Hadoop

第一章 HDFS

1 HDFS简介

Hadoop 分布式系统框架中，首要的基础功能就是文件系统，在 Hadoop 中使用 `FileSystem` 这个抽象类来表示我们的文件系统，这个抽象类下面有很多子实现类，究竟使用哪一种，需要看我们具体的实现类，在我们实际工作中，用到的最多的就是HDFS(分布式文件系统)以及LocalFileSystem(本地文件系统)了。

在现代的企业环境中，单机容量往往无法存储大量数据，需要跨机器存储。统一管理分布在集群上的文件系统称为**分布式文件系统。**

`HDFS（Hadoop Distributed File System）` 是 Hadoop 项目的一个子项目。是 Hadoop 的核心组件之一，Hadoop 非常适于存储大型数据 (比如 TB 和 PB)，其就是使用 HDFS 作为存储系统. HDFS 使用多台计算机存储文件，并且提供统一的访问接口，像是访问一个普通文件系统一样使用分布式文件系统。

2 HDFS常用shell命令

显示当前目录结构

```
# 显示当前目录结构
hadoop fs -ls  <path>
# 递归显示当前目录结构
hadoop fs -ls   -R  <path>
# 显示根目录下内容
hadoop fs -ls  /
```

创建目录

```
# 创建目录
hadoop fs -mkdir  <path>
# 递归创建目录
hadoop fs -mkdir -p  <path>
```

删除操作

```
# 删除文件
hadoop fs -rm  <path>
# 递归删除目录和文件
hadoop fs -rm -R  <path>
```

从本地加载文件到 HDFS

```
# 二选一执行即可
hadoop fs -put  [localsrc] [dst]
hadoop fs - copyFromLocal [localsrc] [dst]
```

从 HDFS 导出文件到本地

```
# 二选一执行即可
hadoop fs -get  [dst] [localsrc]
hadoop fs -copyToLocal [dst] [localsrc]
```

查看文件内容

```
# 二选一执行即可
hadoop fs -text  <path>
hadoop fs -cat  <path>
```

显示文件的最后一千字节

```
hadoop fs -tail  <path>
# 和Linux下一样，会持续监听文件内容变化 并显示文件的最后一千字节
hadoop fs -tail -f  <path>
```

拷贝文件

```
hadoop fs -cp [src] [dst]
```

移动文件

```
hadoop fs -mv [src] [dst]
```

统计当前目录下各文件大小

- 默认单位字节
- -s : 显示所有文件大小总和，
- -h : 将以更友好的方式显示文件大小（例如 64.0m 而不是 67108864）

```
hadoop fs -du  <path>
```

合并下载多个文件

- -nl 在每个文件的末尾添加换行符（LF）
- -skip-empty-file 跳过空文件

```
hadoop fs -getmerge
# 示例 将HDFS上的hbase-policy.xml和hbase-site.xml文件合并后下载到本地的/usr/test.xml
hadoop fs -getmerge -nl  /test/hbase-policy.xml /test/hbase-site.xml /usr/test.xml
```

统计文件系统的可用空间信息

```
hadoop fs -df -h /
```

更改文件复制因子

```
hadoop fs -setrep [-R] [-w] <numReplicas> <path>
```

- 更改文件的复制因子。如果 path 是目录，则更改其下所有文件的复制因子
- -w : 请求命令是否等待复制完成

```
# 示例
hadoop fs -setrep -w 3 /user/hadoop/dir1
```

权限控制

```
# 权限控制和Linux上使用方式一致
# 变更文件或目录的所属群组。 用户必须是文件的所有者或超级用户。
hadoop fs -chgrp [-R] GROUP URI [URI ...]
# 修改文件或目录的访问权限  用户必须是文件的所有者或超级用户。
hadoop fs -chmod [-R] <MODE[,MODE]... | OCTALMODE> URI [URI ...]
# 修改文件的拥有者  用户必须是超级用户。
hadoop fs -chown [-R] [OWNER][:[GROUP]] URI [URI ]
```

文件检测

```
hadoop fs -test - [defsz]  URI
```

可选选项：

- -d：如果路径是目录，返回0。
- -e：如果路径存在，则返回0。
- -f：如果路径是文件，则返回0。
- -s：如果路径不为空，则返回0。
- -r：如果路径存在且授予读权限，则返回0。
- -w：如果路径存在且授予写入权限，则返回0。
- -z：如果文件长度为零，则返回0。

