Hadoop

2.2. 设计思想

2.2.1. 1.分布式文件系统

2.2.2. 2.数据安全性

2.2.3. 3.大数据存储

2.2.4. 4.平台易用性

2.3. 基本结构

2.3.1. NanmeNode

2.3.1.1. 文件元数据

文件名

目录结构

文件属性

生成时间

副本数

权限

2.3.1.2. 文件块列表

2.3.1.3. 文件块位置

2.3.1.4. 文件结构

edits

FSImage

FSTime

2.3.2. DataNode

2.3.2.1. 文件块数据

2.3.2.2. 数据校验和

2.3.2.3. 安全信号

HearBeat

BlockReport

2.3.3. SecondaryNameNode

2.3.3.1. FSImage（元数据镜像文件）

2.3.3.2. FSEdits （元数据操作日志）

2.4. 运行策略

2.4.1. 权限管理

2.4.2. 存储策略

2.4.3. 安全策略

2.4.3.1. 备份策略-文件损坏

文件损坏

2.4.3.2. HearBeat-网络故障

网络故障

2.4.3.3. SecondaryNameNode-机器故障

机器故障

2.4.4. 安全模式

2.5. 深入了解

2.5.1. FileSystem

2.5.1.1. DistribuedFileSystem

2.5.1.2. FilterFileSystem

ChecksumFileSystem

LocalFileSystem

2.5.1.3. RawLocalFileSystem校验和设置

2.5.2. DFSClient

2.5.2.1. Protocal

2.5.2.2. NIO

2.5.3. RPC

2.5.3.1. Socket

3. MapReduce

3.1. 运行策略

3.1.1. shuffle

3.1.1.1. Map

缓冲区同时写出

环行缓冲区

缓冲暂停

sortAndSpill/combineAndSpill

3.1.1.2. Reduce

复制阶段

合并阶段

处理阶段

3.1.1.3. 优化

io.sort.mb

3.1.2. 推测式执行

3.1.2.1. mapred.map.tasks.speculative.execution

3.1.2.2. mapred.reduce.tasks.speculative.executio

3.1.3. JVM重用

3.1.3.1. mapred.job.reuse.jvm.num.tasks

3.1.4. 跳过环记录

3.1.4.1. mapred.map.max.attemps

3.1.4.2. mapred.reduce.max.attemps

3.1.5. 任务执行环境

3.1.5.1. 子主题 **1**

4. Hadoop I/O

4.1. 数据检查

4.1.1. CRC-32循环冗余教研

4.1.2. LocalFileSystem

4.1.2.1. file.crc

4.1.2.2. 512字节/4字节

4.1.2.3. core.xm

校验文件比例io.bytes.per.checksum

所使用的校验系统fs.file.impl

4.1.2.4. 失败

创建bad_files

4.1.3. HDFS

4.1.3.1. DataNode数据检验

Client Upload

DataNode Recieve

4.1.4. 校验和检验

4.1.4.1. JAVA API

一个是在使用open()读取文件前，设置FileSystem中的setVerifyChecksum值为false。

FileSystem fs=new FileSystem();

Fs.setVerifyChecksum(false);

另一个是使用shell命令，比如get命令和copyToLocal命令

4.1.4.2. POSIX

另一个是使用shell命令，比如get命令和copyToLocal命令。

get命令的使用方法如下所示：

hadoop fs-get[-ignoreCrc][-crc]＜src＞＜localdst＞

举个例子：

hadoop fs-get-ignoreCrc input～/Desktop/

get命令会复制文件到本地文件系统。可用-ignorecrc选项复制CRC校验失败的文件，或者使用-crc选项复制文件，以及CRC信息。

copyToLocal的使用方法如下所示：

hadoop fs-copyToLocal[-ignorecrc][-crc]URI＜localdst＞

再举个例子：

hadoop fs-copyToLocal-ignoreCrc input～/Desktop

除了要限定目标路径是一个本地文件外，其他和get命令类似。

禁用校验和检验的最主要目的并不是节约时间，用于检验校验和的开销一般情况都是可以接受的，禁用校验和检验的主要原因是，如果不禁用校验和检验，就无法下载那些已经损坏的文件来查看是否可以挽救，而有时候即使是只能挽救一小部分文件也是很值得的。

