课后作业—倒排索引

2018年8月2日 10:00

a.txt

hello hadoop

hello world

hello hadoop

b.txt

hello spark

hello hive

hello hadoop

c.txt

hello hadoop

1803 hadoop

hadoop hive

最后形成的倒排索引表：

1803 c.txt 1

hadoop c.txt 2 b.txt 1 a.txt 2

hello c.txt 1 b.txt 3 a.txt 3

hive c.txt 1 b.txt 1

spark b.txt 1

world a.txt 1

InvertMapper1代码：

```
public class InvertMapper1 extends Mapper<LongWritable,Text,Text,IntWritable>{


    @Override
    protected void map(LongWritable key, Text value, Mapper<LongWritable, Text, Text,
    IntWritable>.Context context)
                throws IOException, InterruptedException {
        String line=value.toString();
        FileSplit split=(FileSplit) context.getInputSplit();
        String fileName=split.getPath().getName();
        String[] words=line.split(" ");
```

```
            for(String word:words){
                context.write(new Text(word+"|"+fileName), new IntWritable(1));
            }

        }
    }
```

InvertReducer1代码：

```
public class InvertReducer1 extends Reducer<Text,IntWritable,Text,IntWritable>{

    @Override
    protected void reduce(Text key, Iterable<IntWritable> values,
            Reducer<Text, IntWritable, Text, IntWritable>.Context context) throws
            IOException, InterruptedException {
        int count=0;
        for(IntWritable value:values){
            count=count+value.get();
        }
        context.write(key,new IntWritable(count));
    }
}
```

InvertMapper2代码：

```
public class InvertMapper2 extends Mapper<LongWritable,Text,Text,Text>{

    @Override
    protected void map(LongWritable key, Text value, Mapper<LongWritable, Text, Text,
    Text>.Context context)
            throws IOException, InterruptedException {
        String line=value.toString();
        String word=line.split("\\|")[0];
        String FileInfo=line.split("\\|")[1];

        context.write(new Text(word), new Text(FileInfo));
    }
}
```

InvertReducer2代码：

```
public class InvertReducer2 extends Reducer<Text,Text,Text,Text>{

	@Override
	protected void reduce(Text key, Iterable<Text> values,
			Reducer<Text, Text, Text, Text>.Context context)
			throws IOException, InterruptedException {
		String result="";
		for(Text value:values){
			result=result+" "+value.toString();
		}
		context.write(key, new Text(result));
	}
}
```

InverDriver代码：

```
public class InvertDriver {

	public static void main(String[] args) throws Exception {
		Configuration conf=new Configuration();
		Job job=Job.getInstance(conf);

		job.setJarByClass(InvertDriver.class);

		job.setMapperClass(InvertMapper1.class);

		job.setReducerClass(InvertReducer1.class);


		job.setMapOutputKeyClass(Text.class);

		job.setMapOutputValueClass(IntWritable.class);


		job.setOutputKeyClass(Text.class);

		job.setOutputValueClass(IntWritable.class);
```

```
            FileInputFormat.addInputPath(job, new Path("hdfs://192.168.150.137:9000/invert"));

            FileOutputFormat.setOutputPath(job,new
            Path("hdfs://192.168.150.137:9000/invert/result"));


            if(job.waitForCompletion(true)){
                Job job2=Job.getInstance(conf);
                job2.setMapperClass(InvertMapper2.class);
                job2.setReducerClass(InvertReducer2.class);

                job2.setMapOutputKeyClass(Text.class);
                job2.setMapOutputValueClass(Text.class);

                job2.setOutputKeyClass(Text.class);
                job2.setOutputValueClass(Text.class);

                FileInputFormat.addInputPath(job2, new
                Path("hdfs://192.168.150.137:9000/invert/result"));

                FileOutputFormat.setOutputPath(job2,new
                Path("hdfs://192.168.150.137:9000/invert/result2"));

                job2.waitForCompletion(true);
            }
        }
}
```

课后作业—Join

2018年7月16日 20:09

订单表数据

1001 20170710 4 2
1002 20170710 3 100
1003 20170710 2 40
1004 20170711 2 23
1005 20170823 4 55
1006 20170824 3 20
1007 20170825 2 3
1008 20170826 4 23
1009 20170912 2 10
1010 20170913 2 2
1011 20170914 3 14
1012 20170915 2 18

