

尚硅谷大数据技术之 Hadoop (优化&新特性)

(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



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。

让天下没有难学的技术

压缩策略和原则



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

让天下没有难学的技术

1.2 MR 支持的压缩编码

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。

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

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/>

Snappy is a compression/decompression library. It **does not aim for maximum compression**, or compatibility with any other compression library; instead, it **aims for very high speeds** and reasonable compression. For instance, compared to the fastest mode of zlib, Snappy is an order of magnitude faster for most inputs, but the resulting compressed files are anywhere from 20% to 100% bigger. 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



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。

让天下没有难学的技术

1.3.2 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。

让天下没有难学的技术

1.3.3 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。

让天下没有难学的技术

1.3.4 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。

让天下没有难学的技术

1.4 压缩位置选择

压缩可以在 MapReduce 作用的任意阶段启用。

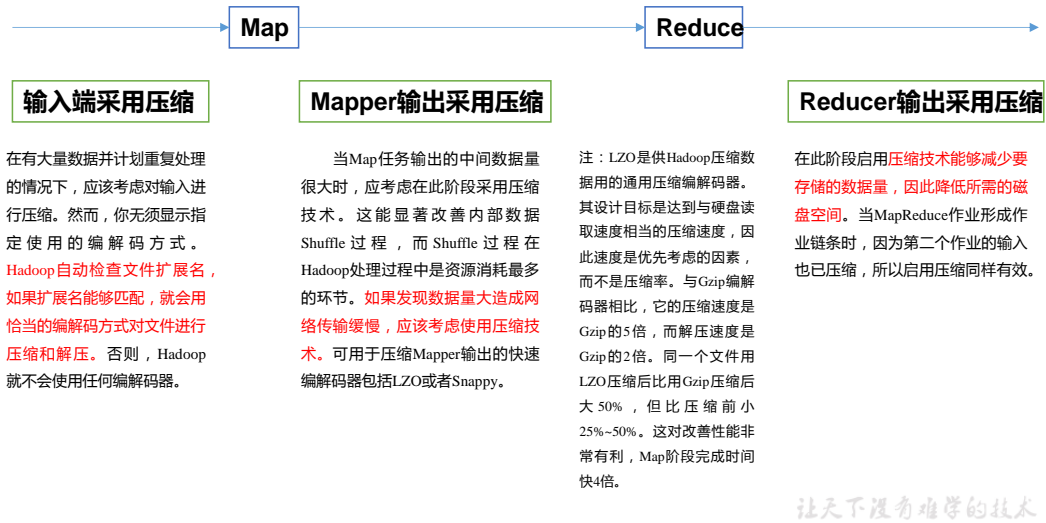


图 MapReduce 数据压缩

1.5 压缩参数配置

要在 Hadoop 中启用压缩，可以配置如下参数：

参数	默认值	阶段	建议
<code>io.compression.codecs</code> (在 <code>core-site.xml</code> 中配置)	无，这个需要在命令行输入 <code>hadoop checknative</code> 查看	输入压缩	Hadoop 使用文件扩展名判断是否支持某种编解码器
<code>mapreduce.map.output.compress</code> (在 <code>mapred-site.xml</code> 中配置)	false	mapper 输出	这个参数设为 true 启用压缩
<code>mapreduce.map.output.compress.codec</code> (在 <code>mapred-site.xml</code> 中配置)	<code>org.apache.hadoop.io.compress.DefaultCodec</code>	mapper 输出	企业多使用 LZO 或 Snappy 编解码器在此阶段压缩数据
<code>mapreduce.output.fileoutputformat.compress</code> (在 <code>mapred-site.xml</code> 中配置)	false	reducer 输出	这个参数设为 true 启用压缩
<code>mapreduce.output.fileoutputformat.compress.codec</code> (在 <code>mapred-site.xml</code> 中配置)	<code>org.apache.hadoop.io.compress.DefaultCodec</code>	reducer 输出	使用标准工具或者编解码器，如 <code>gzip</code> 和 <code>bzip2</code>
<code>mapreduce.output.fileoutputformat.compress.type</code> (在 <code>mapred-site.xml</code> 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



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。

让天下没有难学的技术

测试一下如下压缩方式：

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 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 java.io.File;
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.IOException;

public class TestCompress {
    public static void main(String[] args) throws IOException {
        compress("D:\\input\\inputcompression\\JaneEyre.txt"
, "org.apache.hadoop.io.compress.BZip2Codec");
        //decompress("D:\\input\\inputcompression\\JaneEyre.txt.bz2");
    }
    //压缩
    private static void compress(String filename, String method) throws
IOException {
        //1 获取输入流
```

```
FileInputStream fis = new FileInputStream(new File(filename));

//2 获取输出流
//获取压缩编解码器 codec
CompressionCodecFactory factory = new CompressionCodecFactory(new
Configuration());
CompressionCodec codec = factory.getCodecByName(method);

//获取普通输出流,文件后面需要加上压缩后缀
FileOutputStream fos = new FileOutputStream(new File(filename +
codec.getDefaultExtension()));
//获取压缩输出流,用压缩解码器对 fos 进行压缩
CompressionOutputStream cos = codec.createOutputStream(fos);

//3 流的对拷
IOUtils.copyBytes(fis,cos,new Configuration());

//4 关闭资源
IOUtils.closeStream(cos);
IOUtils.closeStream(fos);
IOUtils.closeStream(fis);
}

//解压缩
private static void decompress(String filename) throws 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
FileInputStream fis = new FileInputStream(new File(filename));
CompressionInputStream cis = codec.createInputStream(fis);

//2 获取输出流
FileOutputStream fos = new FileOutputStream(new File(filename +
".decode"));

//3 流的对拷
IOUtils.copyBytes(cis,fos,new Configuration());

//4 关闭资源
IOUtils.closeStream(fos);
IOUtils.closeStream(cis);
IOUtils.closeStream(fis);
}
}
```

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 conf = new Configuration();

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

        Job job = Job.getInstance(conf);

        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 ? 0 : 1);
    }
}
```