4.2. 数据恢复

4.2.1. 1.检查回复标记

```
//如果数据块已经被回复，则直接跳过恢复阶段
synchronized(ongoingRecovery){
Block tmp=new Block();
tmp.set(block.getBlockId(tmp.set(block.getBlockId(), block.getNumBytes(), GenerationStamp.WILDCARD_
STAMP);
if(ongoingRecovery.get(tmp)！=null){
String msg="Block"+block+"is already being recovered, "+"
ignoring this request to recover it.";
LOG.info(msg);
throw new IOException(msg);
}
ongoingRecovery.put(block, block);
}
```

4.2.2. 2.统计数据块回复状况

在这个阶段，DataNode会检查所有出错数据块备份的DataNode，查看这些节点上数据块的恢复信息，然后将所有版本正确的数据块信息、DataNode信息作为一条记录保存在数据块记录表中。

```
//检查每个数据块备份DataNode
for(DataNodeID id:datanodeids){
try{
//获取数据块信息
BlockRecoveryInfo info=datanode.startBlockRecovery(block);
//数据块已不存在
if(info==null){
continue;
}
//数据块版本较晚
if(info.getBlock().getGenerationStamp()<block.getGenerationStamp()){
continue;
}
//正确版本数据块的信息保存起来
blockRecords.add(new BlockRecord(id, datanode, info));
if(info.wasRecoveredOnStartup()){
rwrCount++;//等待回复数
}else{
rbwCount++;//正在恢复数
}
}catch(IOException e){
++errorCount;//出错数
}
```

4.2.3. 3.找出所有版本数据块中正确长度的版本

在这一步骤中，DataNode会逐个扫描上一阶段中保存的数据块记录，首先判断当前副本是否正在恢复，如果正在恢复则跳过，如果不是正在恢复并且配置参数设置了恢复需要保持原副本长度，则将恢复长度相同的副本加入待恢复队列，否则将所有版本正确的副本加入待恢复队列。

```
for(BlockRecord record:blockRecords){
BlockRecoveryInfo info=record.info;
if(!shouldRecoverRwrs&&info.wasRecoveredOnStartup()){
continue;
}
if(keepLength){
if(info.getBlock().getNumBytes()==block.getNumBytes())
{syncList.add(record);}
}else{
syncList.add(record);
```

```
if(info.getBlock().getNumBytes()＜minlength){
minlength=info.getBlock().getNumBytes();
}
}
}
```

4.2.4. 4.副本同步

```
if(！keepLength){
block.setNumBytes(minlength);
}
return syncBlock(block, syncList, targets, closeFile);
```

4.3. 数据压缩

4.3.1. 优势

4.3.1.1. 空间优势

4.3.1.2. 时间优势

4.3.2. 压缩算法

4.3.2.1. Default

org.apache.hadoop.io.compress.DefaultCodec

4.3.2.2. Gzip

***.Gzipcodec**

4.3.2.3. Bzip2

***.Bzip2Codec**

可分割

4.3.2.4. Zlib

***.zlib**

4.3.3. 使用

4.3.3.1. Java API

设置Map处理后压缩数据的代码示例如下:

```
JobConf conf=new Jobconf();
conf.setBoolean("mapred.compress.map.output", true);
```

设置output输出压缩的代码示例如下:

```
JobConf conf=new Jobconf();
conf.setBoolean("mapred.output.compress", true);
conf.setClass("mapred.output.compression.codec", GzipCodec.class, CompressionCodec.class);
```

