尚硅谷大数据技术之 Hadoop（优化）

(作者：尚硅谷大数据研发部)

版本 V3.0

第 1 章 Hadoop 数据压缩

1.1 概述

压缩概述

尚硅谷

压缩技术能够有效减少底层存储系统（HDFS）读写字节数。压缩提高了网络带宽和磁盘空间的效率。在运行MR程序时，I/O操作、网络数据传输、 Shuffle和Merge要花大量的时间，尤其是数据规模很大和工作负载密集的情况下，因此，使用数据压缩显得非常重要。

鉴于磁盘I/O和网络带宽是Hadoop的宝贵资源，数据压缩对于节省资源、最小化磁盘I/O和网络传输非常有帮助。可以在任意MapReduce阶段启用压缩。不过，尽管压缩与解压操作的CPU开销不高，其性能的提升和资源的节省并非没有代价。

图 1-1 压缩概述

压缩策略和原则

尚硅谷

压缩是提高Hadoop运行效率的一种**优化策略。**

通过对Mapper、Reducer运行过程的数据进行压缩，以减少磁盘IO，提高MR程序运行速度。

注意：采用压缩技术减少了磁盘IO，但同时增加了CPU运算负担。所以，压缩特性运用得当能提高性能，但运用不当也可能降低性能。

压缩基本原则：

（1）运算密集型的job，少用压缩

（2）IO密集型的job，多用压缩

图 1-2 压缩概述

1.2 MR 支持的压缩编码

表 1-1

压缩格式	hadoop 自带？	算法	文件扩展名	是否可切分	换成压缩格式后，原来的程序是否需要修改
DEFLATE	是，直接使用	DEFLATE	.deflate	否	和文本处理一样，不需要修改
Gzip	是，直接使用	DEFLATE	.gz	否	和文本处理一样，不需要修改
bzip2	是，直接使用	bzip2	.bz2	是	和文本处理一样，不需要修改
LZO	否，需要安装	LZO	.lzo	是	需要建索引，还需要指定输入格式
Snappy	是，直接使用	Snappy	.snappy	否	和文本处理一样，不需要修改

为了支持多种压缩/解压缩算法，Hadoop 引入了编码/解码器，如下表所示。

表 1-2

压缩格式	对应的编码/解码器
DEFLATE	org.apache.hadoop.io.compress.DefaultCodec
gzip	org.apache.hadoop.io.compress.GzipCodec
bzip2	org.apache.hadoop.io.compress.BZip2Codec
LZO	com.hadoop.compression.lzo.LzopCodec
Snappy	org.apache.hadoop.io.compress.SnappyCodec

压缩性能的比较

表 1-3

压缩算法	原始文件大小	压缩文件大小	压缩速度	解压速度
gzip	8.3GB	1.8GB	17.5MB/s	58MB/s
bzip2	8.3GB	1.1GB	2.4MB/s	9.5MB/s
LZO	8.3GB	2.9GB	49.3MB/s	74.6MB/s

http://google.github.io/snappy/

On a single core of a Core i7 processor in 64-bit mode, Snappy compresses at about 250 MB/sec or more and decompresses at about 500 MB/sec or more.

1.3 压缩方式选择

1.3.1 Gzip 压缩

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；大部分Linux系统都自带Gzip命令，使用方便。

缺点：不支持Split。

应用场景：当每个文件压缩之后在130M以内的（1个块大小内），都可以考虑用Gzip压缩格式。例如说一天或者一个小时的日志压缩成一个Gzip文件。

图 1-3 Gzip 压缩概述

1.3.2 Bzip2 压缩

Bzip2压缩

优点：支持Split；具有很高的压缩率，比Gzip压缩率都高；Hadoop本身自带，使用方便。

缺点：压缩/解压速度慢。

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；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的情况。

图 1-4 Bzip2 压缩概述

1.3.3 Lzo 压缩

Lzo压缩

优点：压缩/解压速度也比较快，合理的压缩率；支持Split，是Hadoop中最流行的压缩格式；可以在Linux系统下安装lzop命令，使用方便。

缺点：压缩率比Gzip要低一些；Hadoop本身不支持，需要安装；在应用中对Lzo格式的文件需要做一些特殊处理（为了支持Split需要建索引，还需要指定InputFormat为Lzo格式）。