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 conf = new Configuration();

        Job job = Job.getInstance(conf);

        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?0:1);
    }
}
```

2) Mapper 和 Reducer 保持不变 (详见 1.6.2)

第 2 章 Hadoop 企业优化

2.1 MapReduce 跑的慢的原因



MapReduce 程序效率的瓶颈在于两点：

1. 计算机性能
CPU、内存、磁盘健康、网络
2. I/O 操作优化
 - (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) 大量的不可切片的超大压缩文件
 - (6) Spill次数过多
 - (7) Merge次数过多等。

让天下没有难学的技术

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



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。

让天下没有难学的技术

2.2.2 Map 阶段



Map阶段

(1) **减少溢写 (Spill) 次数**：通过调整mapreduce.task.io.sort.mb及mapreduce.map.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mapreduce.task.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。

让天下没有难学的技术

2.2.3 Reduce 阶段



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mapreduce.job.reduce.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。

让天下没有难学的技术

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。

让天下没有难学的技术

2.2.4 I/O 传输

IO传输

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

2) **使用SequenceFile二进制文件。**

让天下没有难学的技术

2.2.5 数据倾斜问题

MapReduce 优化方法

数据倾斜问题

1. 数据倾斜现象

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。

让天下没有难学的技术

MapReduce 优化方法

2. 减少数据倾斜的方法

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：Combiner

使用 Combiner 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。

让天下没有难学的技术

2.3 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)

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)

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：32
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)

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)

配置参数	参数说明
mapreduce.map.maxattempts	每个 Map Task 最大重试次数，一旦重试次数超过该值，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，一旦重试次数超过该值，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（10 分钟）。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.”。
--

2.4 Hadoop 小文件优化方法

2.4.1 Hadoop 小文件弊端

HDFS 上每个文件都要在 NameNode 上创建对应的元数据，这个元数据的大小约为 150byte，这样当小文件比较多时，就会产生很多的元数据文件，一方面会大量占用 NameNode 的内存空间，另一方面就是元数据文件过多，使得寻址索引速度变慢。

小文件过多，在进行 MR 计算时，会生成过多切片，需要启动过多的 MapTask。每个 MapTask 处理的数据量小，导致 MapTask 的处理时间比启动时间还小，白白消耗资源。

2.4.2 Hadoop 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，默认使用 dfs.blocksize 的值，可向下修改 -->
<property>
  <name>mapreduce.job.ubertask.maxbytes</name>
  <value></value>
</property>
```

第 3 章 Hadoop 新特性

3.1 Hadoop2.x 新特性

3.1.1 集群间数据拷贝

1) scp 实现两个远程主机之间的文件复制

```
scp -r hello.txt root@hadoop103:/user/atguigu/hello.txt // 推 push
```

```
scp -r root@hadoop103:/user/atguigu/hello.txt hello.txt // 拉 pull
```

```
scp -r root@hadoop103:/user/atguigu/hello.txt root@hadoop104:/user/atguigu //是通过本
```

地主机中转实现两个远程主机的文件复制；如果在两个远程主机之间 ssh 没有配置的情况下可以使用该方式。

2) 采用 distcp 命令实现两个 Hadoop 集群之间的递归数据复制

```
[atguigu@hadoop102 hadoop-3.1.3]$ bin/hadoop distcp
hdfs://hadoop102:9820/user/atguigu/hello.txt
hdfs://hadoop105:9820/user/atguigu/hello.txt
```

3.1.2 小文件存档



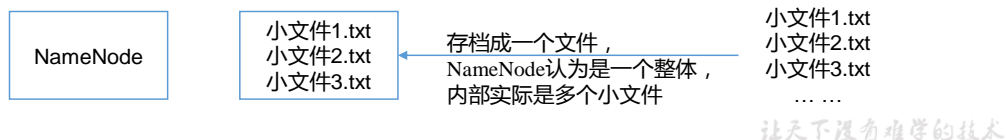
小文件存档

1、HDFS存储小文件弊端

每个文件均按块存储，每个块的元数据存储于NameNode的内存中，因此HDFS存储小文件会非常低效。因为大量的文件会耗尽NameNode中的大部分内存。但注意，存储小文件所需要的磁盘容量和数据块的大小无关。例如，一个1MB的文件设置为128MB的块存储，实际使用的是1MB的磁盘空间，而不是128MB。

2、解决存储小文件办法之一

HDFS存档文件或HAR文件，是一个更高效的文件存档工具，它将文件存入HDFS块，在减少NameNode内存使用的同时，允许对文件进行透明的访问。具体说来，HDFS存档文件对内还是一个一个独立文件，对NameNode而言却是一个整体，减少了NameNode的内存。



让天下没有难学的技术

1) 案例实操

(1) 需要启动 YARN 进程

```
[atguigu@hadoop102 hadoop-3.1.3]$ start-yarn.sh
```

(2) 归档文件

把/user/atguigu/input 目录里面的所有文件归档成一个叫 input.har 的归档文件，并把归档后文件存储到/user/atguigu/output 路径下。

```
[atguigu@hadoop102 hadoop-3.1.3]$ hadoop archive -archiveName input.har -p /user/atguigu/input /user/atguigu/output
```

(3) 查看归档

```
[atguigu@hadoop102 hadoop-3.1.3]$ hadoop fs -ls /user/atguigu/output/input.har
[atguigu@hadoop102 hadoop-3.1.3]$ hadoop fs -ls har:///user/atguigu/output/input.har
```

(4) 解归档文件

```
[atguigu@hadoop102 hadoop-3.1.3]$ hadoop fs -cp har:///user/atguigu/output/input.har/* /user/atguigu
```

3.1.3 回收站

开启回收站功能，可以将删除的文件在不超时的情况下，恢复原数据，起到防止误删除、备份等作用。

1) 回收站参数设置及工作机制

一、开启回收站功能参数说明：

- 1、默认值fs.trash.interval=0，0表示禁用回收站;其他值表示设置文件的存活时间。
- 2、默认值fs.trash.checkpoint.interval=0，检查回收站的间隔时间。如果该值为0，则该值设置和fs.trash.interval的参数值相等。
- 3、要求fs.trash.checkpoint.interval<=fs.trash.interval。

