1. **HDFS/MapReduce/Spark/YARN/Flink架构中主要部件及负责功能**

**HDFS:**

日程表

描述已自动生成

* NameNode
  + 负责HDFS的管理工作，包括管理文件目录结构、位置等元数据、维护 DataNode的状态等
  + 并不实际存储文件
* SecondaryNameNode – NameNode的备份
  + 充当NameNode的备份
  + 一旦NameNode发生故障时利用Secondary NameNode进行恢复
* DataNode
  + 负责数据块的存储
  + 为客户端提供实际的文件数据

**MapReduce:**

日程表

中度可信度描述已自动生成

JobTracker:

* 资源管理
  + 通过监控TaskTracker来管理系统拥有的计算资源
* 作业管理
  + 负责将作业（Job）拆分成任务（Task），并进行任务调度以及跟踪任务的运行进度、资源使用量等信息

TaskTracker:

* 管理本节点的资源
  + TaskTracker使用slot等量划分本节点上的资源量（CPU、内存等）
* 执行JobTracker的命令
  + 接收JobTracker发送过来的命令并执行（如启动新Task、杀死Task等）
* 向JobTracker汇报情况
  + 通过心跳将本节点上资源使用情况和任务运行进度汇报给JobTracker

Task:

* 任务执行
  + JobTracker根据TaskTracker汇报的信息进行调度，命令存在空闲slot的TaskTracker启动Task进程执行map或reduce任务
  + 在Hadoop MapReduce的实现中该进程的名称为Child

Client:

* 提交作业
  + 用户编写的MapReduce程序通过Client提交到JobTracker
  + 用户可通过Client提供的一些接口查看作业运行状态
  + 在Hadoop MapReduce的实现中，该进程的名称为RunJar

**Spark:**

**Standalone:**

图片包含 图示

描述已自动生成

Cluster Manager:集群管理器：负责管理整个系统的资源、监控工作节点

Executor:执行器：负责任务执行

* Executor是运行在工作节点上的一个进程，它启动若干个线程Task或线程组TaskSet来进行执行任务
* 在Standalone部署方式下，Executor进程的名称为CoarseGrainedExecutorBackend

Driver: 驱动器：负责启动应用程序的主方法并管理作业运行

* Spark的架构实现了资源管理和作业管理两大功能的分离
  + Cluster Manager负责集群资源管理
  + Driver负责作业管理

**Yarn:**

**图示

描述已自动生成**

ResourceManager:

* 资源管理器：负责整个系统的资源管理和分配
  + 资源调度器(Resource Scheduler)：分配Container并进行资源调度
  + 应用程序管理器(Application Manager)：管理整个系统中运行的所有应用
    - 应用程序提交
    - 与调度器协商资源以启动ApplicationMaster
    - 监控ApplicationMaster运行状态

NodeManager:

* 节点管理器：负责每个节点资源和任务管理
  + 定时地向RM汇报本节点的资源使用情况和Container运行状态
  + 接受并处理来自AM的Container启动/停止等各种请求

ApplicationMaster:

* 当用户基于Yarn平台提交一个框架应用，Yarn均启动一个 AM用于管理该应用
  + AM与RM调度器协商以获取资源（以Container表示），将获取的资源进一步分配给应用内部的任务
  + AM与NM通信以启动/停止任务，监控所有任务运行状态， 并在任务发生故障时重新申请资源来重启任务

Container:

* Container是资源的抽象表示，包含CPU、内存等资源，是一个动态资源划分单位
* 当AM向RM申请资源时，RM向AM返回以Container表示的资源

**Flink:**

**图示

描述已自动生成**

Client:

* 客户端
  + 将用户编写的DataStream程序翻译为逻辑执行图并进行优化，并将优化后的逻辑执行图提交到JobManager
  + Standalone模式下，Client的进程名为CliFrontend

JobManager:

* 作业管理器
  + 根据逻辑执行图产生物理执行图，负责协调系统的作业执行 ，包括任务调度，协调检查点和故障恢复等
  + Standalone模式下
    - JobManager还负责Flink系统的资源管理
    - JobManager的进程名为StandaloneSessionClusterEntrypoint

TaskManager:

* 任务管理器
  + 用来执行JobManager分配的任务，并且负责读取数据、缓存数据以及与其它TaskManager进行数据传输
  + Standalone模式下
    - TaskManager还负责所在节点的资源管理，将内存等资源抽象成若干个TaskSlot用于任务的执行，
    - TaskManager的进程名为TaskManagerRunner

1. **Spark系统中Standalone Client/Cluster模式的驱动器组件**

图示

描述已自动生成

图示

描述已自动生成

**3. MapReduce中的shuffle过程顺序**

**MapReduce:**

* List([k2, v2]) → [k2, List(v2)]

形状, 箭头

描述已自动生成



何时Shuffle？

* 当系统中的Map任务完成率达到设定阈值时，系统将启动Reduce任务
  + 例如，阈值设定为60%意味着如果系统中共有100个Map任务，那么一旦有60个Map任务已经完成了就可以启动Reduce任务，而不必等到这100个Map任务全部完成
* Reduce任务不会等到所有的Map任务执行结束才拉取Map任务的输出结果，但是拉取的数据必然来自于已经完成运行的Map任务，即已经保存在磁盘上的文件

**Spark中Stage之间的Shuffle:**

* Stage之间的数据传输需要进行Shuffle，该过程与MapReduce中的Shuffle类似
* Shuffle过程可能发生在两个ShuffleMapStage之间，或者ShuffleMapStage与ResultStage之间
* 从Task的层面来看，该过程表现为两组ShuffleMapTask之间，或一组ShuffleMapTask与一组ResultTask之间的数据传输

1. **在MapReduce/Spark/Flink系统中数据类型及操作算子**

**MapReduce：**

* 数据模型：将数据抽象为一系列键值对，在处理过程中对键值对进行转换
* 操作算子：map、reduce

**Spark：**

* 数据模型: RDD
* RDD操作算子：

创建：从本地内存或外部数据源创建RDD，提供了数据输入的功能

转换(Transformation)：描述RDD的转换逻辑，提供对RDD进行变换的功能

行动(Action)：标志转换结束，触发DAG生成

**Flink:**

* 数据模型：DataStream
* 操作算子对DataStream进行变换
  + DataStream经变换操作得到新的DataStream
  + 与Spark中RDD经变换生成新的RDD类似
* 一系列的变换操作构成一张有向无环图，即描述计算过程的DAG
  + 数据源（DataSource）: 描述DataStream数据的来源
  + 转换（Transformation）: 描述DataStream在系统中的转换逻辑
  + 数据池（DataSink）: 描述DataStream数据的走向，标志着计算DAG的结束

1. **YARN中的Container重启功能机制**

Container是资源的抽象表示，包含CPU、内存等资源，是一个动态资源划分单位

当AM向RM申请资源时，RM向AM返回以Container表示的资源

手机截图图有时间和文字

描述已自动生成

1. **大数据中的特性及挑战？**

数据量大：要求大数据管理系统能够存储和管理TB甚至PB级别的数据

数据种类多：要求大数据管理系统能够在同一平台处理多种类型的数据

产生速度快：要求大数据管理系统能够在数据产生的过程中实时处理数据

潜在价值高：要求大数据管理系统能够支持使用机器学习、数据挖掘或人工智能等方法进行分析处理

数据质量低：要求数据管理系统要考虑适应支持更复杂的数据治理、数据分析和机器学习技术

1. **MapReduce/HDFS故障类型及容错方法。**

HDFS:

* NameNode故障: 根据SecondaryNameNode中的FsImage和Editlog数据进行恢复
* DataNode故障: “宕机”，节点上面的所有数据都会被标记为“不可读”

定期检查备份因子

MapReduce:

* + JobTracker故障: 对于MapReduce 1.0的架构，JobTracker故障意味着所有作业需要重新执行

MapReduce 1.0没有处理JobTracker故障的机制，因而成为单点瓶颈

* + TaskTracker故障: JobTracker不会接收到“心跳”

JobTracker会安排其他TaskTracker重新运行失败TaskTracker的任务

* Task故障:

Map Task故障

* + 重新执行Map任务
  + 去HDFS重新读入数据

Reduce Task故障

* + 重新执行Reduce任务

Spark:

* Master故障：ZooKeeper配置多个Master
* Worker故障
* Executor故障
* Driver故障：重启

文本

描述已自动生成

文本

描述已自动生成

文本

描述已自动生成

Yarn:

* Resource Manager故障

如果Resource Manager发生故障，那么它在进行故障恢复时需要从某一持久化存储系统中恢复状态信息，所有应用将会重新执行

我们可以部署多个Resource Manager并通过ZooKeeper进行协调，从而保证Resource Manager的高可用性

* Node Manager故障

Resource Manager认为Node Manager所在节点上所有容器运行的任务也都执行失败，并把执行失败的信息告诉Application Master

* + - AM将向RM重新申请资源运行这些任务
    - RM将分配其它节点的Container执行这些任务

如果发生故障的Node Manager进行恢复，那么它将向Resource Manager重新注册，重置本地的状态信息

* Application Master故障：重启
* Container中的任务故障：重启

1. **宽依赖、窄依赖定义和stage的划分？**



图1. Spark系统的Stage划分示意图

* **窄依赖**表现为一个父RDD的分区对应于一个子RDD的分区或多个父RDD的分区对应于一个子RDD的分区
* **宽依赖**则表现为存在一个父RDD的一个分区对应一个子RDD的多个分区

通过依赖关系进行stage划分：

* 分析各个RDD的偏序关系生成DAG，再通过分析各个RDD中的分区之间的依赖关系来决定如何划分Stage
* 具体划分方法：
  + 在DAG中进行反向解析，遇到宽依赖就断开
  + 遇到窄依赖就把当前的RDD加入到Stage中

为什么将窄依赖尽可能划分在同一个Stage，而将宽依赖划分在不同Stage中？

在Spark中，将窄依赖和宽依赖以不同的方式划分到Stage中，主要是基于它们之间的依赖特性和对计算效率的影响。

1. 窄依赖的特性：

窄依赖表示父RDD（Resilient Distributed Dataset，弹性分布式数据集）的分区与子RDD的分区之间存在一对一或多对一的关系。这种关系使得数据在处理过程中不需要跨节点进行Shuffle（重新分区）。

由于没有Shuffle操作，多个数据处理可以在同一台机器的内存中完成，从而提高了计算效率。

因此，为了充分利用这一优势，窄依赖在Spark中被尽可能划分到同一个Stage中，以便它们能够以流水线（pipeline）的方式迭代执行。

1. 宽依赖的特性：

宽依赖表示父RDD的分区与子RDD的分区之间存在一对多的关系。这种关系通常涉及到Shuffle操作，需要将数据跨节点重新分区。

由于Shuffle的存在，必须等到父RDD的Shuffle处理完成后，才能开始接下来的计算。这意味着宽依赖会阻塞后续的计算过程，直到其完成。

因此，为了避免因宽依赖造成的阻塞，Spark在划分Stage时，遇到宽依赖就会断开当前Stage，形成一个新的Stage。这样，每个Stage都可以独立地执行和完成，从而提高整体的计算效率。

**将窄依赖划分在同一个Stage中，可以充分利用其流水线执行的特性，提高计算效率；而将宽依赖划分在不同Stage中，则可以避免其阻塞后续计算，从而实现更高效的数据处理。**

1. **根据流程图，简述MapReduce/Spark中应用程序的执行流程。**

**HDFS:**

形状

中度可信度描述已自动生成

1. 客户端向NameNode发起文件操作请求
2. NameNode反馈
   * 如果是读写文件操作，则NameNode告知客户端文件块存储的位置信息。
   * 如果是创建、删除、重命名目录或文件等操作，NameNode修改文件目录结构成功后结束。
   * 对于删除操作，HDFS并不会立即去删除DataNode上的数据块，而是等到特定时间才会真正删除。
3. 对于读写文件操作，客户端获知具体位置信息后再与DataNode进行读写交互

