进入 HOME：

```
cd ~
```

编辑 input.txt：

```
I love runoob
I like runoob
I love hadoop
I like hadoop
```

启动 HDFS：

```
start-dfs.sh
```

创建目录：

```
hadoop fs -mkdir /wordcount
hadoop fs -mkdir /wordcount/input
```

上传 input.txt

```
hadoop fs -put input.txt /wordcount/input/
```

执行 Word Count：

```
hadoop jar $HADOOP_HOME/share/hadoop/mapreduce/hadoop-mapreduce-examples-3.1.4.jar
wordcount /wordcount/input /wordcount/output
```

查看执行结果：

```
hadoop fs -cat /wordcount/output/part-r-00000
```

如果一切正常，将会显示以下结果：

```
I       4
hadoop  2
like    2
love    2
runoob  2
```

7.0 MapReduce 编程

分类 Hadoop 教程

在学习了 MapReduce 的使用之后，我们已经可以处理 Word Count 这类统计和检索任务，但是客观上 MapReduce 可以做的事情还有很多。

MapReduce 主要是依靠开发者通过编程来实现功能的，开发者可以通过实现 Map 和 Reduce 相关的方法来进行数据处理。

为了简单的展示这一过程，我们将手工编写一个 Word Count 程序。

注意：MapReduce 依赖 Hadoop 的库，但由于本教程使用的 Hadoop 运行环境是 Docker 容器，难以部署开发环境，所以真实的开发工作（包含调试）将需要一个运行 Hadoop 的计算机。在这里我们仅学习已完成程序的部署。

MyWordCount.java 文件代码

```
/**
 * 引用声明
 * 本程序引用自 http://hadoop.apache.org/docs/r1.0.4/cn/mapred_tutorial.html
 */
package com.runoob.hadoop;
import java.io.IOException;
import java.util.*;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.*;
import org.apache.hadoop.mapred.*;
/**
 * 与 `Map` 相关的方法
 */
class Map extends MapReduceBase implements Mapper<LongWritable, Text, Text,
IntWritable> {
  private final static IntWritable one = new IntWritable(1);
  private Text word = new Text();
  public void map(LongWritable key,
            Text value,
            OutputCollector<Text, IntWritable> output,
            Reporter reporter)
        throws IOException {
    String line = value.toString();
```

```
    StringTokenizer tokenizer = new StringTokenizer(line);
    while (tokenizer.hasMoreTokens()) {
      word.set(tokenizer.nextToken());
      output.collect(word, one);
    }
  }
}
/**
 * 与 `Reduce` 相关的方法
 */
class Reduce extends MapReduceBase implements Reducer<Text, IntWritable, Text,
IntWritable> {
  public void reduce(Text key,
              Iterator<IntWritable> values,
              OutputCollector<Text, IntWritable> output,
              Reporter reporter)
      throws IOException {
    int sum = 0;
    while (values.hasNext()) {
      sum += values.next().get();
    }
    output.collect(key, new IntWritable(sum));
  }
}
public class MyWordCount {
  public static void main(String[] args) throws Exception {
    JobConf conf = new JobConf(MyWordCount.class);
    conf.setJobName("my_word_count");
    conf.setOutputKeyClass(Text.class);
    conf.setOutputValueClass(IntWritable.class);
    conf.setMapperClass(Map.class);
    conf.setCombinerClass(Reduce.class);
    conf.setReducerClass(Reduce.class);
    conf.setInputFormat(TextInputFormat.class);
    conf.setOutputFormat(TextOutputFormat.class);
    // 第一个参数表示输入
    FileInputFormat.setInputPaths(conf, new Path(args[0]));
    // 第二个输入参数表示输出
    FileOutputFormat.setOutputPath(conf, new Path(args[1]));
    JobClient.runJob(conf);
  }
}
```

请将此 Java 文件的内容保存到 NameNode 容器中去，建议位置：

```
/home/hadoop/MyWordCount/com/runoob/hadoop/MyWordCount.java
```

注意：根据当前情况，有的 Docker 环境中安装的 JDK 不支持中文，所以保险起见，请去掉以上代码中的中文注释。

进入目录：

```
cd /home/hadoop/MyWordCount
```

编译：

```
javac -classpath ${HADOOP_HOME}/share/hadoop/mapreduce/hadoop-mapreduce-client-core-3.1.4.jar -classpath ${HADOOP_HOME}/share/hadoop/client/hadoop-client-api-3.1.4.jar com/runoob/hadoop/MyWordCount.java
```

打包：

```
jar -cf my-word-count.jar com
```

执行：

```
hadoop jar my-word-count.jar com.runoob.hadoop.MyWordCount /wordcount/input /wordcount/output2
```

查看结果：

```
hadoop fs -cat /wordcount/output2/part-00000
```

输出：

```
I       4
hadoop  2
like    2
love    2
runoob  2
```