商品表数据

1 chuizi 3999
2 Huawei 3999
3 Xiaomi 2999
4 Apple 5999

最后输出的结果：

20170710 Item [订单id=1003, 订单日期=20170710, 物品id=2, 出货量=40, 品牌=Huawei, 商品单价=3999.0]

20170710 Item [订单id=1002, 订单日期=20170710, 物品id=3, 出货量=100, 品牌=Xiaomi, 商品单价=2999.0]

20170710 Item [订单id=1001, 订单日期=20170710, 物品id=4, 出货量=2, 品牌=Apple, 商品单价=5999.0]

20170711 Item [订单id=1004, 订单日期=20170711, 物品id=2, 出货量=23, 品牌=Huawei, 商品单价=3999.0]

20170823 Item [订单id=1005, 订单日期=20170823, 物品id=4, 出货量=55, 品牌=Apple, 商品单价=5999.0]

20170824 Item [订单id=1006, 订单日期=20170824, 物品id=3, 出货量=20, 品牌=Xiaomi, 商品单价=2999.0]

20170825 Item [订单id=1007, 订单日期=20170825, 物品id=2, 出货量=3, 品牌=Huawei, 商品单价=3999.0]

20170826 Item [订单id=1008, 订单日期=20170826, 物品id=4, 出货量=23, 品牌=Apple, 商品单价=5999.0]

mapper代码：

```
public class JoinMapper extends Mapper<LongWritable,Text,Text,Item>{

    @Override
    protected void map(LongWritable key, Text value, Mapper<LongWritable, Text, Text,
    Item>.Context context)
            throws IOException, InterruptedException {
        String line=value.toString();
        String[] infos=line.split(" ");
        Item item=new Item();
        FileSplit split=(FileSplit) context.getInputSplit();

        if(split.getPath().getName().startsWith("order")){
            item.setId(infos[0]);
            item.setDate(infos[1]);
            item.setPid(infos[2]);
            item.setAmount(Integer.parseInt(infos[3]));
            item.setName("");
            item.setPrice(0.0);

        }else{
            item.setPid(infos[0]);
            item.setName(infos[1]);
            item.setPrice(Double.parseDouble(infos[2]));
            item.setId("");
            item.setDate("");
```

```
                item.setAmount(0);

            }

            context.write(new Text(item.getPid()),item);

        }
    }
```

Item 代码：

```
public class Item implements Writable,Cloneable{

    private String id;
    private String date;
    private String pid;
    private int amount;
    private String name;
    private double price;

    @Override
    public void write(DataOutput out) throws IOException {
        out.writeUTF(id);
        out.writeUTF(date);
        out.writeUTF(pid);
        out.writeInt(amount);
        out.writeUTF(name);
        out.writeDouble(price);


    }

    @Override
    public void readFields(DataInput in) throws IOException {
        this.id=in.readUTF();
        this.date=in.readUTF();
        this.pid=in.readUTF();
        this.amount=in.readInt();
        this.name=in.readUTF();
        this.price=in.readDouble();
```

```
}

public Item clone(){
	Item o=null;
	try {
		o=(Item)super.clone();
	} catch (Exception e) {
		e.printStackTrace();
	}

	return o;
}

public String getId() {
	return id;
}

public void setId(String id) {
	this.id = id;
}

public String getDate() {
	return date;
}

public void setDate(String date) {
	this.date = date;
}

public String getPid() {
	return pid;
}

public void setPid(String pid) {
	this.pid = pid;
}

public int getAmount() {
	return amount;
```

```
    }

    public void setAmount(int amount) {
        this.amount = amount;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public double getPrice() {
        return price;
    }

    public void setPrice(double price) {
        this.price = price;
    }

    @Override
    public String toString() {
        return "Item [订单id=" + id + ", 订单日期=" + date + ", 物品id=" + pid + ", 出货量=" +
        amount + ", 品牌=" + name+ ", 商品单价=" + price + "]";
    }


}
```

reducer代码：

```
public class JoinReducer extends Reducer<Text, Item,Item,NullWritable>{
    Map<String, Item> productMap=new HashMap<>();


    @Override
    protected void reduce(Text key, Iterable<Item> values, Reducer<Text, Item, Item,
    NullWritable>.Context context)
```

```
            throws IOException, InterruptedException {


        List<Item> list=new ArrayList<>();

        for(Item value:values){
            Item item=value.clone();
            list.add(item);
            if(value.getId().equals("")){
                productMap.put(item.getPid(),item);
            }
        }

        for(Item value:list){
            if(!value.getId().equals("")){
            Item productItem=new Item();
            productItem.setId(value.getId());
            productItem.setDate(value.getDate());
            productItem.setAmount(value.getAmount());
            productItem.setPid(value.getPid());

            productItem.setName(productMap.get(value.getPid()).getName());
            productItem.setPrice(productMap.get(value.getPid()).getPrice());

            context.write(productItem, NullWritable.get());
            }
        }

    }
}
```

Driver代码：

```
public class Driver {

    public static void main(String[] args) throws Exception {
        Configuration conf=new Configuration();
        Job job=Job.getInstance(conf);

        job.setJarByClass(Driver.class);
```

```
        job.setMapperClass(JoinMapper.class);
        job.setReducerClass(JoinReducer.class);

        job.setMapOutputKeyClass(Text.class);
        job.setMapOutputValueClass(Item.class);

        job.setOutputKeyClass(Item.class);
        job.setOutputValueClass(NullWritable.class);


        FileInputFormat.setInputPaths(job, new Path("hdfs://192.168.150.137:9000/join"));
        FileOutputFormat.setOutputPath(job, new
        Path("hdfs://192.168.150.137:9000/join/result"));

        job.waitForCompletion(true);
    }
}
```

Map Side Join

2018年7月16日 20:43

概述

在利用MapReduce做Join操作时，经常会出现数据倾斜的情况，产生的原因主要来自于业务，比如Mapper输出的是热销商品的pid，这样会造成某个join操作的reduce收到的数据特别多。

如何解决数据倾斜是一个常谈的话题，不同的框架有不同的处理方案，如果是MR框架的话，我们可以利用DistributedCache（Hadoop内置的分布式缓存机制）来实现。

DistributedCache 是一个提供给Map/Reduce框架的工具，用来缓存指定的文件。当我们使用了这个机制后，MR框架底层会将指定的文件拷贝到slave节点上的缓存中。

使用DistributedCache机制，尤其在做join操作时，可以大大的提高作业的运行效率，并且可以额避免产生数据倾斜。实现思路是：

将Join操作中的**小表**进行缓存，这样每个Map Task在执行时，都是可以在Map Task运行所在的节点的缓冲区拿到小表数据，从而在Map阶段就可以完成Join操作。这样一来，就不需要引入Reducer组件，也就不会产生数据倾斜的问题。

订单表数据

1001 20170710 4 2
1002 20170710 3 100
1003 20170710 2 40
1004 20170711 2 23
1005 20170823 4 55
1006 20170824 3 20
1007 20170825 2 3
1008 20170826 4 23