3 HDFS的JavaAPI

看看就好，感觉很少用java来写，都是直接操作的，敲敲指令不比写这个香？？？

```
package HDFS;

import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.*;
import org.apache.hadoop.fs.permission.FsAction;
import org.apache.hadoop.fs.permission.FsPermission;
import org.apache.hadoop.io.IOUtils;
import org.apache.hadoop.util.Progressable;
import org.junit.After;
import org.junit.Before;
import org.junit.Test;
import java.io.*;
import java.net.URI;
import java.net.URISyntaxException;

/**
 * Project:  BigDataCode
 * Create date:  2023/5/6
 * Created by fujiahao
 */

/**
 * 需求：Hadoop的HDFS-JAVA-API
 *       1.FileSystem
 *       2.创建目录
 *       3.创建指定权限的目录
 *       4.创建文件，并写入内容
 *       5.判断文件是否存在
 *       6.查看文件内容
 *       7.文件重命名
 *       8.删除目录或文件
```

```
 *        9.上传文件到HDFS
 *        10.上传大文件并显示上传进度
 *        11.从HDFS上下载文件
 *        12.查看指定目录下所有文件的信息
 *        13.递归查看指定目录下所有文件的信息
 *        14.查看文件的块信息
 */


public class HdfsClient {

      /**
       * FileSystem是所有HDFS的操作主入口
       * 使用@Before进行标注
       */

      private static FileSystem fileSystem;
      @Before
      public void prepare() {
            try {
                  Configuration configuration = new Configuration();
                  // 单节点Hadoop，副本系数设置为1，默认为3
                  configuration.set("dfs.replication", "1");
                  fileSystem = FileSystem.get(new URI("hdfs://master:9000"), configuration,
"root");
            } catch (InterruptedException e) {
                  e.printStackTrace();
            } catch (IOException e) {
                  e.printStackTrace();
            } catch (URISyntaxException e) {
                  e.printStackTrace();
            }
      }

      @After
      public void destroy() {
            fileSystem = null;
      }


      /**
       * 创建目录
       */
      @Test
      public void mkDir() throws Exception {
            fileSystem.mkdirs(new Path(""));
      }


      /**
       * 创建指定权限的目录
       */
      public void mkDirWithPermission() throws Exception {
            fileSystem.mkdirs(new Path(""),
                        new FsPermission(FsAction.READ_WRITE, FsAction.READ,
FsAction.READ));
      }

