# Lucene

## FILE Format [5.3]

### segment\_x

Int // magic

|--- segments info

String // segment name

Byte[] // segment ID

String // codec name

|--- segment info format

|---

Long // del gen

Int // del count

|--- segment commit info

### .tim

|--- Index Header

Byte // anyAutoPrefixTerms

|--- Posting Header

// header

Vint // BLOCK\_SIZE

|-------------------------------------------------

|--- Posting Data

|-------------------------------------------------

|--- Block

Vint // code code >>> 1 -> entCount

// (code & 1) != 0 -> isLastInFloor

Vint

Vint // number of fields

|--- Terms Data

Vint // field

Vlong // number of terms

Vint // number of bytes for root code

|--- Root code

|-------------------------------------------------

Vlong // sum total of terms freq

Vlong // sum of docs freq

Vint // doc count

Vint // ‘longs size’ unused

|--- min term and max term

|-------------------------------------------------

Eg:

Input: {“abc”,”abe”,”acf”}

Output: { {1,2},{3,4},{4,5}}

**Froniter**

**-> abc**

0 -- | -- 1 -- | -- 2 -- | -- 3 -- | -- ...

S0 S(a) -- S(b) -- S(c)

**-> abd**

0 -- | -- 1 -- | -- 2 -- | -- 3 -- | -- ...

S0 S(a) -- S(b) -- S(c)

|

S(d)

**-> acf**

0 -- | -- 1 -- | -- 2 -- | -- 3 -- | -- ...

S0 S(a) -- S(c) -- S(f)

\

S(b) -- S(e)

|

S(d)

**Finish**

| **10** |

0, 4,3, 2, 101, 27, 2,1, 2, 100, 25, 102, 11, 6,5, 2

|-{3,4}-|-len-|-`e`-|- flag-|-{2,1}-|-len-|-`d`-|-flag-|-`f`-|-flag-|-{5,6}-|-len-|

,99, 22, 10,0,0,0, 98, 0, 97, 6

|-`c`-|-flag-|- **10** |-`b`-|-flag-|-`a`-|-flag

**Flag**

6 = **BIT\_LAST\_ARC | BIT\_TARGET\_NEXT**

22 =

### .tip

|--- Index Header

|------------------------------------------------

Vlong // index start FP

### .fdx

Vint // Version , without zigzag

|--- block

|--- header

**CodecHeader**

Bytes // id

Byte // suffix length

Bytes // suffix

|---

...

Vlong // max pointer

...

|--- write block

Vint // write 0 , end marker

Vlong // max pointer

|--- footer

### .fdt

|--- header

**CodecHeader**

Bytes // id

Byte // suffix length

Bytes // suffix

...

Vint // chunk size

Vint // packed ints version

...

|--- block

Vint // base doc docBase+numBufferedDocs = DocID

Vint // numBufferedDocs << 1 | sliceBit // sliced

// save numStoredFields

/\*

chunk docs = token >>> 1

\*/

// chunk docs == 1

Vint

Vint

// or more than 1

Vint // bits of per stored fields

Vlong //

Vlong // field number << TYPE\_BITS | type

Vint //

Bytes //

Z.. // zigzag

|--- max pointer // avoid random write

Vlong // number of chunks

Vlong // number of dirty chunks

|--- footer [16]

## ../blocktree/..

### FieldReader

### BlockTreeTermsReader

- fields [TreeMap<String, FieldReader>]

- anyAutoPrefixTerms [boolean]

### BlockTreeTermsWriter

### SegmentTermsEnum

- SegmentTermsEnumFrame[] stack

- SegmentTermsEnumFrame currentFrame

- final SegmentTermsEnumFrame staticFrame

+ seekExact(BytesRef, TermState)

+ seekExact(BytesRef)

+ prefixMatches(BytesRef)

+ pushFrame()

+ getFrame (int) // ord

+ seekCeil()

### SegmentTermsEnumFrame

- ord [int]