1009 20170912 2 10

1010 20170913 2 2

1011 20170914 3 14

1012 20170915 2 18

商品表数据

1 chuizi 3999

2 Huawei 3999

3 Xiaomi 2999

4 Apple 5999

入门案例

sideMapper代码：

```
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.net.URI;

import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.FSDataInputStream;
import org.apache.hadoop.fs.FileSystem;
import org.apache.hadoop.fs.Path;
```

sideMapper代码：

```
public class SideMapper extends Mapper<LongWritable, Text,Text,Item>{

    private Map<String,Item> productMap;

    @Override
    protected void setup(Mapper<LongWritable, Text, Text,Item>.Context context)
            throws IOException, InterruptedException {

        productMap=new HashMap<String,Item>();

        Configuration conf = context.getConfiguration();
        URI[] localCacheFiles = context.getCacheFiles();
```

```
        FileSystem fs = FileSystem.get(localCacheFiles[0], conf);
        FSDataInputStream in = fs.open(new Path(localCacheFiles[0]));
        BufferedReader br=new BufferedReader(new InputStreamReader(in));

        String line=null;

        while((line=br.readLine())!=null){
                String[] itemInfo=line.split(" ");
                Item item=new Item();
                item.setPid(itemInfo[0]);
                item.setName(itemInfo[1]);
                item.setPrice(Double.parseDouble(itemInfo[2]));

                productMap.put(item.getPid(),item);

    }
        br.close();

}


@Override
protected void map(LongWritable key, Text value, Mapper<LongWritable, Text,Text,Item>.Context
context)
                throws IOException, InterruptedException {

        String line=value.toString();
        String[] orderInfo=line.split(" ");
        Item item=new Item();
        item.setId(orderInfo[0]);
        item.setDate(orderInfo[1]);
        item.setPid(orderInfo[2]);
        item.setAmount(Integer.parseInt(orderInfo[3]));

        item.setName(productMap.get(item.getPid()).getName());
        item.setPrice(productMap.get(item.getPid()).getPrice());
```

```
            context.write(new Text(item.getDate()), item);


        }
    }
```

Item代码：

```
public class Item implements Writable,Cloneable{


    private String id="";
    private String date="";
    private String pid="";
    private int amount;
    private String name="";
    private double price;

    @Override
    public void write(DataOutput out) throws IOException {
        out.writeUTF(id);
        out.writeUTF(date);
        out.writeUTF(pid);
        out.writeInt(amount);
        out.writeUTF(name);
        out.writeDouble(price);


    }

    @Override
    public void readFields(DataInput in) throws IOException {
        this.id=in.readUTF();
        this.date=in.readUTF();
        this.pid=in.readUTF();
        this.amount=in.readInt();
        this.name=in.readUTF();
        this.price=in.readDouble();

    }

    public Item clone(){
        Item o=null;
        try {
            o=(Item)super.clone();
        } catch (Exception e) {
            e.printStackTrace();
        }

        return o;
    }

    public String getId() {
        return id;
    }
```

```
	public void setId(String id) {
		this.id = id;
	}

	public String getDate() {
		return date;
	}

	public void setDate(String date) {
		this.date = date;
	}

	public String getPid() {
		return pid;
	}

	public void setPid(String pid) {
		this.pid = pid;
	}

	public int getAmount() {
		return amount;
	}

	public void setAmount(int amount) {
		this.amount = amount;
	}

	public String getName() {
		return name;
	}

	public void setName(String name) {
		this.name = name;
	}

	public double getPrice() {
		return price;
	}

	public void setPrice(double price) {
		this.price = price;
	}

	@Override
	public String toString() {
		return "Item [订单id=" + id + ", 订单日期=" + date + ", pid=" + pid + ", 出货量=" + amount + ",
		品牌=" + name
				+ ", 商品单价=" + price + "]";
	}


}
```

SideDriver代码：

```
public class SideDriver {

	public static void main(String[] args) throws Exception {
		Configuration conf=new Configuration();
		Job job=Job.getInstance(conf);

		job.setJarByClass(SideDriver.class);
		job.setMapperClass(SideMapper.class);

		job.setMapOutputKeyClass(Text.class);
		job.setMapOutputValueClass(Item.class);
		job.addCacheFile(new
		Path("hdfs://192.168.150.137:9000/cachejoin/product_phone.txt").toUri());


		FileInputFormat.addInputPath(job, new Path("hdfs://192.168.150.137:9000/join"));
		FileOutputFormat.setOutputPath(job,new Path("hdfs://192.168.150.137:9000/join/result"));

		job.waitForCompletion(true);
	}
}
```

Hadoop压缩机制

Hadooop默认支持的两种压缩格式比较

压缩格式	是否支持split	压缩率	速度	是否hadoop自带	linux命令	换成压缩格式后，原来的应用程序是否要修改	Linux指令操作
gzip	否	很高	比较快	是，直接使用	有	和文本处理一样，不需要修改	压缩：gzip 1.txt 解压缩：gzip -d 1.txt.gz
bzip2	是	最高	慢	是，直接使用	有	和文本处理一样，不需要修改	压缩：bzip2 1.txt 解压缩：bzip2 -d 1.txt.bz2
snappy							
lazo							

1 gzip压缩

优点：压缩率比较高，而且压缩/解压速度也比较快；hadoop本身支持，在应用中处理gzip格式的文件就和直接处理文本一样；有hadoop native库；大部分linux系统都自带gzip命令，使用方便。