```

```
88
89          /**
90           * 创建文件，并写入内容
91           */
92          @Test
93          public void create() throws Exception {
94                // 如果文件存在，默认会覆盖，可以通过第二个参数进行控制
95                // 第三个参数可以控制使用缓冲区的大小
96                FSDataOutputStream out = fileSystem.create(new Path(""), true, 4096);
97                out.write("hello hadoop!".getBytes());
98                out.write("hello spark!".getBytes());
99                out.write("hello flink!".getBytes());
100               // 强制将缓冲区中内容刷出
101               out.flush();
102               out.close();
103         }
104
105
106         /**
107          * 判断文件是否存在
108          */
109         @Test
110         public void exist() throws Exception {
111               boolean exists = fileSystem.exists(new Path(""));
112               System.out.println(exists);
113         }
114
115
116         /**
117          * 查看文件内容(小文本文件内容，直接转换成字符串后输出)
118          */
119
120         @Test
121         public void readToString() throws Exception {
122               FSDataInputStream inputStream = fileSystem.open(new Path(""));
123               String context = inputStreamToString(inputStream, "utf-8");
124               System.out.println(context);
125         }
126
127         /**
128          * 自定义inputStreamToString方法
129          * @param inputStream 输入流
130          * @param encode 指定编码类型
131          * @return 返回内容
132          */
133         private static String inputStreamToString(InputStream inputStream, String encode) {
134               try {
135                     if (encode == null || ("".equals(encode))) {
136                           encode = "utf-8";
137                     }
138                     BufferedReader reader = new BufferedReader(new InputStreamReader(inputStream,
    encode));
139                     StringBuilder builder = new StringBuilder();
140                     String str = "";
141                     while ((str = reader.readLine()) != null) {
142                           builder.append(str).append("\n");
143                     }
144                     return builder.toString();
```

```
145             } catch (IOException e) {
146                   e.printStackTrace();
147             }
148              return null;
149       }
150
151
152        /**
153         * 文件重命名
154         */
155        @Test
156        public void rename() throws Exception {
157              Path oldPath = new Path("");
158              Path newPath = new Path("");
159              boolean result = fileSystem.rename(oldPath, newPath);
160              System.out.println(result);
161       }
162
163
164        /**
165         * 删除目录或文件
166         */
167        @Test
168        public void delete() throws Exception {
169              // 第二个参数代表是否递归删除
170              // 如果path是一个目录，递归删除为true，则删除该目录及所有文件
171              // 如果path是一个目录，递归删除为false，则会抛出异常
172              boolean result = fileSystem.delete(new Path(""), true);
173              System.out.println(result);
174       }
175
176
177        /**
178         * 上传文件到HDFS
179         */
180        @Test
181        public void copyFromLocalFile() throws Exception {
182              // 如果指定的是目录，则会把目录及其中的文件都复制到指定目录下
183              Path src = new Path("");
184              Path dst = new Path("");
185              fileSystem.copyFromLocalFile(src, dst);
186       }
187
188
189        /**
190         * 上传大文件并显示上传进度
191         */
192        @Test
193        public void copyFromLocalBigFile() throws Exception {
194              File file = new File("");
195              final float fileSize = file.length();
196              InputStream in = new BufferedInputStream(new FileInputStream(file));
197
198              FSDataOutputStream out = fileSystem.create(new Path(""),
199                          new Progressable() {
200                                long fileCount = 0;
201
202                                @Override
```

```
                                public void progress() {
                                      fileCount++;
                                      // progress: 每次上传64KB数据后就会被调用一次
                                      System.out.println("上传进度: " + (fileCount * 64 * 1024 / fileSize) * 100 + " %");
                                }
                          });
              IOUtils.copyBytes(in, out, 4096);
        }


        /**
         * 从HDFS上下载文件
         */
        @Test
        public void copyToLocalFile() throws Exception {
              Path src = new Path("");
              Path dst = new Path("");
              // 第一个参数: 下载完成后是否删除原文件, 默认是true
              // 最后一个参数: 是否将RawLocalFileSystem用作本地文件系统
              fileSystem.copyToLocalFile(false, src, dst, true);
        }


        /**
         * 查看指定目录下所有文件的信息
         */
        @Test
        public void listFiles() throws Exception {
              FileStatus[] statuses = fileSystem.listStatus(new Path(""));
              for (FileStatus fileStatus : statuses) {
                    // fileStatus的toString方法被重写过, 直接打印可以看到所有信息
                    System.out.println(fileStatus.toString());
              }
        }


        /**
         * 递归查看指定目录下所有文件的信息
         */
        @Test
        public void listFilesRecursive() throws Exception {
              RemoteIterator<LocatedFileStatus> files = fileSystem.listFiles(new Path(""), true);
              while (files.hasNext()) {
                    System.out.println(files.next());
              }
        }


        /**
         * 查看文件的块信息
         */
        @Test
        public void getFileBlockLocations() throws Exception {
              FileStatus fileStatus = fileSystem.getFileStatus(new Path(""));
              BlockLocation[] blocks = fileSystem.getFileBlockLocations(fileStatus, 0, fileStatus.getLen());
              for (BlockLocation block : blocks) {
```

```
259                     System.out.println(block);
260                 }
261             }
262 
263     }
```

第二章 MapReduce

1 MapReduce 介绍

MapReduce思想在生活中处处可见。或多或少都曾接触过这种思想。MapReduce的思想核心是“分而治之”，适用于大量复杂的任务处理场景（大规模数据处理场景）。即使是发布过论文实现分布式计算的谷歌也只是实现了这种思想，而不是自己原创。

- Map负责“分”，即把复杂的任务分解为若干个“简单的任务”来并行处理。可以进行拆分的前提是这些小任务可以并行计算，彼此间几乎没有依赖关系。
- Reduce负责“合”，即对map阶段的结果进行全局汇总。
- MapReduce运行在yarn集群
    1. ResourceManager
    2. NodeManager

这两个阶段合起来正是MapReduce思想的体现。

还有一个比较形象的语言解释MapReduce:

我们要数图书馆中的所有书。你数1号书架，我数2号书架。这就是“Map”。我们人越多，数书就更快。

现在我们到一起，把所有人的统计数加在一起。这就是“Reduce”。

2 MapReduce 设计构思

MapReduce是一个分布式运算程序的编程框架，核心功能是将用户编写的业务逻辑代码和自带默认组件整合成一个完整的分布式运算程序，并发运行在Hadoop集群上。

既然是做计算的框架，那么表现形式就是有个输入（input），MapReduce操作这个输入（input），通过本身定义好的计算模型，得到一个输出（output）。

对许多开发者来说，自己完完全全实现一个并行计算程序难度太大，而MapReduce就是一种简化并行计算的编程模型，降低了开发并行应用的入门门槛。

Hadoop MapReduce构思体现在如下的三个方面：

`如何对付大数据处理：分而治之`

对相互间不具有计算依赖关系的大数据，实现并行最自然的办法就是采取分而治之的策略。并行计算的第一个重要问题是如何划分计算任务或者计算数据以便对划分的子任务或数据块同时进行计算。不可分拆的计算任务或相互间有依赖关系的数据无法进行并行计算！

`构建抽象模型：Map和Reduce`

MapReduce借鉴了函数式语言中的思想，用Map和Reduce两个函数提供了高层的并行编程抽象模型。

Map: 对一组数据元素进行某种重复式的处理；

Reduce: 对Map的中间结果进行某种进一步的结果整理。

MapReduce中定义了如下的Map和Reduce两个抽象的编程接口，由用户去编程实现:

map: (k1; v1) → [(k2; v2)]

reduce: (k2; [v2]) → [(k3; v3)]

Map和Reduce为程序员提供了一个清晰的操作接口抽象描述。通过以上两个编程接口，大家可以看出MapReduce处理的数据类型是<key,value>键值对。

`MapReduce框架结构`

一个完整的mapreduce程序在分布式运行时有三类实例进程：

- MR AppMaster：负责整个程序的过程调度及状态协调；
- MapTask：负责map阶段的整个数据处理流程；
- ReduceTask：负责reduce阶段的整个数据处理流程。

3 WordCount示例编写

需求：在一堆给定的文本文件中统计输出每一个单词出现的总次数

node01服务器执行以下命令，准备数，数据格式准备如下：

```
cd /export/servers
vim wordcount.txt

#添加以下内容:
hello hello
world world
hadoop hadoop
hello world
hello flume
hadoop hive
hive kafka
flume storm
hive oozie
```

```
将数据文件上传到hdfs上面去
hdfs dfs -mkdir /wordcount/
hdfs dfs -put wordcount.txt /wordcount/
```

- 定义一个mapper类

```
import org.apache.hadoop.io.LongWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Mapper;

import java.io.IOException;

// mapper程序:  需要继承 mapper类, 需要传入 四个类型:
/*  在hadoop中, 对java的类型都进行包装, 以提高传输的效率  writable
      keyin :  k1    Long       ---- LongWritable
      valin :  v1    String     ------ Text
      keyout : k2    String     ------- Text
      valout : v2    Long       -------LongWritable

 */

```

```
public class MapTask extends Mapper<LongWritable,Text,Text,LongWritable> {

      /**
       *
       * @param key  : k1
       * @param value    v1
       * @param context  上下文对象    承上启下功能
       * @throws IOException
       * @throws InterruptedException
       */
      @Override
      protected void map(LongWritable key, Text value, Context context) throws IOException,
InterruptedException {
            //1. 获取 v1 中数据
            String val = value.toString();

            //2. 切割数据
            String[] words = val.split(" ");

            Text text = new Text();
            LongWritable longWritable = new LongWritable(1);
            //3. 遍历循环, 发给 reduce
            for (String word : words) {
                  text.set(word);
                  context.write(text,longWritable);
            }
      }
}
```

- 定义一个reducer类

```
import org.apache.hadoop.io.LongWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Reducer;

import java.io.IOException;

/**
 * KEYIN  :  k2     -----Text
 * VALUEIN :  v2    ------LongWritable
 * KEYOUT  : k3     ------  Text
 * VALUEOUT : v3    ------ LongWritable
 */
public class ReducerTask extends Reducer<Text, LongWritable, Text, LongWritable> {


      @Override
      protected void reduce(Text key, Iterable<LongWritable> values, Context context) throws
IOException, InterruptedException {

            //1. 遍历 values 获取每一个值
            long   v3 = 0;
            for (LongWritable longWritable : values) {

                  v3 += longWritable.get();   //1
            }

            //2. 输出
```

```
            context.write(key,new LongWritable(v3));

        }
    }
```