4.4. 序列化

4.4.1. 目的

4.4.1.1. 进程通信

4.4.1.2. 数据持久化存储

4.4.2. RPC序列化特点

4.4.2.1. 1.紧凑

4.4.2.2. 2.快速

4.4.2.3. 3.可扩展

4.4.2.4. 4.互操作性

4.4.3. Interface -- Writable

4.4.3.1. Method -- void write(DataOutput out)

4.4.3.2. Method -- void readFileds (DataInput in)

4.4.3.3. Implement -- WritableComparator

RawComparator.RawComparator()

4.4.3.4. 数据类型

基本数据类型

BooleanWritable

ByteWritable

IntWritable/VIntWritable

floatWritable

LongWritable/VLongWritable

DoubleWritable

扩展数据类型

NullWritable

BytesWritable/ByteWritable

Text

Text。这可能是这几个自定义类型中相对复杂的一个了。实际上，这是Hadoop中对string类型的重写，但是又与其有一些不同。Text使用标准的UTF-8编码，同时Hadoop使用变长类型VInt来存储字符串，其存储上限是2GB。

Text类型与String类型的主要差别如下：

String的长度定义为String包含的字符个数；Text的长度定义为UTF-8编码的字节数。

ObjectWritable

ObjectWritable是一种多类型的封装。可以适用于Java的基本类型、字符串等。不过，这并不是一个好方法，因为Java在每次被序列化时，都要写入被封装类型的类名。但是如果类型过多，使用静态数组难以表示时，采用这个类仍是不错的做法。

ArrayWritable/TwoDArrayWritable

。ArrayWritable和TwoDArrayWritable，顾名思义，是针对数组和二维数组构建的数据类型。这两个类型声明的变量需要在使用时指定类型，因为ArrayWritable和TwoDArrayWritable并没有空值的构造函数。

ArrayWritable a=new ArrayWritable（IntWritable.class）

同样，在声明它们的子类时，必须使用super（）来指定ArrayWritable和TwoDArrayWritable的数据类型。

```
public class IntArrayWritable extends ArrayWritable{
public IntArrayWritable（）{
super（IntWritable.class）;
}
}
```

一般情况下，ArrayWritable和TwoDArrayWritable都有set（）和get（）函数，在将Text转化为String时，它们也都提供了一个转化函数toArray（）。但是它们没有提供比较器comparator，这点需要注意。同时从TwoDArrayWritable的write和readFields可以看出是横向读写的，同时还会读写每一维的数据长度。

```
public void readFields（DataInput in）throws IOException{
for（int i=0；i＜values.length；i++）{
for（int j=0；j＜values[i].length；j++）{
……
value.readFields（in）;
values[i][j]=value；//保存读取的数据
}
}
}
public void write（DataOutput out）throws IOException{
for（int i=0；i＜values.length；i++）{
out.writeInt（values[i].length）;
}
for（int i=0；i＜values.length；i++）{
for（int j=0；j＜values[i].length；j++）{
values[i][j].write（out）;
}
}
}
```

MapWritable/SortedMapWritable

CompressedWritable

CompressedWritable是保存压缩数据的数据结构。跟之前介绍的数据结构不同，它实现Writable接口，主要面向在Map和Reduce阶段中的大数据对象操作，对这些大数据对象的压缩能够大大加快数据的传输速率。它的主要数据结构是一个byte数组，提供给用户必须实现的函数是readFieldsCompressed和writeCompressed。CompressedWritable在读取数据时先读取二进制字节流，然后调用ensureInflated函数进行解压，在写数据时，将输出的二进制字节流封装成压缩后的二进制字节流。

GenericWritable

这个数据类型是一个通用的数据封装类型。由于是通用的数据封装，它需要保存数据和数据的原始类型，其数据结构如下：

```
private static final byte NOT_SET=-1;
private byte type=NOT_SET;
private Writable instance;
private Configuration conf=null;
```