缺点：不支持split。

2 bzip2压缩

优点：支持split；具有很高的压缩率，比gzip压缩率都高；hadoop本身支持，但不支持native；在linux系统下自带bzip2命令，使用方便。

缺点：压缩/解压速度慢；不支持native。

应用场景：适合对速度要求不高，但需要较高的压缩率的时候，可以作为mapreduce作业的输出格式；或者输出之后的数据比较大，处理之后的数据需要压缩存档减少磁盘空间并且以后数据用得比较少的情况；或者对单个很大的文本文件想压缩减少存储空间，同时又需要支持split，而且兼容之前的应用程序（即应用程序不需要修改）的情况。

应用场景：当每个文件压缩之后在128M以内的（1个块大小内），都可以考虑用gzip压缩格式。譬如说一天或者一个小时的日志压缩成一个gzip文件，运行mapreduce程序的时候通过多个gzip文件达到并发。hive程序，streaming程序，和java写的mapreduce程序完全和文本处理一样，压缩之后原来的程序不需要做任何修改。

Hadoop 压缩API实现

将最后生成的结果文件压缩成gzip格式

CompressDriver代码：

```
public class CompressDriver {

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

        Configuration conf=new Configuration();

        Job job=Job.getInstance(conf);


        job.setJarByClass(CompressDriver.class);

        job.setMapperClass(CompressMapper.class);


        job.setMapOutputKeyClass(Text.class);

        job.setMapOutputValueClass(IntWritable.class);


        //--将最后生成的结果文件压缩

        FileOutputFormat.setCompressOutput(job,true);

        FileOutputFormat.setOutputCompressorClass(job, GzipCodec.class);


        FileInputFormat.setInputPaths(job, new

        Path("hdfs://192.168.234.21:9000/compress"));

        FileOutputFormat.setOutputPath(job, new

        Path("hdfs://192.168.234.21:9000/compress/result"));


        job.waitForCompletion(true);
```

```
    }
}
```

对map任务的输出结果进行压缩

如果对map任务的中间输出结果进行压缩，也会获得不少好处。因为map的结果最后要通过网络通信传递到reduce节点，如果压缩，会减少数据传输量，节省集群带宽资源。

map、reduce代码略。Driver代码：

```
public class CompressDriver {


    public static void main(String[] args) throws Exception {
        Configuration conf=new Configuration();
        Job job=Job.getInstance(conf);
        conf.setBoolean("mapred.compress.map.output", true);
        conf.setClass("mapred.map.output.compression.codec",GzipCodec.cla
        ss,CompressionCodec.class);

        job.setJarByClass(CompressDriver.class);
        job.setMapperClass(CompressMapper.class);
        job.setReducerClass(CompressReducer.class);

        job.setMapOutputKeyClass(Text.class);
        job.setMapOutputValueClass(IntWritable.class);
        job.setOutputKeyClass(Text.class);
        job.setOutputValueClass(IntWritable.class);
```

```
        FileInputFormat.setInputPaths(job, new
        Path("hdfs://192.168.234.21:9000/compress"));
        FileOutputFormat.setOutputPath(job, new
        Path("hdfs://192.168.234.21:9000/compress/result"));

        job.waitForCompletion(true);
    }
}
```

Yarn概述

概述

Apache Hadoop YARN （Yet Another Resource Negotiator，另一种资源协调者）是一种新的 Hadoop 资源管理器，它是一个通用**资源管理**系统，可为上层应用提供统一的资源管理和调度，它的引入为集群在**资源利用率、资源统一管理**和数据共享等方面带来了巨大好处。

YARN的基本思想是将JobTracker的两个主要功能（**资源管理**和**作业调度/监控**）分离，主要方法是创建一个全局的ResourceManager（RM）和若干个针对应用程序的ApplicationMaster（AM）。这里的应用程序是指传统的MapReduce作业。

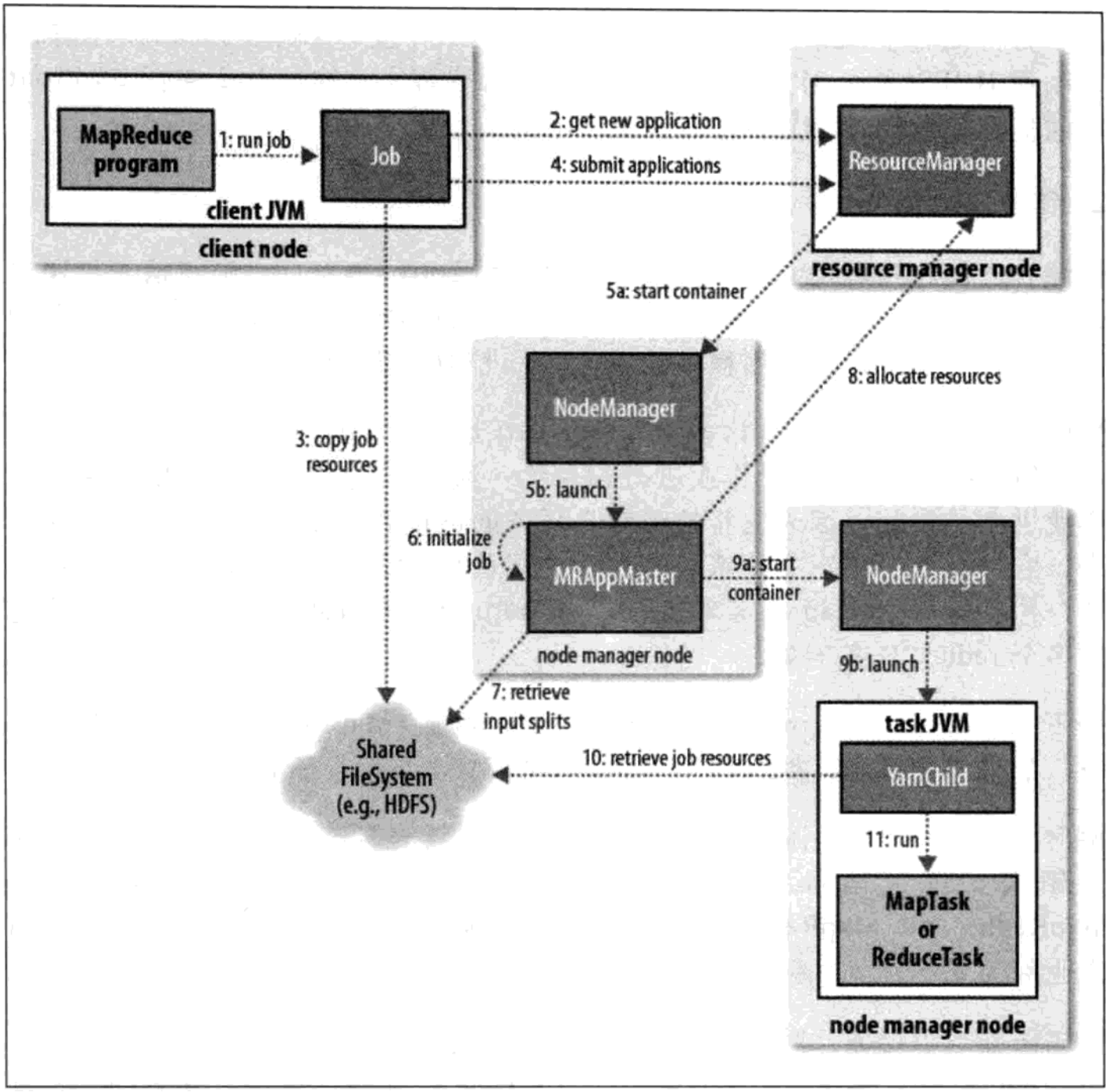
YARN 分层结构的本质是 ResourceManager。这个实体控制整个集群并管理应用程序向基础计算资源的分配。ResourceManager 将各个资源部分（计算、内存、带宽等）精心安排给基础 NodeManager（YARN 的每节点代理）。