- 定义一个主类，用来描述job并提交job

```
import com.sun.org.apache.bcel.internal.generic.NEW;
import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.conf.Configured;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.LongWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.io.nativeio.NativeIO;
import org.apache.hadoop.mapreduce.Job;
import org.apache.hadoop.mapreduce.lib.input.TextInputFormat;
import org.apache.hadoop.mapreduce.lib.output.TextOutputFormat;
import org.apache.hadoop.util.Tool;
import org.apache.hadoop.util.ToolRunner;

// 任务的执行入口：将八步组合在一起
public class JobMain extends Configured implements Tool {
      // 在run方法中编写组装八步
      @Override
      public int run(String[] args) throws Exception {

            Job job = Job.getInstance(super.getConf(), "JobMain");
            //如果提交到集群操作. 需要添加一步 : 指定入口类
            job.setJarByClass(JobMain.class);


            //1. 封装第一步:  读取数据
            job.setInputFormatClass(TextInputFormat.class);
            TextInputFormat.addInputPath(job,new Path("hdfs://node01:8020/wordcount.txt"));

            //2. 封装第二步:  自定义 map程序
            job.setMapperClass(MapTask.class);
            job.setMapOutputKeyClass(Text.class);
            job.setMapOutputValueClass(LongWritable.class);

            //3. 第三步 第四步 第五步 第六步 省略

            //4. 第七步:  自定义reduce程序
            job.setReducerClass(ReducerTask.class);
            job.setOutputKeyClass(Text.class);
            job.setOutputValueClass(LongWritable.class);

            //5) 第八步  : 输出路径是一个目录, 而且这个目录必须不存在的
            job.setOutputFormatClass(TextOutputFormat.class);
            TextOutputFormat.setOutputPath(job,new Path("hdfs://node01:8020/output"));

            //6) 提交任务:
            boolean flag = job.waitForCompletion(true); // 成功  true  不成功 false

            return flag ? 0 : 1;
      }

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