应用场景：一个很大的文本文件，压缩之后还大于200M以上的可以考虑，而且单个文件越大，Lzo优点越越明显。

图 1-5 LZO 压缩概述

1.3.4 Snappy 压缩

Snappy压缩

优点：高速压缩速度和合理的压缩率。

缺点：不支持Split；压缩率比Gzip要低；Hadoop本身不支持，需要安装。

应用场景：当MapReduce作业的Map输出的数据比较大的时候，作为Map到Reduce的中间数据的压缩格式；或者作为一个MapReduce作业的输出和另外一个MapReduce作业的输入。

图 1-6 Snappy 压缩概述

1.4 压缩位置选择

压缩可以在 MapReduce 作用的任意阶段启用，如图 1-7 所示。

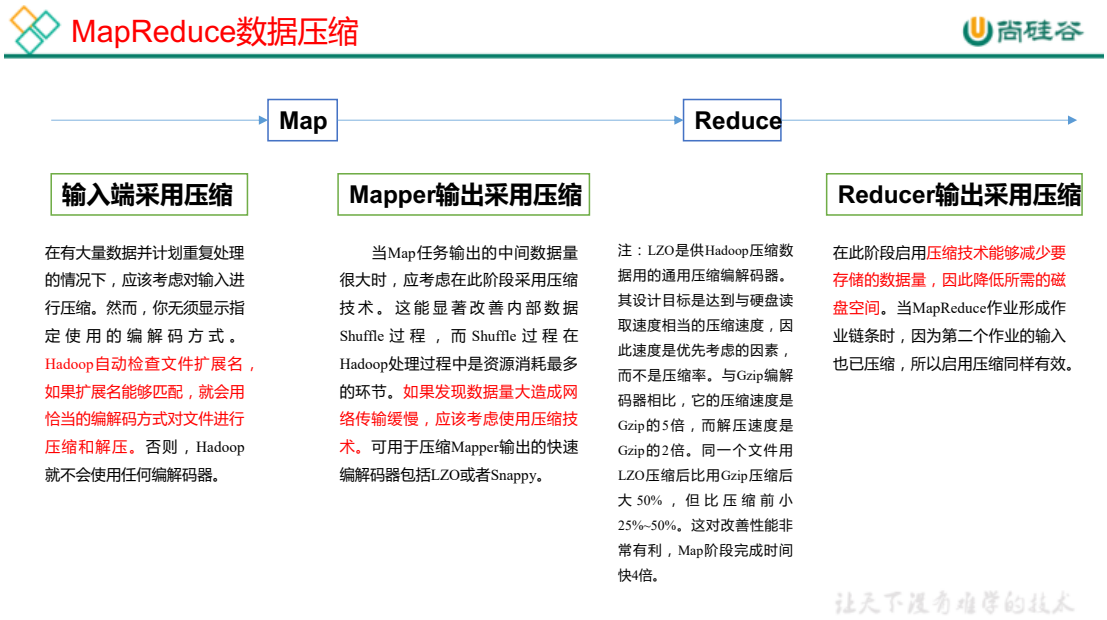

图 1-7 MapReduce 数据压缩

1.5 压缩参数配置

要在 Hadoop 中启用压缩，可以配置如下参数：

表 1-4 配置参数

参数	默认值	阶段	建议
io.compression.codecs（在 core-site.xml 中配置）			Hadoop 使用文件扩展名判断是否支持某种编解码器
mapreduce.map.output.compress（在 mapred-site.xml 中配置）	false	mapper 输出	这个参数设为 true 启用压缩
mapreduce.map.output.compress.codec（在 mapred-site.xml 中配置）	org.apache.hadoop.io.compress.DefaultCodec	mapper 输出	企业多使用 LZO 或 Snappy 编解码

			器在此阶段压缩数据
mapreduce.output.fileoutputformat.compress（在 mapred-site.xml 中配置）	false	reducer 输出	这个参数设为 true 启用压缩
mapreduce.output.fileoutputformat.compress.codec（在 mapred-site.xml 中配置）	org.apache.hadoop.io.compress.DefaultCodec	reducer 输出	使用标准工具或者编解码器，如 gzip 和 bzip2
mapreduce.output.fileoutputformat.compress.type（在 mapred-site.xml 中配置）	RECORD	reducer 输出	SequenceFile 输出使用的压缩类型：NONE 和 BLOCK

1.6 压缩实操案例

1.6.1 数据流的压缩和解压缩

数据流的压缩和解压缩 尚硅谷

CompressionCodec有两个方法可以用于轻松地压缩或解压缩数据。

要想对正在被写入一个输出流的数据进行压缩，我们可以使用createOutputStream(OutputStreamout)方法创建一个CompressionOutputStream，将其以压缩格式写入底层的流。