ResourceManager 还与 ApplicationMaster 一起分配资源，与 NodeManager 一起启动和监视它们的基础应用程序。在此上下文中，ApplicationMaster 承担了以前的 TaskTracker 的一些角色，ResourceManager 承担了 JobTracker 的角色。

ApplicationMaster 管理一个在 YARN 内运行的应用程序的每个实例。ApplicationMaster 负责协调来自 ResourceManager 的资源，并通过 NodeManager 监视容器的执行和资源使用（CPU、内存等的资源

分配）。

Yarn体系架构图



YARN的核心思想

将JobTracker和TaskTacker进行分离，它由下面几大构成组件：

a. 一个全局的资源管理器 ResourceManager

b.ResourceManager的每个节点代理 NodeManager

c. 表示每个应用的 ApplicationMaster

d. 每一个ApplicationMaster拥有多个Container在NodeManager上运行

YARN的主要架构

ResourceManager（RM）

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。**它主要由两个组件构成：**调度器（**Scheduler**）**和**应用程序管理器**（Applications Manager，ASM）。

调度器 调度器根据容量、队列等限制条件（如每个队列分配一定的资源，最多执行一定数量的作业等），将系统中的资源分配给各个正在运行的应用程序。需要注意的是，该调度器是一个“纯调度器”，它不再从事任何与具体应用程序相关的工作，比如不负责监控或者跟踪应用的执行状态等，也不负责重新启动因应用执行失败或者硬件故障而产生的失败任务，这些均交由应用程序相关的ApplicationMaster完成。**调度器仅根据各个应用程序的资源需求进行资源分配，而资源分配单位用一个抽象概念“资源容器”（**Resource Container，**简称**Container**）表示**，Container是一个动态资源分配单位，它将内存、CPU资源封装在一起，从而限定每个任务使用的资源量。

应用程序管理器(Applications Manager)负责管理整个系统中所有应用程序，包括应用程序提交、与调度器协商资源以启动ApplicationMaster、监控ApplicationMaster运行状态并在失败时重新启动它等。

ApplicationMaster（AM）

用户提交的每个应用程序均包含一个AM，主要功能包括：

与RM调度器协商以获取资源（用Container表示）；

将得到的任务进一步分配给内部的任务(资源的二次分配)；

与NM通信以启动/停止任务；

监控所有任务运行状态，并在任务运行失败时重新为任务申请资源以重启任务。

NodeManager（NM）

NM是每个节点上的资源和任务管理器，一方面，它会定时地向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，它接收并处理来自AM的Container启动/停止等各种请求。

Container

Container是YARN中的资源抽象，它封装了某个节点上的内存、CPU资源，当AM向RM申请资源时，RM为AM返回的资源便是用Container表示。YARN会为每个任务分配一个Container，且该任务只能使用该Container中描述的资源。

相关配置（yarn-site.xml）

参数	默认值
yarn.nodemanager.resource.memory-mb	8192（MB)（每台服务nm服务器贡献的内存），工作中，要根据服务器的实际内存来调节。 比如服务器内存：64GB。给操作系统留

	出8G。还需要考虑这个服务器上是否还运行，比如Hbase。给Hbase留出16GB剩下的40GB留给yarn
yarn.nodemanager.resource.cpu-vcores	8,cpu核数，根据实际情况来配置，有几核就配置几个。
yarn.scheduler.minimum-allocation-mb	1024（MB），每个Container最小的使用内存量
yarn.scheduler.maximum-allocation-mb	8192（MB），每个Container最大的使用内存量
yarn.scheduler.minimum-allocation-vcores	1，每个Container最少的使用核数
yarn.scheduler.maximum-allocation-vcores	4，每个Container最多使用的核数
mapreduce.map.memory.mb	1024（MB），每个MapTask运行所有的内存大小。此参数如果在Container的下限和上限之间，就用设置的参数值。 如果不在上限和下限范围，就取下限或上限值
mapreduce.reduce.memory.mb	1024（MB）
yarn.resourcemanager.scheduler.class	org.apache.hadoop.yarn.server.resourcemanager.scheduler.capacity.CapacityScheduler 配置Yarn的调度器类型，默认是容器调度器。 另外两种调度器： ①FIFO 调度器 ②Fair 调度器