由于其特殊的数据结构，在读写时也需要读写对应的数据结构：实际数据和数据类型，并且要保证固定的顺序。

```
public void readFields(DataInput in)throws IOException{
//先读取数据类型
type=in.readByte();
……
//再读取数据
instance.readFields(in);
}
public void write(DataOutput out)throws IOException{
if(type==NOT_SET||instance==null)
throw new IOException("The GenericWritable has NOT been set correctly.type="
+type+", instance="+instance);
//先写出数据类型
out.writeByte(type);
//在写出数据
instance.write(out);
}
```

VersionedWritable

VersionedWritable是一个抽象的版本检查类，它主要保证在一个类的发展过程中，使用旧类编写的程序仍然能由新类解析处理。在这个类的实现中只有简单的三个函数：

```
//返回版本信息
public abstract byte getVersion();
//写出版本信息
```

public void write(DataOutput out)throws IOException{

out.writeByte(getVersion());

}

//读入版本信息

public void readFields(DataInput in)throws IOException{

byte version=in.readByte();

if(version！=getVersion())

throw new VersionMismatchException(getVersion(), version);

}

7.3.2□实现自己的Hadoop数据类型

自定义数据类型

```
package cn.edn.ruc.cloudcomputing.book.chapter07;
import java.io.*;
import org.apache.hadoop.io.*;
public class NumPair implements WritableComparable＜NumPair＞{
private LongWritable line;
private LongWritable location;
public NumPair(){
set(new LongWritable(0), new LongWritable(0));
}
public void set(LongWritable first, LongWritable second)
{
this.line=first;
this.location=second;
}
public NumPair(LongWritable first, LongWritable second){
set(first, second);
}
public NumPair(int first, int second){
set(new LongWritable(first), new LongWritable(second));
}
public LongWritable getLine(){
return line;
}
public LongWritable getLocation(){
return location;
}
@Override
```

```
public void readFields(DataInput in)throws IOException
{
line.readFields(in);
location.readFields(in);
}
@Override
public void write(DataOutput out)throws IOException{
line.write(out);
location.write(out);
}
public boolean equals(NumPair o){
if((this.line==o.line)&&(this.location==o.location))
return true;
return false;
}
@Override
public int hashCode(){
return line.hashCode()*13+location.hashCode();
}
@Override
public int compareTo(NumPair o){
if((this.line==o.line)&&(this.location==o.location))
return 0;
return-1;
}
```

4.4.3.5. 文件类

SequenceFile

数据格式

Header是头，它记录的内容如图7-4所示，现在一一对其进行解释：

SequenceFile数据格式（未压缩和Record压缩格式）

version（版本号）：这是一个形如SEQ4或SEQ5的字节数组，一共占四个字节；

keyClassName（key类名）和valueClassName（value类名）：这两个都是String类型，记录的是key和value的数据类型；

compression（压缩）：这是一个布尔类型，它记录的是在这个文件中压缩是否启用；

blockCompression（Block压缩）：布尔类型，记录Block压缩是否启用；

compressor class(压缩类):这是Hadoop内封装的用于压缩key和value的代码;

metadata(元数据):用于记录文件的元数据,文件的元数据是一个<属性名,值>对的列表;

Record:它是数据内容,其内容简单明了,相信大家看图就很容易明白。

Sync-marker:它是一个标记,可以允许程序快速找到文件中随机的一个点。它可以使MapReduce程序更有效率地分割大文件。

需要注意的是,Sync-marker每隔几百个字节会出现一次,因此最后的SequenceFile会是形如图7-5所示的序列文件。

图□7-5□SequenceFile数据存储示例

Sync出现的位置取决于字节数,而不是间隔的Recorder的个数。

从上面的内容可以知道,未压缩与只压缩value的SequenceFile数据格式有两点不同,一是compression(是否压缩)的值不同,二是value存储的数据是否经过了压缩不同。

结构类型

未压缩的key/value对;