**MapReduce:**

图片包含 图形用户界面

描述已自动生成

1. Client将用户编写的MapReduce作业的配置信息、jar包等信息上传到共享文件系统，通常是HDFS。
2. Client提交作业给JobTracker，即告知作业信息的位置。
3. JobTracker读取作业的信息，生成一系列Map和Reduce任务，调度给拥有空闲slot的TaskTracker。
4. TaskTracker根据JobTacker的指令启动Child进程执行Map任务，Map任务将从共享文件系统读取输入数据。
5. JobTracker从TaskTracker处获得Map任务进度信息。
6. 一旦Map任务完成后，JobTacker将Reduce任务分发给TaskTracker。
7. TaskTracker根据JobTacker的指令启动Child进程执行Reduce任务，Reduce任务将从Map任务所在节点的本地磁盘拉取Map的输出结果。
8. JobTracker从TaskTracker处获得Reduce任务进度信息。
9. 当Reduce任务运行结束并将结果写入共享文件系统，则意味着整个作业执行完毕。

**Spark:**

图示

描述已自动生成

1. 启动Driver，以Standalone模式为例
   * 如果使用Client部署方式，客户端直接启动Driver，并向Master注册
   * 如果使用Cluster部署方式，客户端将应用程序提交给Master，由Master选择一个Worker启动Driver进程(DriverWrapper)
2. 构建基本运行环境，即由Driver创建SparkContext，向Cluster Manager进行资源申请，并由Driver进行任务分配和监控
3. Cluster Manager通知工作节点启动Executor进程，该进程内部以多线程方式运行任务
4. Executor进程向Driver注册
5. SparkContext构建DAG并进行任务划分，从而交给Executor进程中的线程来执行任务