Hadoop2.0JVM重用及调优

hadoop2.0 uber功能

在 Yarn（Hadoop MapReduce v2）里提供了uber模式，开启这种模式可以让JVM重用。据Arun的说法，启用该功能能够让一些任务的执行效率提高2到3倍（“we've observed 2x-3x speedup for some jobs”）。

1) uber的原理：

Yarn的默认配置会禁用uber组件，即不允许JVM重用。我们先看看在这种情况下，Yarn是如何执行一个MapReduce job的。首先，Resource Manager里的Application Manager会为每一个application(比如一个用户提交的MapReduce Job)在NodeManager里面申请一个container，然后在该container里面启动一个Application Master。container在Yarn中是分配资源的容器(内存、cpu核数等)，它启动时便会相应启动一个JVM。此时，Application Master便陆续为application包含的每一个task(一个Map task或Reduce task)向Resource Manager申请一个container。等每得到一个container后，便要求该container所属的NodeManager将此container启动，然后就在这个container里面执行相应的task。等这个task执行完后，这个container便会被NodeManager收回，而container所拥有的JVM也相应地被退出。在这种情况下，可以看出**每一个JVM仅会执行一Task，JVM并未被重用。**

用户可以通过启用uber组件来允许JVM重用——即在同一个container里面依次执行多个task。在yarn-site.xml文件中，改变一下几个参数的配置即可启用uber的方法：

参数| 默认值 | 描述

- mapreduce.job.ubertask.enable | (false) | 是否启用user功能。**如果启用了该功能，则会将一个“小的application”的所有子task在同一个JVM里面执行**，达到JVM重用的目的。这个JVM便是负责该application的ApplicationMaster所用的JVM（运行在其container里）。那具体什么样的application算是“小的application"呢？下面几个参数便是用来定义何谓一个“小的application"

- mapreduce.job.ubertask.maxmaps | 9 | map任务数的阀值，如果一个application包含的map数小于该值的定义，那么该application就会被认为是一个小的application

- mapreduce.job.ubertask.maxreduces | 1 | reduce任务数的阀值，如果一个application包含的reduce数小于该值的定义，那么该application就会被认为是一个小的application。不过目前Yarn不支持该值大于1的情况
“CURRENTLY THE CODE CANNOT SUPPORT MORE THAN ONE REDUCE”

- mapreduce.job.ubertask.maxbytes | | application的输入大小的阀值。默认为dfs.block.size的值。当实际的输入大小部小于该值的设定，便会认为该application为一个小的application。

最后，我们来看当uber功能被启用的时候，Yarn是如何执行一个application的。首先，Resource Manager里的Application Manager会为每一个application在NodeManager里面申请一个container，然后在该container里面启动一个Application Master。containe启动时便会相应启动一个JVM。此时，如果uber功能被启用，并且该application被认为是一个“小的application”，那么

Application Master便会将该application包含的每一个task依次在这个container里的JVM里顺序执行，直到所有task被执行完

("WIth 'uber' mode enabled, you'll run everything within the container of the AM itself")。这样Application Master便不用再为每一个task向Resource Manager去申请一个单独的container，最终达到了 JVM重用（资源重用）的目的。

📖 **在yarn-site.xml里的配置示例：**

```
<!-- 开启uber模式（针对小作业的优化） -->
  <property>
    <name>mapreduce.job.ubertask.enable</name>
    <value>true</value>
  </property>

  <!-- 配置启动uber模式的最大map数 -->
  <property>
    <name>mapreduce.job.ubertask.maxmaps</name>
    <value>9</value>
  </property>

  <!-- 配置启动uber模式的最大reduce数 -->
  <property>
    <name>mapreduce.job.ubertask.maxreduces</name>
    <value>1</value>
  </property>
```