记录压缩的Key/value对(这种情况下只有value被压缩);

Block压缩的key/value对(在这种情况下,key与value被分别记录到块中并压缩)

MapFile

数据存储为文件夹中索引

io.map.index.skip索引设置

扩展类

ArrayFile

<Int, Value》

SetFile

<Key>

BloomMapFile

并非直接子类

5. Yarn

5.1. Version 1.0问题

5.1.1. JobTracker单点瓶颈

5.1.2. TaskTracker作业分配信息太简单

5.1.3. 作业延迟过高

在MapReduce运行作业之前，需要TaskTracker汇报自己的资源情况和运行情况，JobTracker根据获取的信息分配作业，TaskTracker获取任务之后再开始运行。这样的结果是通信的延迟造成作业启动时间过长。最显著的影响是小作业并不能及时完成。

5.1.4. 编程框架不够灵活

虽然现在的MapReduce框架允许用户自己定义各个阶段的处理函数和对象，但是MapReduce框架还是限制了编程的模式及资源的分配

5.2. 设计要求

5.2.1. 可靠性（Reliability）

5.2.2. 可用性（Availability）

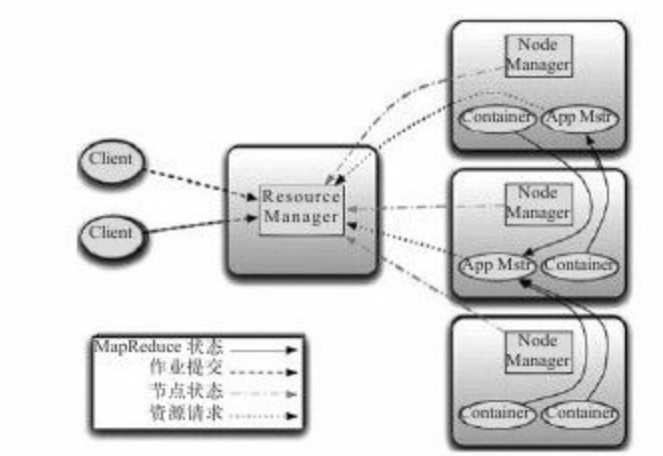
5.2.3. 扩展性（Scalability）

5.2.4. 向后兼容（Backward Compatibility）

5.2.5. 可预测延迟（Predictable Latency）

5.2.6. 其他框架兼容性

5.3. 架构



5.3.1. 资源管理器ResourceManager

5.3.1.1. 应用管理器ApplicationManager

负责接收作业，协商获取第一个资源容器用于执行应用的任务主题并为重启失败的应用主题分配容器。应用管理器负责启动系统中所有应用的应用主体并管理其生命周期。在启动应用主体之后，应用管理器通过应用主体定期发送的“心跳”来监督应用主体，保证其可用性，如果应用主体失败，就需要将其重启

调度协商组件

应用主体容器管理组件

应用主体监控组件

5.3.1.2. 调度器 Scheduler

负责资源的分配。

调度器根据各个应用的资源需求和集群各个节点的资源容器（Resource Container，是集群节点将自身内存、CPU、磁盘等资源封装在一起的抽象概念）进行调度

5.3.2. 节点管理器NodeManager

节点管理器是每个结点的框架代理。它负责启动应用的容器，监控容器的资源使用（包括CPU、内存、硬盘和网络带宽等），并把这些用信息汇报给调度器。应用对应的应用主体负责通过协商从调度器处获取资源容器，并跟踪这些容器的状态和应用执行的情况。