```
52              Configuration configuration = new Configuration();
53              JobMain jobMain = new JobMain();
54              int i = ToolRunner.run(configuration, jobMain, args); //返回值 退出码
55
56              System.exit(i); // 退出程序  0 表示正常  其他值表示有异常 1
57          }
58      }
```

提醒：代码开发完成之后，就可以打成jar包放到服务器上面去运行了，实际工作当中，都是将代码打成jar包，开发main方法作为程序的入口，然后放到集群上面去运行。

第三章 Yarn

yarn的架构和原理

yarn的基本介绍和产生背景

YARN是Hadoop2引入的通用的资源管理和任务调度的平台，可以在YARN上运行MapReduce、Tez、Spark等多种计算框架，只要计算框架实现了YARN所定义的接口，都可以运行在这套通用的Hadoop资源管理和任务调度平台上。

Hadoop 1.0是由HDFS和MapReduce V1组成的，YARN出现之前是MapReduce V1来负责资源管理和任务调度，MapReduce V1由JobTracker和TaskTracker两部分组成。

MapReduce V1有如下缺点：

1. 扩展性差：

   在MapReduce V1中，JobTracker同时负责资源管理和任务调度，而JobTracker只有一个节点，所以JobTracker成为了制约系统性能的一个瓶颈，制约了Hadoop平台的扩展性。

2. 可靠性低：

   MapReduce V1中JobTracker存在单点故障问题，所以可靠性低。

3. 资源利用率低：

   MapReduce V1采用了基于槽位的资源分配模型，槽位是一种粗粒度的资源划分单位。

- 一是通常情况下为一个job分配的槽位不会被全部利用。
- 二是一个MapReduce任务的Map阶段和Reduce阶段会划分了固定的槽位，并且不可以共用，很多时候一种类型的槽位资源很紧张而另外一种类型的槽位很空闲，导致资源利用率低。

1. 不支持多种计算框架

   MapReduce V1这种资源管理和任务调度方式只适合MapReduce这种计算框架，而MapReduce这种离线计算框架很多时候不能满足应用需求。

yarn的优点：

1. 支持多种计算框架

   YARN是通用的资源管理和任务调度平台，只要实现了YARN的接口的计算框架都可以运行在YARN上。

2. 资源利用率高

   多种计算框架可以共用一套集群资源，让资源充分利用起来，提高了利用率。

3. 运维成本低

避免一个框架一个集群的模式，YARN降低了集群的运维成本。

4. 数据可共享

共享集群模式可以让多种框架共享数据和硬件资源，减少数据移动带来的成本。

hadoop 1.0 和 hadoop 2.0 的区别

1. 组成部分

Hadoop1.0由HDFS和MapReduce组成，Hadoop2.0由HDFS和YARN组成。

1. HDFS可扩展性

Hadoop1.0中的HDFS只有一个NameNode，制约着集群文件个数的增长，Hadoop2.0增加了HDFS联盟的架构，可以将NameNode所管理的NameSpace水平划分，增加了HDFS的可扩展性。

1. HDFS的可靠性

Hadoop1.0中的HDFS只有一个NameNode，存在着单点故障的问题，Hadoop2.0提供了HA的架构，可以实现NameNode的热备份和热故障转移，提高了HDFS的可靠性。

1. 可支持的计算框架

Hadoop1.0中只支持MapReduce一种计算框架，Hadoop2.0因为引入的YARN这个通用的资源管理与任务调度平台，可以支持很多计算框架了。

1. 资源管理和任务调度

Hadoop1.0中资源管理和任务调度依赖于MapReduce中的JobTracker，JobTracker工作很繁重，很多时候会制约集群的性能。

Hadoop2.0中将资源管理任务分给了YARN的ResourceManage，将任务调度分给了YARN的ApplicationMaster。

yarn 集群的架构和工作原理

YARN的基本设计思想是将MapReduce V1中的JobTracker拆分为两个独立的服务：ResourceManager和ApplicationMaster。ResourceManager负责整个系统的资源管理和分配，ApplicationMaster负责单个应用程序的的管理。

1. **ResourceManager**

RM是一个全局的资源管理器，负责整个系统的资源管理和分配，它主要由两个部分组成：调度器（Scheduler）和应用程序管理器（Application Manager）。

调度器根据容量、队列等限制条件，将系统中的资源分配给正在运行的应用程序，在保证容量、公平性和服务等级的前提下，优化集群资源利用率，让所有的资源都被充分利用 。

应用程序管理器负责管理整个系统中的所有的应用程序，包括应用程序的提交、与调度器协商资源以启动ApplicationMaster、监控ApplicationMaster运行状态并在失败时重启它。

1. **ApplicationMaster**

用户提交的一个应用程序会对应于一个ApplicationMaster，它的主要功能有：

- 与RM调度器协商以获得资源，资源以Container表示。
- 将得到的任务进一步分配给内部的任务。
- 与NM通信以启动/停止任务。
- 监控所有的内部任务状态，并在任务运行失败的时候重新为任务申请资源以重启任务。

1. **nodeManager**

NodeManager是每个节点上的资源和任务管理器，一方面，它会定期地向RM汇报本节点上的资源使用情况和各个Container的运行状态；另一方面，他接收并处理来自AM的Container启动和停止请求。

1. **container**

Container是YARN中的资源抽象，封装了各种资源。一个应用程序会分配一个Container，这个应用程序只能使用这个Container中描述的资源。

不同于MapReduceV1中槽位slot的资源封装，Container是一个动态资源的划分单位，更能充分利用资源。

yarn 的任务提交流程

当jobclient向YARN提交一个应用程序后，YARN将分两个阶段运行这个应用程序：一是启动ApplicationMaster;第二个阶段是由ApplicationMaster创建应用程序，为它申请资源，监控运行直到结束。

具体步骤如下:

1. 用户向YARN提交一个应用程序，并指定ApplicationMaster程序、启动ApplicationMaster的命令、用户程序。
2. RM为这个应用程序分配第一个Container，并与之对应的NM通讯，要求它在这个Container中启动应用程序ApplicationMaster。
3. ApplicationMaster向RM注册，然后拆分为内部各个子任务，为各个内部任务申请资源，并监控这些任务的运行，直到结束。
4. AM采用轮询的方式向RM申请和领取资源。
5. RM为AM分配资源，以Container形式返回
6. AM申请到资源后，便与之对应的NM通讯，要求NM启动任务。
7. NodeManager为任务设置好运行环境，将任务启动命令写到一个脚本中，并通过运行这个脚本启动任务
8. 各个任务向AM汇报自己的状态和进度，以便当任务失败时可以重启任务。
9. 应用程序完成后，ApplicationMaster向ResourceManager注销并关闭自己