Hadoop小文件的处理方法

2018年7月20日 10:51

处理方式一：开启Hadoop的JVM重用机制，避免海量小文件（海量的map任务）带来的JVM频繁启停。

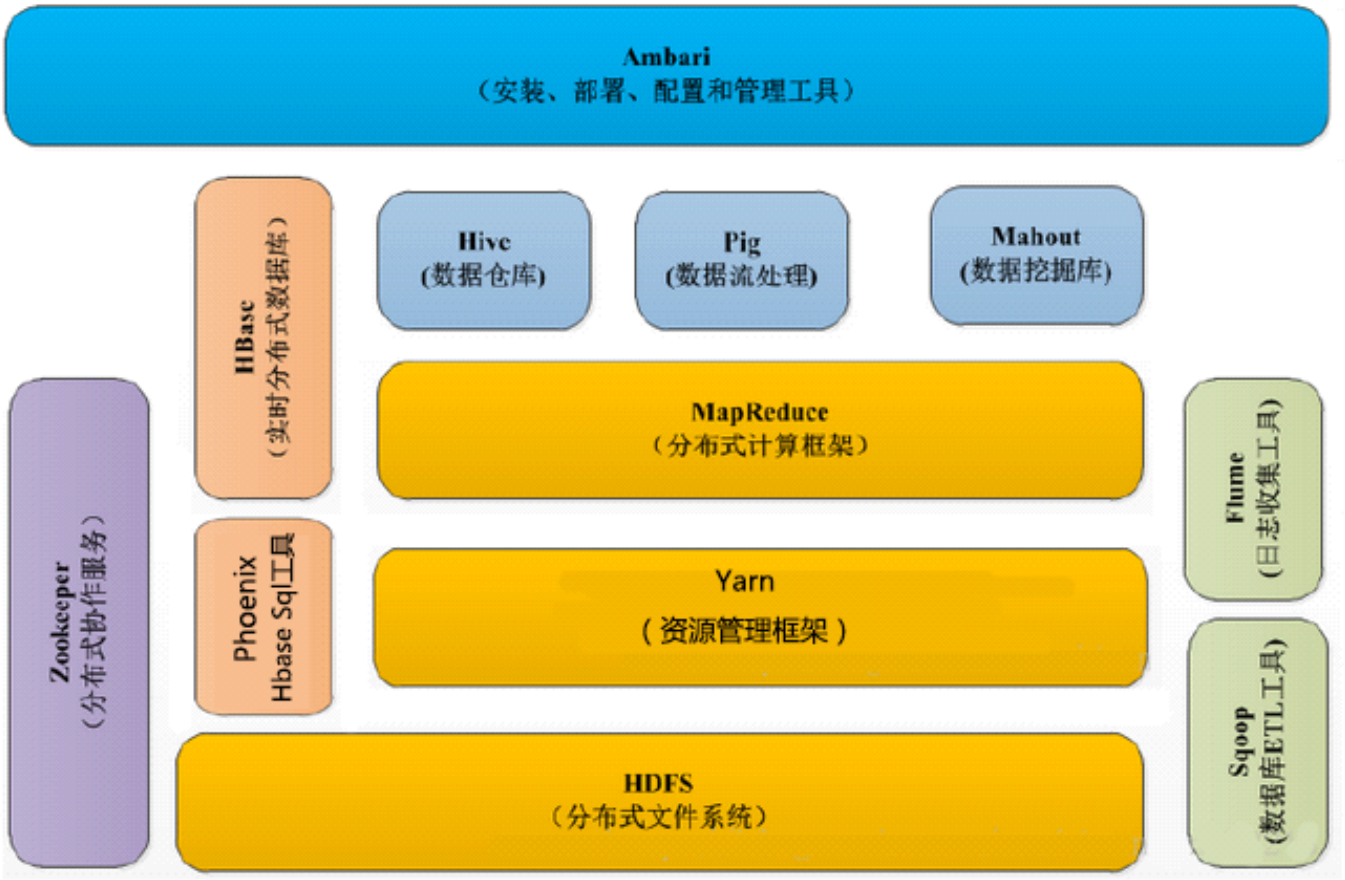
处理方式二：将多个小文件合成一个文件或合成少量文件，这样可以减少map的任务数量

代码示例：

```
.....

job.setInputFormatClass(CombineTextInputFormat.class);

CombineTextInputFormat.setMaxInputSplitSize(job, 179);

.....
```

Hadoop生态系统

Hadoop生态系统



1、HDFS（Hadoop分布式文件系统）

源自于Google的GFS论文，发表于2003年10月，HDFS是GFS克隆版。

HDFS是Hadoop体系中数据存储管理的基础。它是一个高度容错的系统，能检测和应对硬件故障，用于在低成本的通用硬件上运行。

HDFS简化了文件的一致性模型，通过流式数据访问，提供高吞吐量应用程序数据访问功能，适合带有大型数据集的应用程序。

它提供了一次写入多次读取的机制，数据以块的形式，同时分布在集群不同物理机器上。

2、Mapreduce（分布式计算框架）

源自于google的MapReduce论文，发表于2004年12月，Hadoop MapReduce是google MapReduce 克隆版。

MapReduce是一种分布式计算模型，用以进行大数据量的计算。它屏蔽了分布式计算框架细节，将计

算抽象成map和reduce两部分，

其中Map对数据集上的独立元素进行指定的操作，生成键-值对形式中间结果。Reduce则对中间结果中相同“键”的所有“值”进行规约，以得到最终结果。

MapReduce非常适合在大量计算机组成的分布式并行环境里进行数据处理。

3. HBASE（分布式列存数据库）

源自Google的Bigtable论文，发表于2006年11月，HBase是Google Bigtable克隆版

HBase是一个建立在HDFS之上，面向列的针对结构化数据的可伸缩、高可靠、高性能、分布式和面向列的动态模式数据库。

HBase采用了BigTable的数据模型：增强的稀疏排序映射表（Key/Value），其中，键由行关键字、列关键字和时间戳构成。

HBase提供了对大规模数据的随机、实时读写访问，同时，HBase中保存的数据可以使用MapReduce来处理，它将数据存储和并行计算完美地结合在一起。

4. Zookeeper（分布式协作服务）

源自Google的Chubby论文，发表于2006年11月，Zookeeper是Chubby克隆版

解决分布式环境下的数据管理问题：统一命名，状态同步，集群管理，配置同步等。

《The Chubby Lock Service for loosely coupled distributed system》

Hadoop的许多组件依赖于Zookeeper，它运行在计算机集群上面，用于管理Hadoop操作。

5. HIVE（数据仓库）

由facebook开源，最初用于解决海量结构化的日志数据统计问题。

Hive定义了一种类似SQL的查询语言(HQL),将SQL转化为MapReduce任务在Hadoop上执行。通常用于离线分析。

HQL用于运行存储在Hadoop上的查询语句，Hive让不熟悉MapReduce开发人员也能编写数据查询语句，然后这些语句被翻译为Hadoop上面的MapReduce任务。