相反，要想对从输入流读取而来的数据进行解压缩，则调用createInputStream(InputStreamin)函数，从而获得一个CompressionInputStream，从而从底层的流读取未压缩的数据。

图 1-8 数据流的压缩和解压缩

测试一下如下压缩方式：

表 1-5

DEFLATE	org.apache.hadoop.io.compress.DefaultCodec
gzip	org.apache.hadoop.io.compress.GzipCodec
bzip2	org.apache.hadoop.io.compress.BZip2Codec

```
package com.atguigu.mapreduce.compress;
import java.io.File;
import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.FileOutputStream;
import java.io.IOException;
import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.IOUtils;
import org.apache.hadoop.io.compress.CompressionCodec;
import org.apache.hadoop.io.compress.CompressionCodecFactory;
import org.apache.hadoop.io.compress.CompressionInputStream;
import org.apache.hadoop.io.compress.CompressionOutputStream;
import org.apache.hadoop.util.ReflectionUtils;

public class TestCompress {

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

    compress("e:/hello.txt","org.apache.hadoop.io.compress.BZip2Co
dec");
//     decompress("e:/hello.txt.bz2","e:/hello.txt");
    }

    // 1、压缩
    private static void compress(String filename, String method)
throws Exception {

        // （1）获取输入流
        FileInputStream fis = new FileInputStream(new
File(filename));

        Class codecClass = Class.forName(method);

        CompressionCodec codec = (CompressionCodec)
ReflectionUtils.newInstance(codecClass, new Configuration());

        // （2）获取输出流
        FileOutputStream fos = new FileOutputStream(new
File(filename + codec.getDefaultExtension()));
      CompressionOutputStream cos = codec.createOutputStream(fos);

        // （3）流的对拷
        IOUtils.copyBytes(fis, cos, 1024*1024*5, false);

       // （4）关闭资源
        cos.close();
        fos.close();
        fis.close();
```

```
    }

    // 2、解压缩
    private static void decompress(String filename,String dest) throws FileNotFoundException, IOException {

        // （0）校验是否能解压缩
        CompressionCodecFactory factory = new CompressionCodecFactory(new Configuration());

        CompressionCodec codec = factory.getCodec(new Path(filename));

        if (codec == null) {
            System.out.println("cannot find codec for file " + filename);
            return;
        }

        // （1）获取输入流
        CompressionInputStream cis = codec.createInputStream(new FileInputStream(new File(filename)));

        // （2）获取输出流
        FileOutputStream fos = new FileOutputStream(new File(dest));

        // （3）流的对拷
        IOUtils.copyBytes(cis, fos, 1024*1024*5, false);

        // （4）关闭资源
        cis.close();
        fos.close();
    }
}
```

1.6.2 Map 输出端采用压缩

即使你的 MapReduce 的输入输出文件都是未压缩的文件，你仍然可以对 Map 任务的中间结果输出做压缩，因为它要写在硬盘并且通过网络传输到 Reduce 节点，对其压缩可以提高很多性能，这些工作只要设置两个属性即可，我们来看下代码怎么设置。

1．给大家提供的 Hadoop 源码支持的压缩格式有：BZip2Codec 、DefaultCodec

```
package com.atguigu.mapreduce.compress;
import java.io.IOException;
import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.io.compress.BZip2Codec;
import org.apache.hadoop.io.compress.CompressionCodec;
import org.apache.hadoop.io.compress.GzipCodec;
import org.apache.hadoop.mapreduce.Job;
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;
```

```
public class WordCountDriver {

    public static void main(String[] args) throws IOException,
ClassNotFoundException, InterruptedException {

        Configuration configuration = new Configuration();

        // 开启 map 端输出压缩
    configuration.setBoolean("mapreduce.map.output.compress",
true);
        // 设置 map 端输出压缩方式
    configuration.setClass("mapreduce.map.output.compress.codec"
, BZip2Codec.class, CompressionCodec.class);

        Job job = Job.getInstance(configuration);

        job.setJarByClass(WordCountDriver.class);

        job.setMapperClass(WordCountMapper.class);
        job.setReducerClass(WordCountReducer.class);

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

        job.setOutputKeyClass(Text.class);
        job.setOutputValueClass(IntWritable.class);

        FileInputFormat.setInputPaths(job, new Path(args[0]));
        FileOutputFormat.setOutputPath(job, new Path(args[1]));

        boolean result = job.waitForCompletion(true);

        System.exit(result ? 1 : 0);
    }
}
```