5.3.2.1. 为应用启用调度器已分配给应用的容器

5.3.2.2. 保证已启用的容器不会使用超过分配的资源量

5.3.2.3. 为**task**构建容器环境，包括二进制可执行文件，**jars**等

5.3.2.4. 为所在的节点提供一个管理本地存储资源的简单服务

5.3.3. 应用主体**ApplicationMaster**

5.3.3.1. 作用

与调度器协商资源

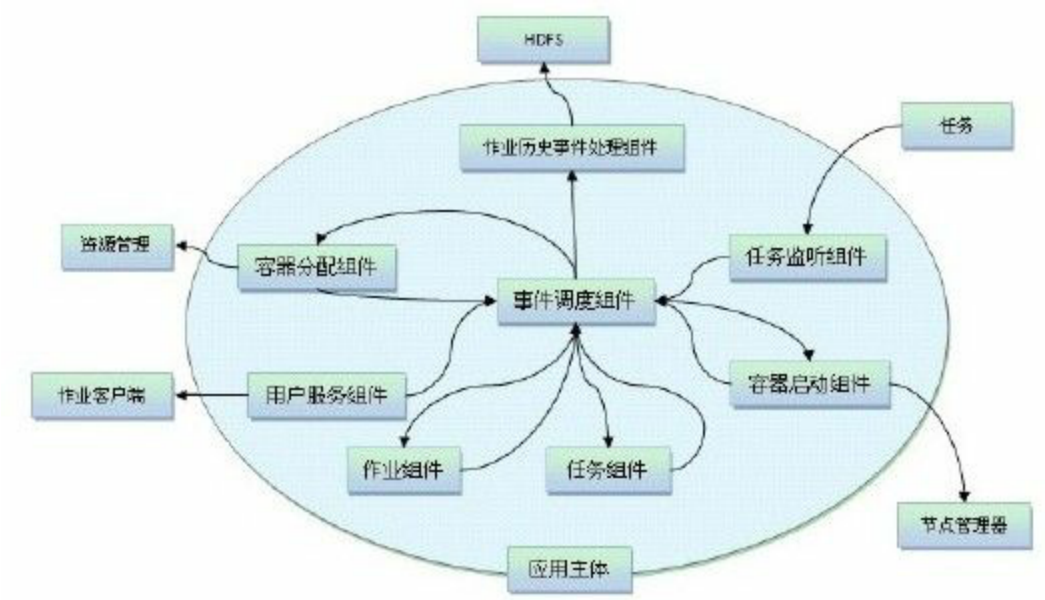
与节点管理器合作，在合适的容器中运行对应的组件**task**，并监控这些**task**执行

如果**container**出现故障，应用主体会重新向调度器申请其他资源

计算应用程序所需的资源量，并转化成调度器可识别的协议信息包

在应用主体出现故障后，应用管理器会负责重启它，但由应用主体自己从之前保存的应用程序执行状态中恢复应用程序

5.3.3.2. 应用主体组件事件流



事件调度组件

容器分配组件

用户服务组件

任务监听组件

任务组件

容器启动组件

作业历史事件处理组件

作业组件

5.3.4. 容器Container

在MapReduce

V2中，系统资源的组织形式是将节点上的可用资源分割，每一份通过封装组织成系统的一个资源单元，即Container（比如固定大小的内存分片、CPU核心数、网络带宽量和硬盘空间块等。在现在提出的MapReduce V2中，所谓资源是指内存资源，每个节点由多个512MB或1GB大小的内存容器组成）。而不是像MapReduce V1中那样，将资源组织成Map池和Reduce池。应用主体可以申请任意多个该内存整数倍大小的容器。由于

将每个节点上的内存资源分割成了大小固定、地位相同的容器，这些内存容器就可以在任务执行中进行互换，从而提高利用率，避免了在MapReduce
V1中作业在Reduce池上的瓶颈问题和缺乏资源互换的问题。资源容器的主要职责就是运行、保存或传输应用主体提交的作业或需要存储和传输的数据。