**Yarn:**

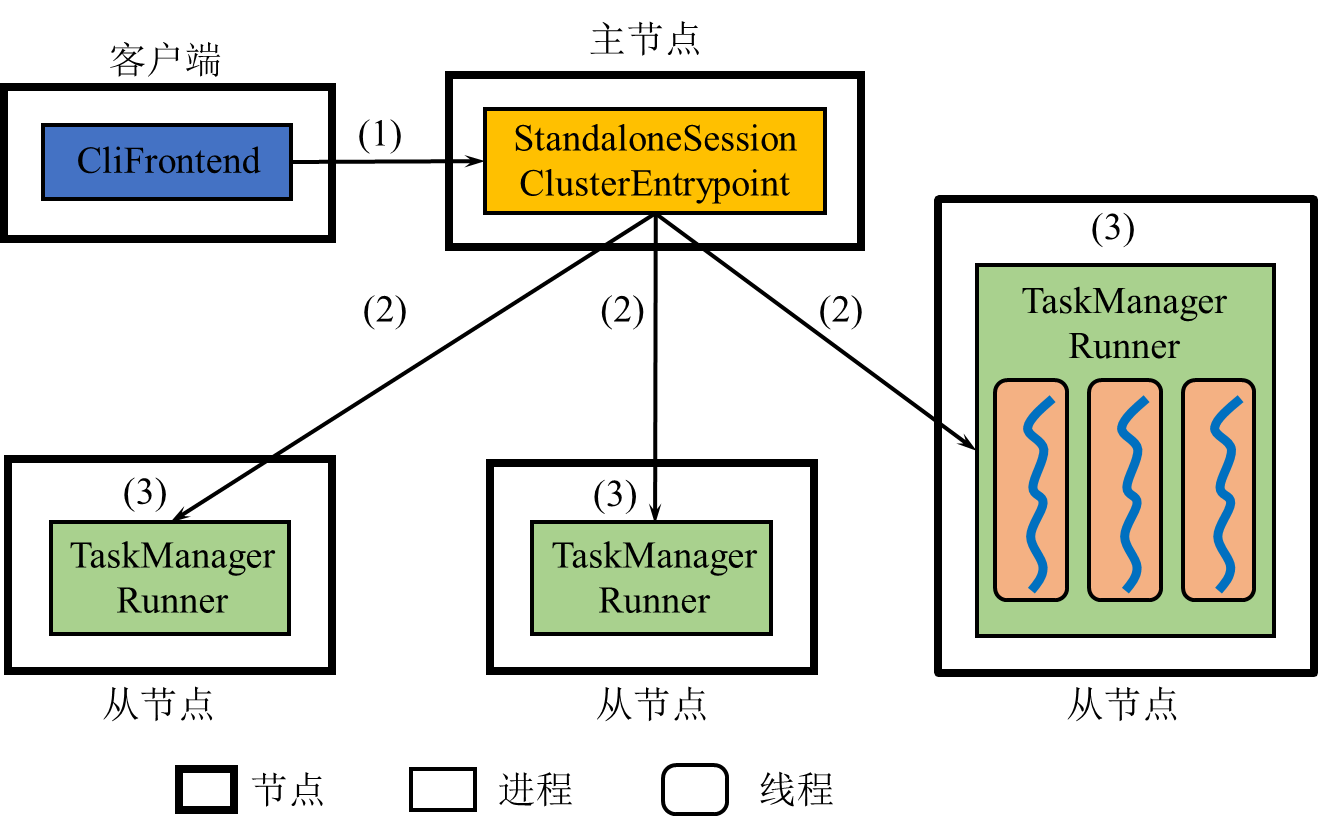
图形用户界面

描述已自动生成

1. 用户编写客户端应用程序，向Yarn提交应用程序。
2. RM负责接收和处理来自客户端的请求，尝试为该程序分配第一个Container，若分配成功则在Container中启动应用程序的AM。
3. AM向RM注册，这样客户端可通过RM查看应用程序的资源使用情况。AM将应用解析为作业并进一步分解为若干任务，并向RM申请启动这些任务的资源。
4. RM向提出申请的AM分配以Container形式表示的资源。一旦AM申请到资源后，在多个任务间进行资源分配。
5. AM确定资源分配方案后，便与对应的NM通信，在相应的Container中启动工作进程用于执行任务。
6. 各个任务向AM汇报自己的状态和进度，以便让AM随时掌握各个任务的运行状态。
7. 随着任务执行结束，AM逐步释放所占用的资源，最终向 RM注销并关闭自己。