2．Mapper 保持不变

```
package com.atguigu.mapreduce.compress;
import java.io.IOException;
import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.LongWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Mapper;

public class WordCountMapper extends Mapper<LongWritable, Text,
Text, IntWritable>{

    Text k = new Text();
    IntWritable v = new IntWritable(1);

    @Override
    protected void map(LongWritable key, Text value, Context
context)throws IOException, InterruptedException {

        // 1 获取一行
        String line = value.toString();
```

```
        // 2 切割
        String[] words = line.split(" ");

        // 3 循环写出
        for(String word:words){
            k.set(word);
            context.write(k, v);
        }
    }
}
```

3．Reducer 保持不变

```
package com.atguigu.mapreduce.compress;
import java.io.IOException;
import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Reducer;

public class WordCountReducer extends Reducer<Text, IntWritable,
Text, IntWritable>{

    IntWritable v = new IntWritable();

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

        int sum = 0;

        // 1 汇总
        for(IntWritable value:values){
            sum += value.get();
        }

        v.set(sum);

        // 2 输出
        context.write(key, v);
    }
}
```

1.6.3 Reduce 输出端采用压缩

基于 WordCount 案例处理。

1．修改驱动

```
package com.atguigu.mapreduce.compress;
import java.io.IOException;
import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.io.compress.BZip2Codec;
import org.apache.hadoop.io.compress.DefaultCodec;
import org.apache.hadoop.io.compress.GzipCodec;
```

```
import org.apache.hadoop.io.compress.Lz4Codec;
import org.apache.hadoop.io.compress.SnappyCodec;
import org.apache.hadoop.mapreduce.Job;
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;

public class WordCountDriver {

   public static void main(String[] args) throws IOException,
ClassNotFoundException, InterruptedException {

       Configuration configuration = new Configuration();

       Job job = Job.getInstance(configuration);

       job.setJarByClass(WordCountDriver.class);

       job.setMapperClass(WordCountMapper.class);
       job.setReducerClass(WordCountReducer.class);

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

       job.setOutputKeyClass(Text.class);
       job.setOutputValueClass(IntWritable.class);

       FileInputFormat.setInputPaths(job, new Path(args[0]));
       FileOutputFormat.setOutputPath(job, new Path(args[1]));

       // 设置 reduce 端输出压缩开启
       FileOutputFormat.setCompressOutput(job, true);

       // 设置压缩的方式
       FileOutputFormat.setOutputCompressorClass(job,
BZip2Codec.class);
//     FileOutputFormat.setOutputCompressorClass(job,
GzipCodec.class);
//     FileOutputFormat.setOutputCompressorClass(job,
DefaultCodec.class);

       boolean result = job.waitForCompletion(true);

       System.exit(result?1:0);
   }
}
```

2．Mapper 和 Reducer 保持不变

第 2 章 Hadoop 企业优化

2.1 MapReduce 跑的慢的原因

MapReduce 跑的慢的原因

MapReduce 程序效率的瓶颈在于两点：

1 □□□□□

CPU、内存、磁盘健康、网络

2 I/O □□□□

（1）数据倾斜

（2）Map和Reduce数设置不合理

（3）Map运行时间太长，导致Reduce等待过久

（4）小文件过多

（5）大量的不可分块的超大文件

（6）Spill次数过多

（7）Merge次数过多等。

图 2-1 MapReduce 跑的慢的原因

2.2 MapReduce 优化方法

MapReduce 优化方法主要从六个方面考虑：数据输入、Map 阶段、Reduce 阶段、IO 传输、数据倾斜问题和常用的调优参数。

2.2.1 数据输入

MapReduce优化方法

6.2.1 数据输入

（1）合并小文件：在执行MR任务前将小文件进行合并，大量的小文件会产生大量的Map任务，增大Map任务装载次数，而任务的装载比较耗时，从而导致MR运行较慢。

（2）采用CombineTextInputFormat来作为输入，解决输入端大量小文件场景。

图 2-2 MapReduce 优化-数据输入角度

2.2.2 Map 阶段

6.2.2 Map阶段

（1）减少溢写（Spill）次数：通过调整io.sort.mb及sort.spill.percent参数值，增大触发Spill的内存上限，减少Spill次数，从而减少磁盘IO。