- BlockTermState // lucene50 IntBlockTermState

- entCount [int] // Number of entries (term of sub-block)

// in this block

- isLastInFloor [boolean] // (code & 1) != 0

// a floor block, or last sub-block of a // floor block

- versionAutoPrefix [boolean]

+

### IntersectTermsEnum

- boolean useAutoPrefixTerm

- IntersectTermsEnumFrame[5]

- FST.Arc[5]

- RunAutoMaton

- FieldReader

+ popPushNext()

### IntersectTermsEnumFrame

- final int ord

- long fp

- long fpOrig

- long fpEnd

- long lastSubFP

- int state

- int lastState

- entCount

- nextEnt

- boolean isLeafBlock // true all entries are terms

- ByteArrayDataInput suffixesReader

- ByteArrayDataInput statsReader

- ByteArrayDataInput floorDataReader

+ nextLeaf()

... | .. vint .. |

suffix

## ../compressing/..

### CompressionMode

**{**

**FAST**

**HIGH\_COMPRESSION**

**FAST\_DECOMPRESSION**

**DUMMY**

**}**

### CompressingStoredFieldsFormat

### StoredFieldsWriter

+ merge(MergeState)

**MergeVisitor [IC]**

## ../codec/..

### TermState

- copyFrom (TermState)

#### OrdTermState

- ord [long] // Term ordinal, its position in the full of // sorted terms

##### BlockTermState

- totalTermFreq [long]

- blockFilePointer [long]

- termBlockOrd [int] // the term’s ord in the current block

### StoredFieldFormat

### ../lucene50/..

### ../lucene53/..

#### Lucene50StoredFieldFormat

**Mode**

**{**

**// chunk size , max docs per chunk, block size**

**BEST\_SPEED**

**1<<14, 128, 1024**

**BEST\_COMPRESSION**

**61440, 512, 1024**

**}**

// chunk size || max docs per chunk flush

### CompressingStoredFieldsWriter

- endOffsets [int[]] //

- numStoredFields [int[]] //

- numBufferedDocs [int]

+ writeTLong // T for timestamp

+ finishDocument()

> triggerFlush()

> flush()

### CompressingStoredFieldsIndexWriter

- blockSize [final int]

- totalDocs

- startPointerDeltas [long[blockSize]]

### CompressingStoredFieldsReader

- BlockState

- CompressingStoredFieldsIndexReader // .fdx

- InputIndex // field intputstream

- int[] offset

- int[] numStoredFields

+ visitDocument(int, StoredFieldVisitor)

**SerializedDocument [IC]**

- DataInput

- length // the number of bytes on which the document is encoded

- numStoredFields

+ block(int docID) // binary search

+ relativeChunk(int block, int relativeDoc) // binary search

+ doReset(int docID)

**BlockState [IC]**

- docBase [int]

## Utils

### ../automaton/..

#### Automaton

#### CompiledAutomaton

### IntsRefBuilder

### ../fst/..

#### FST

**INPUT\_TYPE** {BYTE1, BYTE2, BYTE4} //

**Flags {**

**BIT\_FINAL\_ARC** [1 << 0]

**BIT\_LAST\_ARC** [1 << 1]

**BIT\_TARGET\_NEXT** [1 << 2]

}

+ writeLabel(DataOutput, int v)

// INPUT\_TYPE 决定写入v的长度

- long startNode

- Outputs<T>

- T emptyOutput

- flags [byte]

--- ?? ---

#### Arc

- int label //

- Node

- boolean isFinal

- T output

- T nextFinalOutput

### BytesStore

+ reverse

#### Builder

| --

- IntsRefBuilder lastInput;

- UnCompiledNode[] frontier

+ freezeTail

doPrune

doCompile

lastInput.length() > prefixLenPlus1

+ compileNode

#### CompileNode

#### UnCompiledNode

### NodeHash

### PACKED

#### BulkOperation

+ of(PackedInts.Format,int) // bits of per values

+ computeIterations(int, int) // valueCount, ramBudget

#### DirectWriter

#### PackedLongValues

+ pack()

#### PackedInts

**Format**

**{**

**PACKED**

**PACKED\_SINGLE\_BLOCK**

**}**

+ fastestFormatAndBits

+ getMutable()

PackedInts.Format

[

PACKED\_SINGLE\_BLOCK

PACKED

]

+ fastestFormatAndBits

+ byteBlockCount()

// The minimum number of byte blocks to encode in a single iteration ...