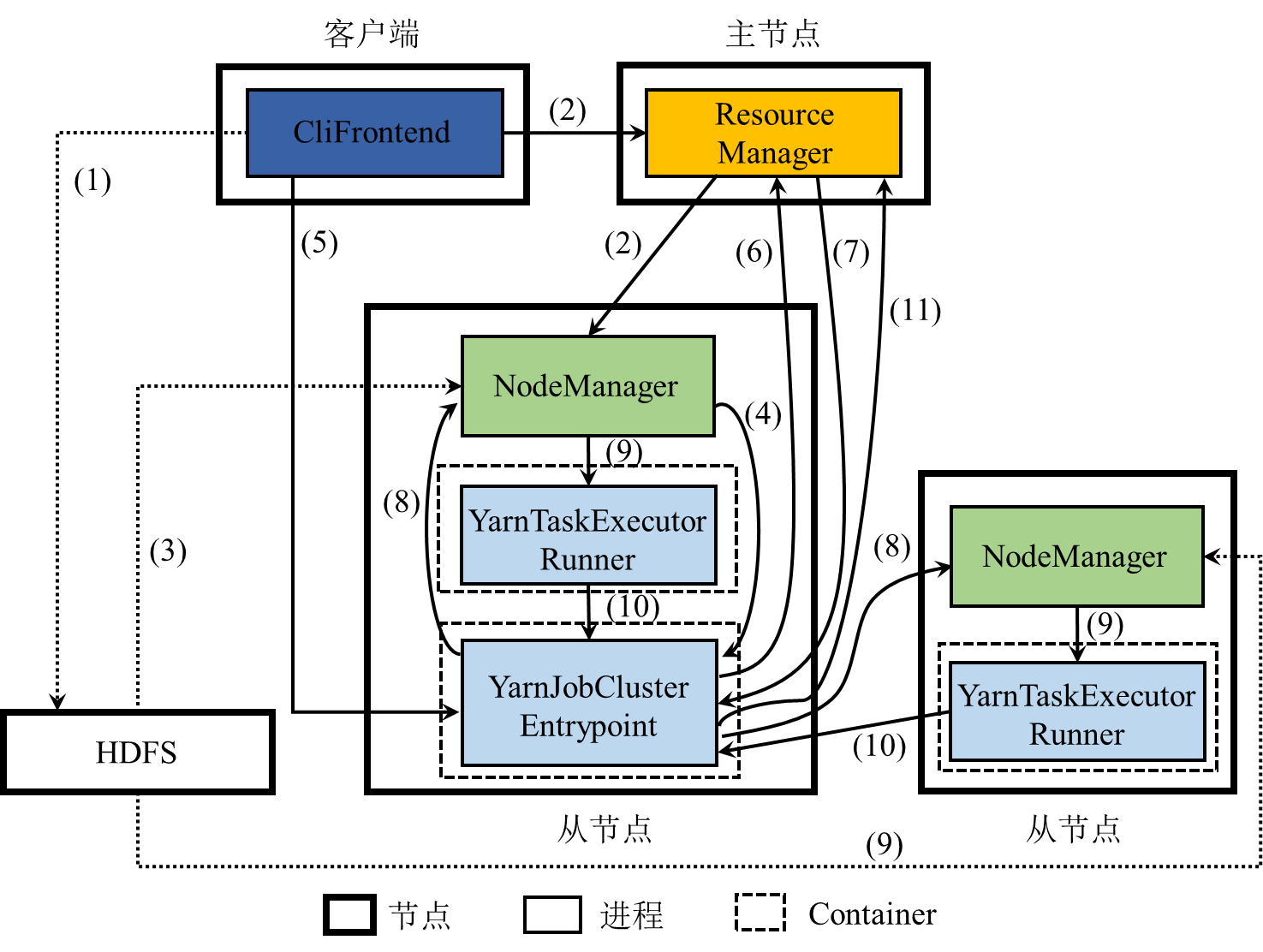
**Flink:**

**Standalone模式：**



1. 客户端将用户编写的程序进行解析，并将解析后的作业描述交给StandaloneSess-ionClusterEntrypoint
2. StandaloneSessionClusterEntrypoint根据作业的描述进行任务分解，确定各个TaskManagerRunner所负责执行的任务
3. TaskManagerRunner执行所负责的任务

**Yarn模式下：**



1. 客户端启动CliFrontend进程，CliFrontend将用户编写的程序进行解析，并将运行Flink系统的jar包以及配置文件上传至HDFS
2. CliFrontend向ResourceManager申请启动YarnJobClusterEntrypoint (ApplicationMaster)，ResourceManager确定启动YarnJobClusterEntrypoint的节点
3. 需启动YarnJobClusterEntrypoint进程的节点上的NodeManager将HDFS中的jar包与配置文件下载到该节点
4. NodeManager启动YarnJobClusterEntrypoint进程
5. CliFrontend进程将解析后的作业描述交给YarnJobClusterEntrypoint
6. YarnJobClusterEntrypoint向ResourceManager注册，这样客户端可以通过 ResourceManager查看Flink应用程序的资源使用情况。 YarnJobClusterEntrypoint根据作业的描述进行任务分解，并向ResourceManager申请启动这些任务的资源
7. ResourceManager以Container形式向提出申请的YarnJobClusterEntrypoint分配资源。得到资源后，它在多个任务间进行资源分配
8. YarnJobClusterEntrypoint确定资源分配方案后，便与对应的NodeManager通信
9. 如果该NodeManager所在节点尚未下载，则将HDFS中的jar包与配置文件下载到本地，并在相应的Container中启动相应的YarnTaskExecutorRunner进程用于执行任务
10. 各个任务向YarnJobClusterEntrypoint汇报自己的状态和进度，以便让 YarnJobClusterEntrypoint随时掌握各个任务的运行状态
11. 随着部分任务执行结束，YarnJobClusterEntrypoint逐步释放所占用的资源，最终向ResourceManager注销并关闭自己
12. **根据提示采用MapReduce/Spark系统根据提示实现文本单词去重程序。**