二、回收站工作机制：

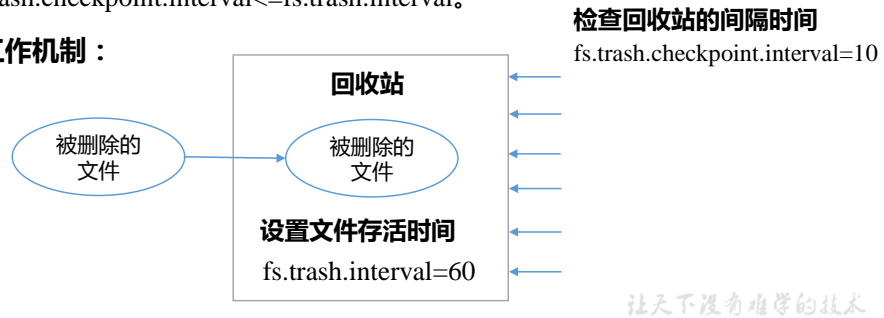


图 回收站

2) 启用回收站

修改 core-site.xml，配置垃圾回收时间为 1 分钟。

```
<property>
  <name>fs.trash.interval</name>
  <value>1</value>
</property>
```

3) 查看回收站

回收站目录在 hdfs 集群中的路径：/user/atguigu/.Trash/....

4) 通过程序删除的文件不会经过回收站，需要调用 moveToTrash()才进入回收站

```
Trash trash = New Trash(conf);
trash.moveToTrash(path);
```

5) 通过网页上直接删除的文件也不会走回收站。

6) 只有在命令行利用 hadoop fs -rm 命令删除的文件才会走回收站。

```
[atguigu@hadoop102 hadoop-3.1.3]$ hadoop fs -rm -r /user/atguigu/input
2020-07-14 16:13:42,643 INFO fs.TrashPolicyDefault: Moved:
'hd fs://hadoop102:9820/user/atguigu/input' to trash at:
hd fs://hadoop102:9820/user/atguigu/.Trash/Current/user/atguigu/input
```

7) 恢复回收站数据

```
[atguigu@hadoop102 hadoop-3.1.3]$ hadoop fs -mv
/user/atguigu/.Trash/Current/user/atguigu/input /user/atguigu/input
```

3.2 Hadoop3.x 新特性

3.2.1 多 NN 的 HA 架构

HDFS NameNode 高可用性的初始实现为单个活动 NameNode 和单个备用 NameNode，将 edits 复制到三个 JournalNode。该体系结构能够容忍系统中一个 NN 或一个 JN 的故障。

但是,某些部署需要更高层次的容错能力。Hadoop3.x 允许用户运行多个备用 NameNode。例如,通过配置三个 NameNode 和五个 JournalNode,群集能够容忍两个节点而不是一个节点的故障。

3.2.2 纠删码

HDFS 中的默认 3 副本方案在存储空间和其他资源(例如,网络带宽)中具有 200% 的开销。但是,对于 I/O 活动相对较低暖 and 冷数据集,在正常操作期间很少访问其他块副本,但仍会消耗与第一个副本相同的资源量。

纠删码(Erasure Coding)能够在不到 50% 的数据冗余情况下提供和 3 副本相同的容错能力,因此,使用纠删码作为副本机制的改进是自然而然的。

查看集群支持的纠删码策略: `hdfs ec -listPolicies`

第 4 章 Hadoop HA 高可用

4.1 HA 概述

(1) 所谓 HA (High Availability), 即高可用 (7*24 小时不中断服务)。

(2) 实现高可用最关键的策略是消除单点故障。HA 严格来说应该分成各个组件的 HA 机制: HDFS 的 HA 和 YARN 的 HA。

(3) Hadoop2.0 之前,在 HDFS 集群中 NameNode 存在单点故障(SPOF)。

(4) NameNode 主要在以下两个方面影响 HDFS 集群

- NameNode 机器发生意外,如宕机,集群将无法使用,直到管理员重启
- NameNode 机器需要升级,包括软件、硬件升级,此时集群也将无法使用

HDFS HA 功能通过配置 Active/Standby 两个 NameNodes 实现在集群中对 NameNode 的热备来解决上述问题。如果出现故障,如机器崩溃或机器需要升级维护,这时可通过此种方式将 NameNode 很快的切换到另外一台机器。

4.2 HDFS-HA 工作机制

通过多个 NameNode 消除单点故障

4.2.1 HDFS-HA 工作要点

1) 元数据管理方式需要改变

内存中各自保存一份元数据;

Edits 日志只有 Active 状态的 NameNode 节点可以做写操作;

所有的 NameNode 都可以读取 Edits;

共享的 Edits 放在一个共享存储中管理 (qjournal 和 NFS 两个主流实现);

2) 需要一个状态管理功能模块

实现了一个 zkfailover, 常驻在每一个 namenode 所在的节点, 每一个 zkfailover 负责监控自己所在 NameNode 节点, 利用 zk 进行状态标识, 当需要进行状态切换时, 由 zkfailover 来负责切换, 切换时需要防止 brain split 现象的发生。

3) 必须保证两个 NameNode 之间能够 ssh 无密码登录

4) 隔离 (Fence), 即同一时刻仅仅有一个 NameNode 对外提供服务

4.2.2 HDFS-HA 自动故障转移工作机制

自动故障转移为 HDFS 部署增加了两个新组件: ZooKeeper 和 ZKFailoverController (ZKFC) 进程, 如图 3-20 所示。ZooKeeper 是维护少量协调数据, 通知客户端这些数据的改变和监视客户端故障的高可用服务。HA 的自动故障转移依赖于 ZooKeeper 的以下功能:

1. 故障检测

集群中的每个 NameNode 在 ZooKeeper 中维护了一个会话, 如果机器崩溃, ZooKeeper 中的会话将终止, ZooKeeper 通知另一个 NameNode 需要触发故障转移。

2. 现役 NameNode 选择

ZooKeeper 提供了一个简单的机制用于唯一的选择一个节点为 active 状态。如果目前现役 NameNode 崩溃, 另一个节点可能从 ZooKeeper 获得特殊的排外锁以表明它应该成为现役 NameNode。

ZKFC 是自动故障转移中的另一个新组件, 是 ZooKeeper 的客户端, 也监视和管理 NameNode 的状态。每个运行 NameNode 的主机也运行了一个 ZKFC 进程, ZKFC 负责:

1) 健康监测

ZKFC 使用一个健康检查命令定期地 ping 与之在相同主机的 NameNode, 只要该 NameNode 及时地回复健康状态, ZKFC 认为该节点是健康的。如果该节点崩溃, 冻结或进入不健康状态, 健康监测器标识该节点为非健康的。

2) ZooKeeper 会话管理

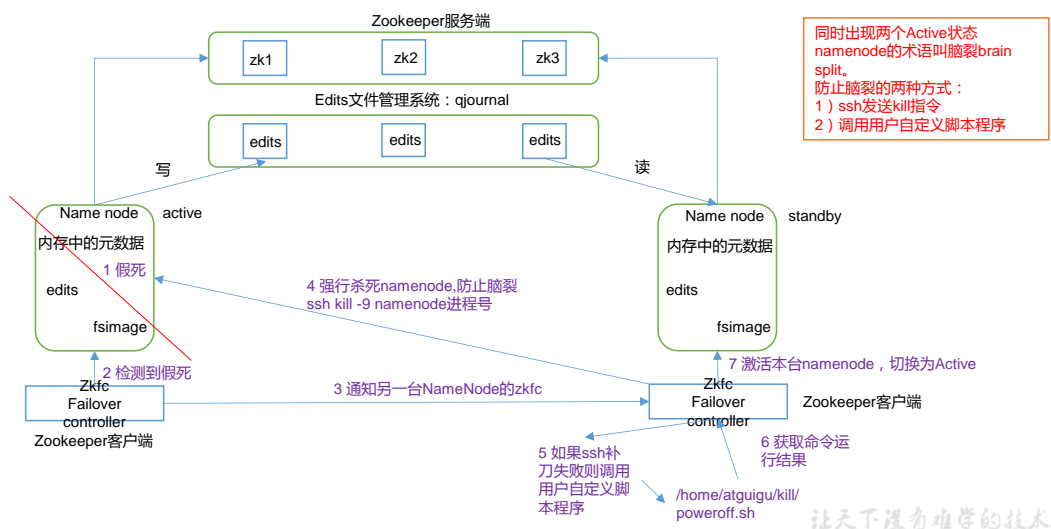
当本地 NameNode 是健康的, ZKFC 保持一个在 ZooKeeper 中打开的会话。如果本地 NameNode 处于 active 状态, ZKFC 也保持一个特殊的 znode 锁, 该锁使用了 ZooKeeper 对

短暂节点的支持，如果会话终止，锁节点将自动删除。

3) 基于 ZooKeeper 的选择

如果本地 NameNode 是健康的，且 ZKFC 发现没有其它的节点当前持有 znode 锁，它将为自已获取该锁。如果成功，则它已经赢得了选择，并负责运行故障转移进程以使它的本地 NameNode 为 Active。

HDFS-HA故障转移机制



4.3 HDFS-HA 集群配置

4.3.1 环境准备

- (1) 修改 IP
- (2) 修改主机名及主机名和 IP 地址的映射
- (3) 关闭防火墙
- (4) ssh 免密登录
- (5) 安装 JDK，配置环境变量等

4.3.2 规划集群

hadoop102	hadoop103	hadoop104
NameNode	NameNode	NameNode
ZKFC	ZKFC	ZKFC
JournalNode	JournalNode	JournalNode
DataNode	DataNode	DataNode
ZK	ZK	ZK
	ResourceManager	

NodeManager	NodeManager	NodeManager
-------------	-------------	-------------

4.3.3 配置 Zookeeper 集群

1) 集群规划

在 hadoop102、hadoop103 和 hadoop104 三个节点上部署 Zookeeper。

2) 解压安装

(1) 解压 Zookeeper 安装包到/opt/module/目录下

```
[atguigu@hadoop102 software]$ tar -zxvf zookeeper-3.5.7.tar.gz -C /opt/module/
```

(2) 在/opt/module/zookeeper-3.5.7/这个目录下创建 zkData

```
[atguigu@hadoop102 zookeeper-3.5.7]$ mkdir -p zkData
```

(3) 重命名/opt/module/zookeeper-3.4.14/conf 这个目录下的 zoo_sample.cfg 为 zoo.cfg

```
[atguigu@hadoop102 conf]$ mv zoo_sample.cfg zoo.cfg
```

3) 配置 zoo.cfg 文件

(1) 具体配置

```
dataDir=/opt/module/zookeeper-3.5.7/zkData
```

增加如下配置

```
#####cluster#####
server.2=hadoop102:2888:3888
server.3=hadoop103:2888:3888
server.4=hadoop104:2888:3888
```

(2) 配置参数解读

Server.A=B:C:D。

A 是一个数字，表示这个是第几号服务器；

B 是这个服务器的 IP 地址；

C 是这个服务器与集群中的 Leader 服务器交换信息的端口；

D 是万一集群中的 Leader 服务器挂了，需要一个端口来重新进行选举，选出一个新的 Leader，而这个端口就是用来执行选举时服务器相互通信的端口。

集群模式下配置一个文件 myid，这个文件在 dataDir 目录下，这个文件里面有一个数据就是 A 的值，Zookeeper 启动时读取此文件，拿到里面的数据与 zoo.cfg 里面的配置信息比较从而判断到底是哪个 server。