+ byteValueCount()

// The number of values that can be stored in **byte block**

##### Mutable

##### MutableImpl

##### PackedReaderIterator

(

int // mem, how much memory the iterator is allowed to use to // **read-ahead**

// use compute iterations

)

- PackedInts.Format //

- int // packed ints version

- BulkOperation

- int bitsPackedValue //

- int valueCount //

- int iterations //

- byte[] // next blocks, [iterations \* byteBlockCount]

- LongsRef // next values

- int postion

- next()

- next(int) //

+ iterations(int) // mem

iterations =

## Codec

### FieldsProducer

## Index

### BaseCompositeReader

### IndexWriter

-

- ReadPool

- rollbackSegments [List<SegmentCommitInfo>]

### DefaultIndexingChain

+ processDocument()

> startStoredFields()

> processField()

+ flush(SegmentWriteState)

### SegmentInfos

+ readCommit(Directory, String) // segmentFileName

+ readCommit [**static]**

### SegmentReader

### STORE

#### IOContext

#### BufferedIndexInput

+ bufferSize(IOContext)

case MERGE: 4096

default: 1024

#### NIOFSDirectory

**NIOFSInputIndex [IC]**

### MERGE

#### MergeState

#### SegmentMerger

- FieldInfosBuilder

+ mergeTerms(SegmentWriteState)

+ merge()

-> mergeFieldInfos

-> mergeFields

#### TermState

BlockTermState

### MergePolicy

#### TieredMergePolicy

### MergeScheduler

#### ConcurrentMergeScheduler

- maxThreadCount // spins ? 1 : min(4, core/2)

- maxMergeCount // spins ? 6 : maxThreadCount+5

+ updateIOThrottle

+ updateMergeThreads()

+ merge(IndexWriter, MergeTrigger, )

+ doMerge(IndexWriter)

IndexWriter.merge

### IndexDeletionPolicy

#### KeepOnlyLastCommitDeletionPolicy

#### IndexFileDeleter

#### IntexCommit

### FlushPolicy

+ onDelete(DocumentsWriteFlushControl, ThreadState)

#### FlushByRamOrCountsPolicy

+ onInsert(DocumentsWriterFlushControl, ThreadState)

If (flushOnDocCount) ...

else if (flushOnRAM)

+ onDelete() [**override**]

> flushOnDeleteTerm() // indexWriterConfig.getMaxBufferedDelete...

//

#### DocumentsWriterFlushControl

- boolean fullflush

#### TermContext

#### StoredFieldsWriter

## Search

### IndexSearcher

+ createNormalizedWeight()

### LeafCollector

#### FilterLeafCollector

### Scorer

#### BooleanScorer

Bucket[2084]

+ advance()

### MultiTermQueryConstantScoreWrapper

+ bool collectTerms()

+ createWeight() []

-> ConstantScoreWeight

+ collectTerms()

+ rewrite()

-> WeightOrDocIdSet

### Query

#### AutomatonQuery

#### MultiTermQuery

- RewriteMethod = CONSTANT\_SCORE\_REWRITE {

+ rewrite()

-> MultiTermQueryConstantScoreWrapper

}

-

## POSTINGS

### PostingsFormat

#### Lucene50PostingsFormat

## DocValues

### DocValuesProducer

+ getDocsWithField(FieldInfo)