MapReduc图示, 日程表

中度可信度描述已自动生成图片包含 图形用户界面

描述已自动生成

1. */\*\**
2. \* 描述：单词去重
3. \*
4. \*/
5. public class DistinctWordMR {
7. public static void main(String[] args) throws Exception {
8. *// 指定hdfs相关的参数*
9. Configuration conf = new Configuration();
10. conf.set("fs.defaultFS", "hdfs://hadoop06:9000");
11. System.setProperty("HADOOP\_USER\_NAME", "hadoop");
13. Job job = Job.getInstance(conf);
14. *// 设置jar包所在路径*
15. job.setJarByClass(DistinctWordMR.class);
17. *// 指定mapper类和reducer类*
18. job.setMapperClass(DistinctWordMRMapper.class);
19. job.setReducerClass(DistinctWordMRReducer.class);
21. *// 指定maptask的输出类型*
22. job.setMapOutputKeyClass(Text.class);
23. job.setMapOutputValueClass(NullWritable.class);
24. *// 指定reducetask的输出类型*
25. job.setOutputKeyClass(Text.class);
26. job.setOutputValueClass(NullWritable.class);
28. *// 指定该mapreduce程序数据的输入和输出路径*
29. *//  Path inputPath = new Path("d:/wordcount/input");*
30. *//  Path outputPath = new Path("d:/wordcount/output");*
31. Path inputPath = new Path("/wc/input");
32. Path outputPath = new Path("/wc/output");
33. FileSystem fs = FileSystem.get(conf);
34. if (fs.exists(outputPath)) {
35. fs.delete(outputPath, true);
36. }
37. FileInputFormat.setInputPaths(job, inputPath);
38. FileOutputFormat.setOutputPath(job, outputPath);
40. *// 最后提交任务*
41. boolean waitForCompletion = job.waitForCompletion(true);
42. System.exit(waitForCompletion ? 0 : 1);
43. }
45. */\*\**
46. \* 描述：单词去重MR中的mapper组件。 读取文件然后切分出单词
47. \*/
48. private static class DistinctWordMRMapper extends Mapper<LongWritable, Text, Text, NullWritable> {
50. private Text outkey = new Text();
52. */\*\**
53. \* 在单词计数的场景中。 把单词作为key输出即可， 不用输出value
54. \*/
55. @Override
56. protected void map(LongWritable key, Text value, Context context) throws IOException, InterruptedException {
58. String[] split = value.toString().split(" ");
59. for (String word : split) {
60. outkey.set(word);
61. context.write(outkey, NullWritable.get());
62. }
63. }
64. }
66. */\*\**
67. \* 描述：单词去重的MR程序的reducer组件
68. \*/
69. private static class DistinctWordMRReducer extends Reducer<Text, NullWritable, Text, NullWritable> {
70. @Override
71. protected void reduce(Text key, Iterable<NullWritable> values, Context context) throws IOException, InterruptedException {
73. */\*\**
74. \* reduce方法没调用一次，就接收到一组相同的单词。所以，在此因为是去重的业务，所以直接输出一次key即可。就表示这一组单词就取一个。就相当于实现去重的业务
75. \*/
76. context.write(key, NullWritable.get());
77. }
78. }
79. }

Spark:

1. public class DistinctApp implements SparkConfInfo{
3. public static void main(String[]args){
5. String filePath = "F:\\test\\log.txt";
6. SparkSession sparkSession = new DistinctApp().getSparkConf("distinct");
7. List<String> words = sparkSession.sparkContext()
8. .textFile(filePath, 4)
9. .toJavaRDD()
10. .flatMap(v -> Arrays.asList(v.split("[(\\s+)(\r?\n),.。'’]")).iterator())
11. .filter(v -> v.matches("[a-zA-Z-]+"))
12. .map(String::toLowerCase)
13. .distinct()
14. .collect();
16. words.forEach((v) -> System.out.println(v));
17. sparkSession.stop();
18. }
19. }
21. public interface SparkConfInfo {
23. default SparkSession getSparkConf(String appName){
24. SparkConf sparkConf = new SparkConf();
25. if(System.getProperty("os.name").toLowerCase().contains("win")) {
26. sparkConf.setMaster("local[4]");
27. }else
28. {
29. sparkConf.setMaster("spark://hadoop01:7077,hadoop02:7077,hadoop03:7077");
30. sparkConf.set("spark.driver.host","192.168.150.1");   *//本地ip，必须与spark集群能够相互访问，如：同一个局域网*
31. sparkConf.setJars(new String[] {".\\out\\artifacts\\spark\_test\\spark-test.jar"});   *//项目构建生成的路径*
32. }
34. SparkSession session = SparkSession.builder().appName(appName).config(sparkConf).config(sparkConf).getOrCreate();
35. return session;
36. }
37. }