4) 集群操作

(1) 在/opt/module/zookeeper-3.5.7/zkData 目录下创建一个 myid 的文件

```
[atguigu@hadoop102 zkData]$ touch myid
```

添加 myid 文件，注意一定要在 linux 里面创建，在 notepad++里面很可能乱码

(2) 编辑 myid 文件

```
[atguigu@hadoop102 zkData]$ vi myid
```

在文件中添加与 server 对应的编号：如 2

(3) 拷贝配置好的 zookeeper 到其他机器上

```
[atguigu@hadoop102 module]$ scp -r zookeeper-3.5.7/
atguigu@hadoop103:/opt/module/
[atguigu@hadoop102 module]$ scp -r zookeeper-3.5.7/
atguigu@hadoop104:/opt/module/
```

并分别修改 myid 文件中内容为 3、4

(4) 分别启动 zookeeper

```
[atguigu@hadoop102 zookeeper-3.5.7]$ bin/zkServer.sh start
[atguigu@hadoop103 zookeeper-3.5.7]$ bin/zkServer.sh start
[atguigu@hadoop104 zookeeper-3.5.7]$ bin/zkServer.sh start
```

(5) 查看状态

```
[atguigu@hadoop104 zookeeper-3.5.7]$ bin/zkServer.sh status
JMX enabled by default
Using config: /opt/module/zookeeper-3.5.7/bin/../conf/zoo.cfg
Mode: follower
[atguigu@hadoop104 zookeeper-3.5.7]$ bin/zkServer.sh status
JMX enabled by default
Using config: /opt/module/zookeeper-3.5.7/bin/../conf/zoo.cfg
Mode: leader
[atguigu@hadoop104 zookeeper-3.5.7]$ bin/zkServer.sh status
JMX enabled by default
Using config: /opt/module/zookeeper-3.5.7/bin/../conf/zoo.cfg
Mode: follower
```

4.3.4 配置 HDFS-HA 集群

1) 官方地址: <http://hadoop.apache.org/>

2) 在 opt 目录下创建一个 ha 文件夹

```
[atguigu@hadoop102 ~]$ cd /opt
[atguigu@hadoop102 opt]$ sudo mkdir ha
[atguigu@hadoop102 opt]$ sudo chown atguigu:atguigu /opt/ha
```

3) 将/opt/module/下的 hadoop-3.1.3 拷贝到/opt/ha 目录下 (记得删除 data 和 log 目录)

```
[atguigu@hadoop102 opt]$ cp -r /opt/module/hadoop-3.1.3 /opt/ha/
```

4) 配置 hadoop-env.sh

```
export JAVA_HOME=/opt/module/jdk1.8.0_212
```

5) 配置 core-site.xml

```
<configuration>
<!-- 把多个 NameNode 的地址组装成一个集群 mycluster -->
  <property>
    <name>fs.defaultFS</name>
    <value>hdfs://mycluster</value>
  </property>
<!-- 指定 hadoop 运行时产生文件的存储目录 -->
  <property>
    <name>hadoop.tmp.dir</name>
```



```
<value>/opt/ha/hadoop-3.1.3/data</value>
</property>
</configuration>
```

6) 配置 hdfs-site.xml

```
<configuration>
<!-- NameNode 数据存储目录 -->
<property>
<name>dfs.namenode.name.dir</name>
<value>file://${hadoop.tmp.dir}/name</value>
</property>
<!-- DataNode 数据存储目录 -->
<property>
<name>dfs.datanode.data.dir</name>
<value>file://${hadoop.tmp.dir}/data</value>
</property>
<!-- JournalNode 数据存储目录 -->
<property>
<name>dfs.journalnode.edits.dir</name>
<value>${hadoop.tmp.dir}/jn</value>
</property>
<!-- 完全分布式集群名称 -->
<property>
<name>dfs.nameservices</name>
<value>mycluster</value>
</property>
<!-- 集群中 NameNode 节点都有哪些 -->
<property>
<name>dfs.ha.namenodes.mycluster</name>
<value>nn1,nn2,nn3</value>
</property>
<!-- NameNode 的 RPC 通信地址 -->
<property>
<name>dfs.namenode.rpc-address.mycluster.nn1</name>
<value>hadoop102:8020</value>
</property>
<property>
<name>dfs.namenode.rpc-address.mycluster.nn2</name>
<value>hadoop103:8020</value>
</property>
<property>
<name>dfs.namenode.rpc-address.mycluster.nn3</name>
<value>hadoop104:8020</value>
</property>
<!-- NameNode 的 http 通信地址 -->
<property>
<name>dfs.namenode.http-address.mycluster.nn1</name>
<value>hadoop102:9870</value>
</property>
<property>
<name>dfs.namenode.http-address.mycluster.nn2</name>
<value>hadoop103:9870</value>
</property>
<property>
<name>dfs.namenode.http-address.mycluster.nn3</name>
<value>hadoop104:9870</value>
</property>
</configuration>
```

```
<!-- 指定 NameNode 元数据在 JournalNode 上的存放位置 -->
<property>
  <name>dfs.namenode.shared.edits.dir</name>
  <value>qjournal://hadoop102:8485;hadoop103:8485;hadoop104:8485/mycluster</value>
</property>
<!-- 访问代理类: client 用于确定哪个 NameNode 为 Active -->
<property>
  <name>dfs.client.failover.proxy.provider.mycluster</name>

<value>org.apache.hadoop.hdfs.server.namenode.ha.ConfiguredFailoverProxyProvider</value>
</property>
<!-- 配置隔离机制, 即同一时刻只能有一台服务器对外响应 -->
<property>
  <name>dfs.ha.fencing.methods</name>
  <value>sshfence</value>
</property>
<!-- 使用隔离机制时需要 ssh 密钥登录-->
<property>
  <name>dfs.ha.fencing.ssh.private-key-files</name>
  <value>/home/atguigu/.ssh/id_rsa</value>
</property>
</configuration>
```