5.4. 设计细节

5.4.1. 资源协商

5.4.2. 调度

调度器接收到应用主体的请求之后会根据自己的全局计划及各种限制返回对请求的回复。回复中主要包括三类信息：最新分配的资源容器列表、在应用主体和资源管理器上次交互之后完成任务的应用指定资源容器的状态、当前集群中应用程序可用的资源数量。应用主体可以收集完成容器的信息并对失败任务做出反应。可用资源量可以为应用主体接下来的资源申请提供参考，比如应用主体可以使用这些信息来合理分配Map和Reduce各自请求的资源数量，进而防止死锁（最明显的情况是Reduce请求占用所有的剩余可用资源）

5.4.2.1. 选择系统中“最低服务”的队列。这个队列可以是等待时间最长的队列，或者等待时间与已分配资源之比最大的队列等

5.4.2.2. 从队列中选择拥有最高优先级的作业

5.4.2.3. 满足被选出的作业的资源请求

5.4.3. 资源监控

调度器定期从节点管理器处收集已分配资源的使用信息。同时，调度器还会将已完成任务容器的状态设置为可用，以便有需求的应用申请使用

5.4.4. 应用提交

5.4.4.1. 用户提交作业到应用管理器

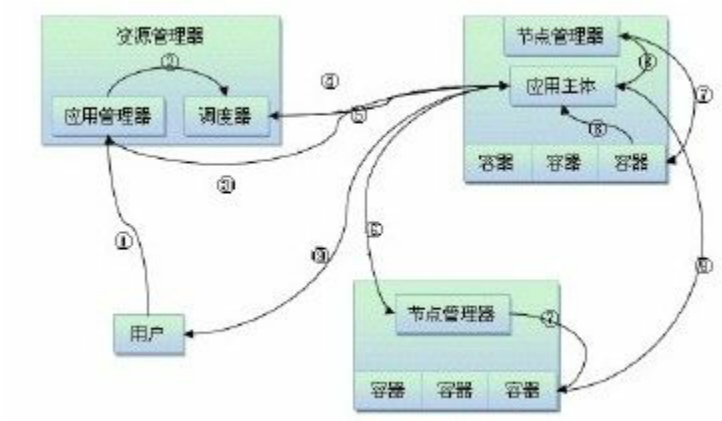
具体的步骤是在用户提交作业之后，MapReduce框架为用户分配一个新的应用ID，并将应用的定义打包上传到HDFS上用户的应用缓存目录中。最后提交此应用给应用管理器

5.4.4.2. 应用管理器接受应用提交

5.4.4.3. 应用管理器同调度器协商获取运行应用主体所需的第一个资源容器，并执行应用主体

5.4.4.4. 应用管理器将启动的应用主体细节信息发还给用户，以便其监督应用的进度

5.5. Yarn作业执行流程



5.5.1. MapReduce框架接收用户提交的作业，并为其分配一个新的应用**ID**，并将应用的定义打包上传到**HDFS**上用户的应用缓存目录中，然后提交此应用给应用管理器。

5.5.2. 应用管理器同调度器协商获取运行应用主体所需的第一个资源容器

5.5.3. 应用管理器在获取的资源容器上执行应用主体

5.5.4. 应用主体计算应用所需资源，并发送资源请求到调度器

5.5.5. 调度器根据自身统计的可用资源状态和应用主体的资源请求，分配合适的资源容器给应用主体

5.5.6. 应用主体与所分配容器的节点管理器通信，提交作业情况和资源使用说明

5.5.7. 节点管理器启用容器并运行任务

5.5.8. 应用主体监控容器上任务的执行情况

5.5.9. 应用主体反馈作业的执行状态信息和完成状态

5.6. 优势

5.6.1. 分散了JobTracker的任务

资源管理任务由资源管理器负责，作业启动、运行和监测任务由分布在集群节点上的应用主体负责。这样大大减缓了MapReduce V1中JobTracker单点瓶颈和单点风险的问题，大大提高了集群的扩展性和可用性。

5.6.2. ApplicationMaster提升扩展性

5.6.3. 在资源管理器上使用ZooKeeper实现故障转移

5.6.4. 集群资源统一组织成资源容器，提高资源利用率