（2）减少合并（Merge）次数：通过调整io.sort.factor参数，增大Merge的文件数目，减少Merge的次数，从而缩短MR处理时间。

（3）在Map之后，**不影响业务逻辑前提下，先进行Combine处理**，减少 I/O。

图 2-3 MapReduce 优化- Map 阶段

2.2.3 Reduce 阶段

6.2.3 Reduce阶段

（1）合理设置Map和Reduce数：两个都不能设置太少，也不能设置太多。太少，会导致Task等待，延长处理时间；太多，会导致Map、Reduce任务间竞争资源，造成处理超时等错误。

（2）设置Map、Reduce共存：调整slowstart.completedmaps参数，使Map运行到一定程度后，Reduce也开始运行，减少Reduce的等待时间。

（3）规避使用Reduce：因为Reduce在用于连接数据集的时候将会产生大量的网络消耗。

图 2-4 MapReduce 优化 – Reduce 阶段

6.2.3 Reduce阶段

（4）合理设置Reduce端的Buffer：默认情况下，数据达到一个阈值的时候，Buffer中的数据就会写入磁盘，然后Reduce会从磁盘中获得所有的数据。也就是说，Buffer和Reduce是没有直接关联的，中间多次写磁盘->读磁盘的过程，既然有这个弊端，那么就可以通过参数来配置，使得Buffer中的一部分数据可以直接输送到Reduce，从而减少IO开销：mapreduce.reduce.input.buffer.percent，默认为0.0。当值大于0的时候，会保留指定比例的内存读Buffer中的数据直接拿给Reduce使用。这样一来，设置Buffer需要内存，读取数据需要内存，Reduce计算也要内存，所以要根据作业的运行情况进行调整。

图 2-5 MapReduce 优化 – Reduce 阶段

2.2.4 I/O 传输

6.2.4 IO传输

1）采用数据压缩的方式，减少网络IO的的时间。安装Snappy和LZO压缩编码器。

2）使用SequenceFile二进制文件。

图 2-6 MapReduce 优化 – IO 传输角度

2.2.5 数据倾斜问题

MapReduce优化方法

6.2.5 数据倾斜问题

1 □□□□□□

数据频率倾斜——某一个区域的数据量要远远大于其他区域。

数据大小倾斜——部分记录的大小远远大于平均值。

图 2-7 MapReduce 优化 – 数据倾斜角度

MapReduce优化方法

2 □□□□□□□□□□

方法1：抽样和范围分区

可以通过对原始数据进行抽样得到的结果集来预设分区边界值。

方法2：自定义分区

基于输出键的背景知识进行自定义分区。例如，如果Map输出键的单词来源于一本书。且其中某几个专业词汇较多。那么就可以自定义分区将这这些专业词汇发送给固定的一部分Reduce实例。而将其他的都发送给剩余的Reduce实例。

方法3：Combine

使用Combine可以大量地减小数据倾斜。在可能的情况下，Combine的目的就是聚合并精简数据。

方法4：采用Map Join，尽量避免Reduce Join。

图 2-8 MapReduce 优化 – 数据倾斜角度

2.2.6 常用的调优参数

1．资源相关参数

（1）以下参数是在用户自己的 MR 应用程序中配置就可以生效（mapred-default.xml）

表 2-1

配置参数	参数说明
mapreduce.map.memory.mb	一个 MapTask 可使用的资源上限（单位:MB），默认为 1024。如果 MapTask 实际使用的资源量超过该值，则会被强制杀死。
mapreduce.reduce.memory.mb	一个 ReduceTask 可使用的资源上限（单位:MB），默认为 1024。如果 ReduceTask 实际使用的资源量超过该值，则会被强制杀死。
mapreduce.map.cpu.vcores	每个 MapTask 可使用的最多 cpu core 数目，默认值: 1
mapreduce.reduce.cpu.vcores	每个 ReduceTask 可使用的最多 cpu core 数目，默认值: 1
mapreduce.reduce.shuffle.parallelcopies	每个 Reduce 去 Map 中取数据的并行数。默认值是 5
mapreduce.reduce.shuffle.merge.percent	Buffer 中的数据达到多少比例开始写入磁盘。默认值 0.66
mapreduce.reduce.shuffle.input.buffer.percent	Buffer 大小占 Reduce 可用内存的比例。默认值 0.7
mapreduce.reduce.input.buffer.percent	指定多少比例的内存用来存放 Buffer 中的数据，默认值是 0.0