7) 分发配置好的 hadoop 环境到其他节点

4.3.5 启动 HDFS-HA 集群

1) 将 HADOOP_HOME 环境变量更改到 HA 目录

```
[atguigu@hadoop102 ~]$ sudo vim /etc/profile.d/my_env.sh
```

将 HADOOP_HOME 部分改为如下

```
##HADOOP_HOME
export HADOOP_HOME=/opt/ha/hadoop-3.1.3
export PATH=$PATH:$HADOOP_HOME/bin
export PATH=$PATH:$HADOOP_HOME/sbin
```

2) 在各个 JournalNode 节点上, 输入以下命令启动 journalnode 服务

```
[atguigu@hadoop102 ~]$ hdfs --daemon start journalnode
[atguigu@hadoop103 ~]$ hdfs --daemon start journalnode
[atguigu@hadoop104 ~]$ hdfs --daemon start journalnode
```

3) 在[nn1]上, 对其进行格式化, 并启动

```
[atguigu@hadoop102 ~]$ hdfs namenode -format
[atguigu@hadoop102 ~]$ hdfs --daemon start namenode
```

4) 在[nn2]和[nn3]上, 同步 nn1 的元数据信息

```
[atguigu@hadoop103 ~]$ hdfs namenode -bootstrapStandby
[atguigu@hadoop104 ~]$ hdfs namenode -bootstrapStandby
```

5) 启动[nn2]和[nn3]

```
[atguigu@hadoop103 ~]$ hdfs --daemon start namenode
[atguigu@hadoop104 ~]$ hdfs --daemon start namenode
```

6) 查看 web 页面显示

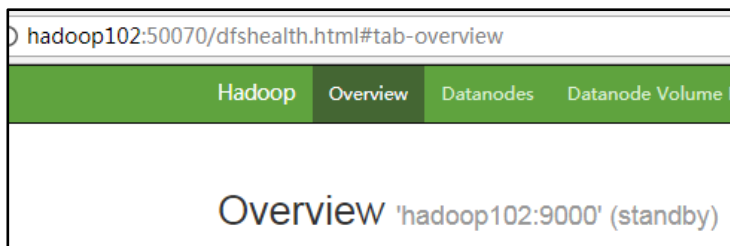


图 hadoop102(standby)

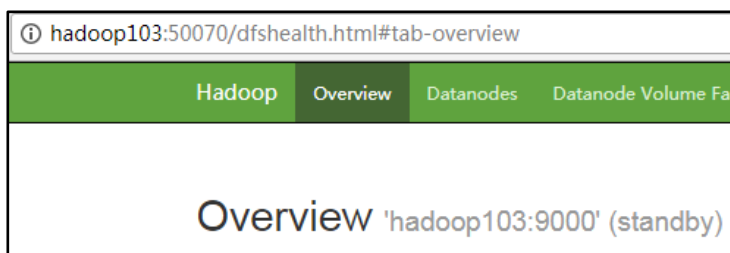


图 hadoop103(standby)

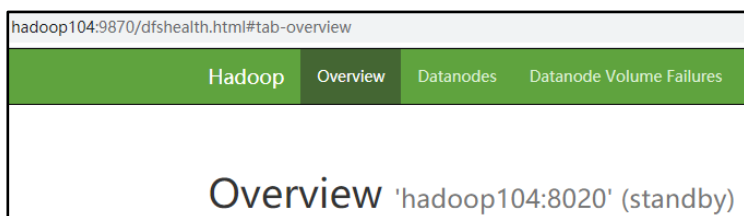


图 hadoop104(standby)

7) 在所有节点上, 启动 datanode

```
[atguigu@hadoop102 ~]$ hdfs --daemon start datanode
[atguigu@hadoop103 ~]$ hdfs --daemon start datanode
[atguigu@hadoop104 ~]$ hdfs --daemon start datanode
```

8) 将[nn1]切换为 Active

```
[atguigu@hadoop102 ~]$ hdfs haadmin -transitionToActive nn1
```

9) 查看是否 Active

```
[atguigu@hadoop102 ~]$ hdfs haadmin -getServiceState nn1
```

4.3.6 配置 HDFS-HA 自动故障转移

1) 具体配置

(1) 在 hdfs-site.xml 中增加

```
<!-- 启用 nn 故障自动转移 -->
<property>
  <name>dfs.ha.automatic-failover.enabled</name>
  <value>true</value>
</property>
```

(2) 在 core-site.xml 文件中增加

```
<!-- 指定 zkfc 要连接的 zkServer 地址 -->
<property>
  <name>ha.zookeeper.quorum</name>
  <value>hadoop102:2181,hadoop103:2181,hadoop104:2181</value>
```

```
</property>
```

(3) 修改后分发配置文件

```
[atguigu@hadoop102 etc]$ pwd
/opt/ha/hadoop-3.1.3/etc
[atguigu@hadoop102 etc]$ xsync hadoop/
```

2) 启动

(1) 关闭所有 HDFS 服务:

```
[atguigu@hadoop102 ~]$ stop-dfs.sh
```

(2) 启动 Zookeeper 集群:

```
[atguigu@hadoop102 ~]$ zkServer.sh start
[atguigu@hadoop103 ~]$ zkServer.sh start
[atguigu@hadoop104 ~]$ zkServer.sh start
```

(3) 启动 Zookeeper 以后, 然后再初始化 HA 在 Zookeeper 中状态:

```
[atguigu@hadoop102 ~]$ hdfs zkfc -formatZK
```

(4) 启动 HDFS 服务:

```
[atguigu@hadoop102 ~]$ start-dfs.sh
```

(5) 可以去 zkCli.sh 客户端查看 Namenode 选举锁节点内容:

```
[zk: localhost:2181(CONNECTED) 7] get -s /hadoop-
ha/mycluster/ActiveStandbyElectorLock
```

```
myclusternn2    hadoop103  <>(<>)
cZxid = 0x10000000b
ctime = Tue Jul 14 17:00:13 CST 2020
mZxid = 0x10000000b
mtime = Tue Jul 14 17:00:13 CST 2020
pZxid = 0x10000000b
cversion = 0
dataVersion = 0
aclVersion = 0
ephemeralOwner = 0x40000da2eb70000
dataLength = 33
numChildren = 0
```