6.Pig(ad-hoc脚本）

由yahoo!开源，设计动机是提供一种基于MapReduce的ad-hoc(计算在query时发生)数据分析工具

Pig定义了一种数据流语言—Pig Latin，它是MapReduce编程的复杂性的抽象,Pig平台包括运行环境和用于分析Hadoop数据集的脚本语言(Pig Latin)。

其编译器将Pig Latin翻译成MapReduce程序序列将脚本转换为MapReduce任务在Hadoop上执行。通常用于进行离线分析。

7.Sqoop(数据ETL/同步工具）

Sqoop是SQL-to-Hadoop的缩写，主要用于传统数据库和Hadoop之前传输数据。数据的导入和导出本质上是Mapreduce程序，充分利用了MR的并行化和容错性。

Sqoop利用数据库技术描述数据架构，用于在关系数据库、数据仓库和Hadoop之间转移数据。

8.Flume（日志收集工具）

Cloudera开源的日志收集系统，具有分布式、高可靠、高容错、易于定制和扩展的特点。

它将数据从产生、传输、处理并最终写入目标的路径的过程抽象为数据流，在具体的数据流中，数据源支持在Flume中定制数据发送方，从而支持收集各种不同协议数据。

同时，Flume数据流提供对日志数据进行简单处理的能力，如过滤、格式转换等。此外，Flume还具有能够将日志写往各种数据目标（可定制）的能力。

总的来说，Flume是一个可扩展、适合复杂环境的海量日志收集系统。当然也可以用于收集其他类型数据

9.Mahout（数据挖掘算法库）

Mahout起源于2008年，最初是Apache Lucent的子项目，它在极短的时间内取得了长足的发展，现在是Apache的顶级项目。

Mahout的主要目标是创建一些可扩展的机器学习领域经典算法的实现，旨在帮助开发人员更加方便快捷地创建智能应用程序。

Mahout现在已经包含了聚类、分类、推荐引擎（协同过滤）和频繁集挖掘等广泛使用的数据挖掘方法。

除了算法，Mahout还包含数据的输入/输出工具、与其他存储系统（如数据库、MongoDB 或Cassandra）集成等数据挖掘支持架构。

10. Yarn(分布式资源管理器）

YARN是下一代MapReduce，即MRv2，是在第一代MapReduce基础上演变而来的，主要是为了解

决原始Hadoop扩展性较差，不支持多计算框架而提出的。

Yarn是下一代 Hadoop 计算平台，yarn是一个通用的运行时框架，用户可以编写自己的计算框架，在该运行环境中运行。

用于自己编写的框架作为客户端的一个lib，在运用提交作业时打包即可。该框架为提供了以下几个组件：

- 资源管理：包括应用程序管理和机器资源管理
- 资源双层调度
- 容错性：各个组件均有考虑容错性
- 扩展性：可扩展到上万个节点

11. Phoenix（hbase sql接口）

Apache Phoenix 是HBase的SQL驱动，Phoenix 使得Hbase 支持通过JDBC的方式进行访问，并将你的SQL查询转换成Hbase的扫描和相应的动作。

12.Ambari（安装部署配置管理工具）

Apache Ambari 的作用来说，就是创建、管理、监视 Hadoop 的集群，是为了让 Hadoop 以及相关的大数据软件更容易使用的一个web工具。

就业：生产环境的服务器设置

2018年7月15日 23:23

浪潮产品 https://wap.zol.com.cn/406/405438/param.html

按每天最大的访问量1亿条记录来计算：

①每小时上报一次，我们将一小时之内产生的数据在HDFS上生成一个文件。

假设每小时产生1000万条数据，每条数据300字节

则每一小时生成的文件大小为：10000000*300/1024/1024=2861MB=2.8GB

②依据上面的假设，每天1亿的访问量，会生成10个2.8GB大小的文件。

③namenode的内存计算：namenode用到内存的地方就是管理元数据，

每个文件的元数据大小是150字节，如果每天仅生成10个文件，则每天吃掉的内存仅为1.5kb

但是如果1亿条数据，按1000条记录生成一个文件，而不是按1000万条记录生成一个文件，

则每天生成的文件数为：10万个文件，而10万个文件每天吃掉的内存为：14MB

如果一直不清理这些文件，则一个月namenode用掉的内存：448MB，一年吃掉的内存为：5.25GB

④关于Yarn及内存预留的问题，这个不太清除你们公司具体服务器的情况和集群规模，给不出具体建议下面仅做一个参考。

Reserved Memory = Reserved for stack memory + Reserved for HBase Memory (If HBase is on the same node)

比如：系统总内存126GB，一般预留给操作系统24GB，如果有Hbase再预留给Hbase24GB。剩下归Yarn调度管理

⑤至于服务器核数的问题，不用纠结，肯定是越多越好。关键是看公司财力了。

一般来说：

1.部门级的服务器通常在2至4个PIII Xeon（至强）处理器，适合中型企业（如金融、邮电等行业）作为数据中心、Web站点等应用。

2.企业级服务器属于高档服务器，通常普遍可支持4至8个PIII Xeon（至强）或P4 Xeon（至强）处理器，主要适用于需要处理大量数据、高处理速度和对可靠性要求极高的大型企业和重要行业（如金融、证券、交通、邮电、通信等行业）

推荐一款浪潮的服务器，价格在：30400--51600，仅作为参考：

CPU类型	Intel 至强E5-2600 v3
CPU型号	Xeon E5-2609 v3
CPU频率	1.9GHz
标配CPU数量	1颗
最大CPU数量	2颗
总线规格	QPI 6.4GT/s
CPU核心	六核（Haswell）
CPU线程数	六线程
内存类型	DDR4
内存容量	8GB
最大内存容量	640GB
存储硬盘接口类型	SATA/SAS
标配硬盘容量	2TB
硬盘描述	可支持7200转 3.5寸 SAS及SATA硬盘