（2）应该在 YARN 启动之前就配置在服务器的配置文件中才能生效（yarn-default.xml）

表 2-2

配置参数	参数说明
yarn.scheduler.minimum-allocation-mb	给应用程序 Container 分配的最小内存，默认值：1024
yarn.scheduler.maximum-allocation-mb	给应用程序 Container 分配的最大内存，默认值：8192
yarn.scheduler.minimum-allocation-vcores	每个 Container 申请的最小 CPU 核数，默认值：1
yarn.scheduler.maximum-allocation-vcores	每个 Container 申请的最大 CPU 核数，默认值：4
yarn.nodemanager.resource.memory-mb	给 Containers 分配的最大物理内存，默认值：8192

（3）Shuffle 性能优化的关键参数，应在 YARN 启动之前就配置好（mapred-default.xml）

表 2-3

配置参数	参数说明
mapreduce.task.io.sort.mb	Shuffle 的环形缓冲区大小，默认 100m
mapreduce.map.sort.spill.percent	环形缓冲区溢出的阈值，默认 80%

2．容错相关参数(MapReduce 性能优化)

表 2-4

配置参数	参数说明
mapreduce.map.maxattempts	每个 Map Task 最大重试次数，一旦重试参数超过该值，则认为 Map Task 运行失败，默认值：4。
mapreduce.reduce.maxattempts	每个 Reduce Task 最大重试次数，一旦重试参数超过该值，则认为 Map Task 运行失败，默认值：4。
mapreduce.task.timeout	Task 超时时间，经常需要设置的一个参数，该参数表达的意思为：如果一个 Task 在一定时间内没有任何进入，即不会读取新的数据，也没有输出数据，则认为该 Task 处于 Block 状态，可能是卡住了，也许永远会卡住，为了防止因为用户程序永远 Block 住不退出，则强制设置了一个该超时时间（单位毫秒），默认是 600000。如果你的程序对每条输入数据的处理时间过长（比如会访问数据库，通过网络拉取数据等），建议将该参数调大，该参数过小常出现的错误提示是“ AttemptID:attempt_14267829456721_123456_m_000224_0 Timed out after 300 secsContainer killed by the ApplicationMaster.”。

2.3 HDFS 小文件优化方法

2.3.1 HDFS 小文件弊端

HDFS 上每个文件都要在 NameNode 上建立一个索引，这个索引的大小约为 150byte，这样当小文件比较多的时候，就会产生很多的索引文件，一方面会大量占用 NameNode 的内存空间，另一方面就是索引文件过大使得索引速度变慢。

2.3.2 HDFS 小文件解决方案

1) 小文件优化的方向：

（1）在数据采集的时候，就将小文件或小批数据合成大文件再上传 HDFS。

（2）在业务处理之前，在 HDFS 上使用 MapReduce 程序对小文件进行合并。

（3）在 MapReduce 处理时，可采用 CombineTextInputFormat 提高效率。

（4）开启 uber 模式，实现 jvm 重用

2) Hadoop Archive

是一个高效的将小文件放入 HDFS 块中的文件存档工具，能够将多个小文件打包成一个 HAR 文件，从而达到减少 NameNode 的内存使用

3) SequenceFile

SequenceFile 是由一系列的二进制 k/v 组成，如果为 key 为文件名，value 为文件内容，

可将大批小文件合并成一个大文件

4) CombineTextInputFormat

CombineTextInputFormat 用于将多个小文件在切片过程中生成一个单独的切片或者少量的切片。

5) 开启 uber 模式，实现 jvm 重用。默认情况下，每个 Task 任务都需要启动一个 jvm 来运行，如果 Task 任务计算的数据量很小，我们可以让同一个 Job 的多个 Task 运行在一个 Jvm 中，不必为每个 Task 都开启一个 Jvm.

开启 uber 模式，在 mapred-site.xml 中添加如下配置

```
<!--  开启 uber 模式 -->
<property>
  <name>mapreduce.job.ubertask.enable</name>
  <value>true</value>
</property>
<!-- uber 模式中最大的 mapTask 数量,可向下修改  -->
<property>
  <name>mapreduce.job.ubertask.maxmaps</name>
  <value>9</value>
</property>
<!-- uber 模式中最大的 reduce 数量，可向下修改 -->
<property>
  <name>mapreduce.job.ubertask.maxreduces</name>
  <value>1</value>
</property>
<!-- uber 模式中最大的输入数据量，如果不配置，则使用 dfs.blocksize 的值，可向下修改 -->
<property>
  <name>mapreduce.job.ubertask.maxbytes</name>
  <value></value>
</property>
```

6) 配置 mapreduce.job.jvm.numtasks 参数实现在一个 Jvm 中运行多个 Task . 如果设置为 -1 ,这没有数量限制。 一般设置在 10-20 之间.