3) 验证

(1) 将 Active NameNode 进程 kill, 查看网页端三台 Namenode 的状态变化

```
[atguigu@hadoop102 ~]$ kill -9 namenode 的进程 id
```

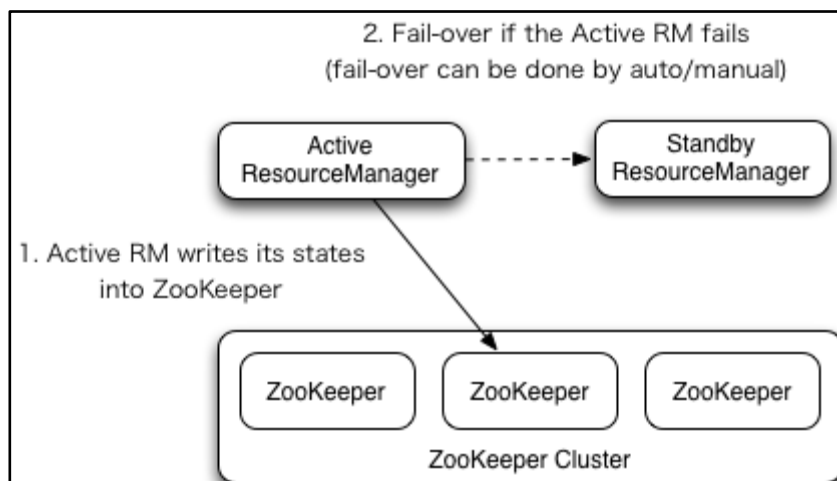
4.4 YARN-HA 配置

4.4.1 YARN-HA 工作机制

1) 官方文档:

<http://hadoop.apache.org/docs/r3.1.3/hadoop-yarn/hadoop-yarn-site/ResourceManagerHA.html>

2) YARN-HA 工作机制



4.4.2 配置 YARN-HA 集群

1) 环境准备

- (1) 修改 IP
- (2) 修改主机名及主机名和 IP 地址的映射
- (3) 关闭防火墙
- (4) ssh 免密登录
- (5) 安装 JDK，配置环境变量等
- (6) 配置 Zookeeper 集群

2) 规划集群

hadoop102	hadoop103	hadoop104
NameNode	NameNode	NameNode
JournalNode	JournalNode	JournalNode
DataNode	DataNode	DataNode
ZK	ZK	ZK
ResourceManager	ResourceManager	
NodeManager	NodeManager	NodeManager

3) 具体配置

(1) yarn-site.xml

```
<configuration>

  <property>
    <name>yarn.nodemanager.aux-services</name>
    <value>mapreduce_shuffle</value>
  </property>

  <!-- 启用 resourcemanager ha -->
  <property>
```

```
<name>yarn.resourcemanager.ha.enabled</name>
<value>true</value>
</property>

<!-- 声明两台 resourcemanager 的地址 -->
<property>
  <name>yarn.resourcemanager.cluster-id</name>
  <value>cluster-yarn1</value>
</property>
<!--指定 resourcemanager 的逻辑列表-->
<property>
  <name>yarn.resourcemanager.ha.rm-ids</name>
  <value>rm1,rm2</value>
</property>
<!-- ===== rm1 的配置 ===== -->
<!-- 指定 rm1 的主机名 -->
<property>
  <name>yarn.resourcemanager.hostname.rm1</name>
  <value>hadoop102</value>
</property>
<!-- 指定 rm1 的 web 端地址 -->
<property>
  <name>yarn.resourcemanager.webapp.address.rm1</name>
  <value>hadoop102:8088</value>
</property>
<!-- 指定 rm1 的内部通信地址 -->
<property>
  <name>yarn.resourcemanager.address.rm1</name>
  <value>hadoop102:8032</value>
</property>
<!-- 指定 AM 向 rm1 申请资源的地址 -->
<property>
  <name>yarn.resourcemanager.scheduler.address.rm1</name>
  <value>hadoop102:8030</value>
</property>
<!-- 指定供 NM 连接的地址 -->
<property>
  <name>yarn.resourcemanager.resource-tracker.address.rm1</name>
  <value>hadoop102:8031</value>
</property>
<!-- ===== rm2 的配置 ===== -->
<!-- 指定 rm2 的主机名 -->
<property>
  <name>yarn.resourcemanager.hostname.rm2</name>
  <value>hadoop103</value>
</property>
<property>
  <name>yarn.resourcemanager.webapp.address.rm2</name>
  <value>hadoop103:8088</value>
</property>
<property>
  <name>yarn.resourcemanager.address.rm2</name>
  <value>hadoop103:8032</value>
</property>
<property>
  <name>yarn.resourcemanager.scheduler.address.rm2</name>
  <value>hadoop103:8030</value>
```

```
</property>
<property>
  <name>yarn.resourcemanager.resource-tracker.address.rm2</name>
  <value>hadoop103:8031</value>
</property>

<!-- 指定 zookeeper 集群的地址 -->
<property>
  <name>yarn.resourcemanager.zk-address</name>
  <value>hadoop102:2181,hadoop103:2181,hadoop104:2181</value>
</property>

<!-- 启用自动恢复 -->
<property>
  <name>yarn.resourcemanager.recovery.enabled</name>
  <value>true</value>
</property>

<!-- 指定 resourcemanager 的状态信息存储在 zookeeper 集群 -->
<property>
  <name>yarn.resourcemanager.store.class</name>
<value>org.apache.hadoop.yarn.server.resourcemanager.recovery.ZKRMStateStore</value>
</property>
<!-- 环境变量的继承 -->
<property>
  <name>yarn.nodemanager.env-whitelist</name>

<value>JAVA_HOME,HADOOP_COMMON_HOME,HADOOP_HDFS_HOME,HADOOP_CONF_DIR,CLASSPATH_PREPEND_DISTCACHE,HADOOP_YARN_HOME,HADOOP_MAPRED_HOME</value>
</property>
</configuration>
```

(2) 同步更新其他节点的配置信息, 分发配置文件

```
[atguigu@hadoop102 etc]$ xsync hadoop/
```

4) 启动 hdfs

```
[atguigu@hadoop102 ~]$ start-dfs.sh
```

5) 启动 YARN

(1) 在 hadoop102 或者 hadoop103 中执行:

```
[atguigu@hadoop102 ~]$ start-yarn.sh
```

(2) 查看服务状态

```
[atguigu@hadoop102 ~]$ yarn rmadmin -getServiceState rm1
```

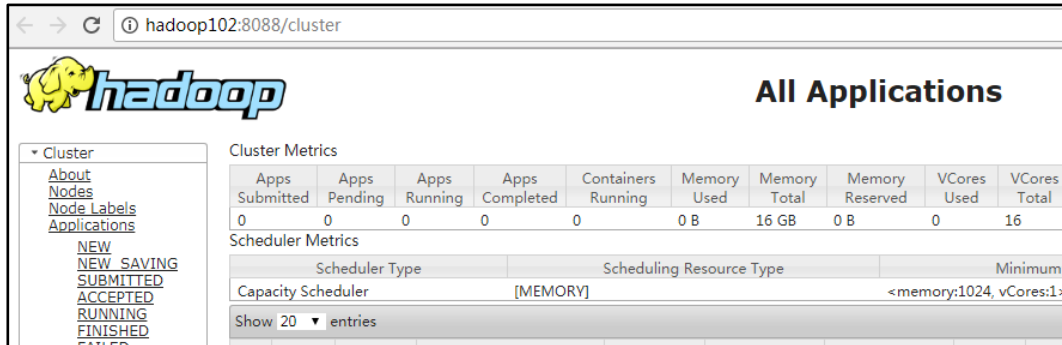
(3) 可以去 zkCli.sh 客户端查看 ResourceManager 选举锁节点内容:

```
[atguigu@hadoop102 ~]$ zkCli.sh
[zk: localhost:2181(CONNECTED) 16] get -s /yarn-leader-election/cluster-yarn1/ActiveStandbyElectorLock

cluster-yarn1rm1
cZxid = 0x100000022
ctime = Tue Jul 14 17:06:44 CST 2020
mZxid = 0x100000022
mtime = Tue Jul 14 17:06:44 CST 2020
```

```
pZxid = 0x100000022
cversion = 0
dataVersion = 0
aclVersion = 0
ephemeralOwner = 0x30000da33080005
dataLength = 20
numChildren = 0
```

(4) web 端查看 hadoop102:8088 和 hadoop103:8088 的 YARN 的状态, 和 NameNode 对比, 查看区别



The screenshot shows the Hadoop YARN web interface for cluster hadoop102:8088. The page title is "All Applications". On the left, there is a navigation menu with options: Cluster, About, Nodes, Node Labels, Applications, NEW, NEW SAVING, SUBMITTED, ACCEPTED, RUNNING, FINISHED, and FAILED. The main content area displays "Cluster Metrics" and "Scheduler Metrics".

Apps Submitted	Apps Pending	Apps Running	Apps Completed	Containers Running	Memory Used	Memory Total	Memory Reserved	VCores Used	VCores Total
0	0	0	0	0	0 B	16 GB	0 B	0	16

Scheduler Metrics

Scheduler Type	Scheduling Resource Type	Minimum
Capacity Scheduler	[MEMORY]	<memory:1024, vCores:1>

Show 20 entries

4.5 HDFS Federation 架构设计

4.5.1 NameNode 架构的局限性

1) Namespace (命名空间) 的限制

由于 NameNode 在内存中存储所有的元数据 (metadata), 因此单个 NameNode 所能存储的对象 (文件+块) 数目受到 NameNode 所在 JVM 的 heap size 的限制。50G 的 heap 能够存储 20 亿 (200million) 个对象, 这 20 亿个对象支持 4000 个 DataNode, 12PB 的存储 (假设文件平均大小为 40MB)。随着数据的飞速增长, 存储的需求也随之增长。单个 DataNode 从 4T 增长到 36T, 集群的尺寸增长到 8000 个 DataNode。存储的需求从 12PB 增长到大于 100PB。

2) 隔离问题

由于 HDFS 仅有一个 NameNode, 无法隔离各个程序, 因此 HDFS 上的一个实验程序就很有可能影响整个 HDFS 上运行的程序。

3) 性能的瓶颈

由于是单个 NameNode 的 HDFS 架构, 因此整个 HDFS 文件系统的吞吐量受限于单个 NameNode 的吞吐量。

4.5.2 HDFS Federation 架构设计

能不能有多个 NameNode

NameNode	NameNode	NameNode
元数据	元数据	元数据
Log	machine	电商数据/话单数据

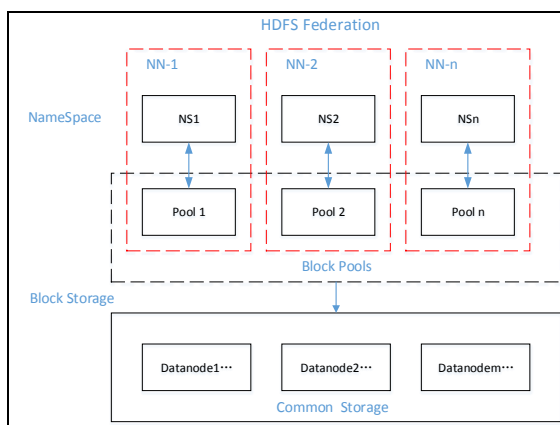


图 HDFS Federation 架构设计

4.5.3 HDFS Federation 应用思考

不同应用可以使用不同 NameNode 进行数据管理图片业务、爬虫业务、日志审计业务。

Hadoop 生态系统中，不同的框架使用不同的 NameNode 进行管理 NameSpace。（隔离性）