## Writer

### Directory

MMapDirectory

### IndexWriter

## Reader

### SegmentReader

+ initDocValuesProducer

SortedDocValues

SortedDocValuesField

### DocValuesFormat

Lucene50DocValuesFormat

.dvd

.dvm // metadata

### DocValuesProducer

Lucene50DocValuesProducer

### DocValuesConsumer

### Lucene50DocValuesConsumer

- Map<String, NumericEntry>

  + readNumericEntry

|------ VInt ------|

|-- read format --|

 Index

 PostingsFormat

 Lucene50PostingsWriter

   Search

 IndexSearcher

  Query

  NumericRangeQuery

  + getTermsEnum

NumericUtils.splitIntRange

 TermRangeQuerystatic Automaton toAutomaton()

 Scorer

   BooleanScorer

  + scoreDocument

  MinShouldMatchSumScorer

  DisiPriorityQueue head [size: scorers.size() - minShouldMatch

DisiWrapper[] tail [length: minShouldMatch]// save the maximum cost

    DisiWrapper lead

  DisiWrapper

 DocIdSetIterator \*\*

+ cost()

  MultiTermQuery

  合并多个term倒排表

  ConstantScoreScorer

 RewriteMethod

 TwoPhaseIterator

  MultiTermQueryConstantScoreWrapper

   CompiledAutomaton

    Utils

 BitSet

+ 1]

// is the minimum heap

DocIdSetBuilder

- buffer[] = int[0]

- bufferSize = 0;

- int threshold = maxDoc >>> 7;

+ add()

 Algorithm

   TimSort

  Hadoop

 Yarn

 ResourceManager

 在YARN中,ResourceManager负责集群中所有资源的统一管理和分配,它接收来自各个节

(NodeManager)的资源汇报信息,并把这些信息按照一定的策略分配给各个应用程序(实际上

ApplicationManager)

 ￼

             ResourceManager主要由以下几个部分组成:

用户交互

YARN分别针对普通用户,管理员和Web提供了三种对外服务,分别对

ClientRMService、AdminService和WebApp: ClientRMService

ClientRMService是为普通用户提供的服务,它会处理来自客户端各种RPC请求,比如提交应用程序、

止应用程序,获取应用程序运行状态等。

AdminService

YARN为管理员提供了一套独立的服务接口,以防止大量的普通用户请求使管理员发送的管理命令

死,管理员可通过这些接口管理集群,比如动态更新节点列表,更新ACL列表,更新队列信息等。

WebApp

为了更加友好地展示集群资源使用情况和应用程序运行状态等信息,YARN对外提供了一个Web 界面

这一部分是YARN仿照haml(http://haml.info/)开发的一个轻量级嵌入式Web框架。具体讨论见

https://issues.apache.org/jira/browse/MAPREDUCE-2399

 ApplicationMaster

 ApplicationMasterLauncher

- Queue<Runnable> masterEvents

- LauncherThread

  Event

LAUNCH

add AMLauncher in Queue

 CLEANUP

 AMLivelinessMonitor

ApplicationMasterService

ApplicationMaster

    YARN中内存资源的调度和隔离

基于以上考虑,YARN允许用户配置每个节点上可用的物理内存资源,注意,这里是“可用的”,因为

个节点上的内存会被若干个服务共享,比如一部分给YARN,一部分给HDFS,一部分给HBase等,YAR

配置的只是自己可以使用的,配置参数如下:

yarn.nodemanager.vmem-pmem-ratio

任务每使用1MB物理内存,最多可使用虚拟内存量,默认是2.1。

yarn.nodemanager.pmem-check-enabled

是否启动一个线程检查每个任务正使用的物理内存量,如果任务超出分配值,则直接将其杀掉,默认

true。

yarn.nodemanager.vmem-check-enabled

是否启动一个线程检查每个任务正使用的虚拟内存量,如果任务超出分配值,则直接将其杀掉,默认true。

yarn.scheduler.minimum-allocation-mb

单个任务可申请的最少物理内存量,默认是1024(MB),如果一个任务申请的物理内存量少于该值,

该对应的值改为这个数。

yarn.scheduler.maximum-allocation-mb

单个任务可申请的最多物理内存量,默认是8192(MB)。

默认情况下,YARN采用了线程监控的方法判断任务是否超量使用内存,一旦发现超量,则直接将

杀死。由于Cgroups对内存的控制缺乏灵活性(即任务任何时刻不能超过内存上限,如果超过,则直接

其杀死或者报OOM),而Java进程在创建瞬间内存将翻倍,之后骤降到正常值,这种情况下,采用线

监控的方式更加灵活(当发现进程树内存瞬间翻倍超过设定值时,可认为是正常现象,不会将任务

死),因此YARN未提供Cgroups内存隔离机制。

yarn.resourcemanager.am.max-attempts

一个全局的AM重试次数的限制,yarn提交应用时,还可以为单独一个应用设置最大重试次数

当attempt失败时,如果设置keepContainersAcrossAppAttempts了,resource manager会决定

个attempt的container是否仍然保留着。

       YARN中CPU资源的调度和隔离

  在YARN中,CPU资源的组织方式仍在探索中,目前(2.2.0版本)只是一个初步的,非常粗粒度的实

方式,更细粒度的CPU划分方式已经提出来了,正在完善和实现中。

目前的CPU被划分成虚拟CPU(CPU virtual Core),这里的虚拟CPU是YARN自己引入的概念,初

是,考虑到不同节点的CPU性能可能不同,每个CPU具有的计算能力也是不一样的,比如某个物理CP

的计算能力可能是另外一个物理CPU的2倍,这时候,你可以通过为第一个物理CPU多配置几个虚

CPU弥补这种差异。用户提交作业时,可以指定每个任务需要的虚拟CPU个数。在YARN中,CPU相关

置参数如下:

yarn.nodemanager.resource.cpu-vcores

表示该节点上YARN可使用的虚拟CPU个数,默认是8,注意,目前推荐将该值设值为与物理CPU核数

目相同。如果你的节点CPU核数不够8个,则需要调减小这个值,而YARN不会智能的探测节点的物

CPU总数。

yarn.scheduler.minimum-allocation-vcores

单个任务可申请的最小虚拟CPU个数,默认是1,如果一个任务申请的CPU个数少于该数,则该对应的

改为这个数。

yarn.scheduler.maximum-allocation-vcores

单个任务可申请的最多虚拟CPU个数,默认是32。

默认情况下,YARN是不会对CPU资源进行调度的,你需要配置相应的资源调度器让你支持,具体可参

我的这两篇文章:

(1)Hadoop YARN配置参数剖析(4)—Fair Scheduler相关参数

(2)Hadoop YARN配置参数剖析(5)—Capacity Scheduler相关参数

默认情况下,NodeManager不会对CPU资源进行任何隔离,你可以通过启用Cgroups让你支持CP隔离。

由于CPU资源的独特性,目前这种CPU分配方式仍然是粗粒度的。举个例子,很多任务可能是IO密集

的,消耗的CPU资源非常少,如果此时你为它分配一个CPU,则是一种严重浪费,你完全可以让他与其

几个任务公用一个CPU,也就是说,我们需要支持更粒度的CPU表达方式。

借鉴亚马逊EC2中CPU资源的划分方式,即提出了CPU最小单位为EC2 Compute Unit(ECU),一

ECU代表相当于1.0-1.2 GHz 2007 Opteron or 2007 Xeon处理器的处理能力。YARN提出了CPU最小

位YARN Compute Unit(YCU),目前这个数是一个整数,默认是720,由参

yarn.nodemanager.resource.cpu-ycus-per-core设置,表示一个CPU core具备的计算能力(该featur

在2可能增加到2.3.0版本中),这样,用户提交作业时,直接指定需要的YCU即可,比如指定值为360,

示用1/2个CPU core,实际表现为,只使用一个CPU core的1/2计算时间。注意,在操作系统层,CPU资

是按照时间片分配的,你可以说,一个进程使用1/3的CPU时间片,或者1/5的时间片。对于CPU资源划

和调度的探讨,可参考以下几个链接:

https://issues.apache.org/jira/browse/YARN-1089

https://issues.apache.org/jira/browse/YARN-1024

Hadoop 新特性、改进、优化和Bug分析系列5:YARN-3

    Timeline Server

1.背景介绍:

在hadoop2.4版本之前对任务执行的监控只开发了针对MR的Job History Server,它可以提供给用

用户查询已经运行完成的作业的信息,但是后来,随着在YARN上面集成的越来越多的计算框架,比

spark、Tez,也有必要为基于这些计算引擎的技术开发相应的作业任务监控工具,所以hadoop的开

人员就考虑开发一款更加通用的Job History Server,即YARN Timeline Server。

 2.Introduction

<hadoop2.7.1>

以一种通用的方式向YARN Timeline Server上注册应用程序的当前和历史状态,便于存储和检索

它主要有两大职责:

(1)持久化应用程序的具体信息

收集并检索应用程序或者框架的具体信息。例如,hadoop MR框架里面的与分片线关系的信息,

如map tasks、reduce tasks、counters等。应用程序开发者可以在App Master端或者应用程序需

containers中通过TimelineClient将这些信息发送给Timeline Server。

这些信息都可以通过REST APIs在具体App或者执行框架的UI界面查询到。

(2)持久化已完成的应用程序的Generic information

关于这一点,在Application history server中,显然只支持MR框架的job。Generic information包

了像\*queue-name,\*用户信息等用户程序级别的数据,还有设置在ApplicationSubmissionContext

的信息,用于运行应用程序的application-attempts 列表,关于每个application-attempt的

息,container列表以及运行在每个 application-attempt \*下的每个container的信息。

 3.Timeline Structure￼

       (1)Timeline Domain

Timeline Domain为Timeline Server提供了一个命令空间,使得用户可以搜集多个节点,将它们

其他用户和应用程序隔离开来。Timeline server security就定义在这一层。

一个Domain首先是用于存储用户的信息、读写ACL信息、创建和修改时间戳。每个Domain以一个

一的ID在整个YARN集群中标识。

(2)Timeline Entity

一个Timeline Entity(即Timeline实体)包含一个概念实体的元信息以及它的相关的events.一个

体可以是一个application,一个 application attempt,一个container卓尔其他任何的应用自定义

object。

它还包含多个Primary filters用于作为timeline store中多个实体的索引。其他的数据可以以非

引的方式存储。每个实体都通过一个EntityId和EntityType唯一的确定。

(3)Timeline Events

Timeline Events用于描述一个与某个具体Application的timeline实体相关的event。用户也可以

意定义一个event方法,比如启动一个应用程序,获取分配的container、操作失败或者其他的与用户

集群操作相关的失败信息等等。

   Spark

  RDD

全称为Resilient Distributed Datasets,是一个容错的、并行的数据结构,可以让用户显式地将数

存储到磁盘和内存中,并能控制数据的分区。同时,RDD还提供了一组丰富的操作来操作这些数据。

这些操作中,诸如map、flatMap、filter等转换操作实现了monad模式,很好地契合了Scala的集合操作

除此之外,RDD还提供了诸如join、groupBy、reduceByKey等更为方便的操作(注意,reduceByKey

action,而非transformation),以支持常见的数据运算。

----------------------------------------------------------------------------------------------------------------------

在hadoop中一个独立的计算,例如在一个迭代过程中,除可复制的文件系统(HDFS)外没有提供其

存储的概念,这就导致在网络上进行数据复制而增加了大量的消耗,而对于两个的MapReduce作业

间数据共享只有一个办法,就是将其写到一个稳定的外部存储系统,如分布式文件系统。这会引入数

备份、磁盘I/O以及序列化,这些都会引起大量的开销,从而占据大部分的应用执行时间。所以我们发

如果在计算过程中如能共享数据,那将会降低集群开销同时还能减少任务执行时间。而 spark中的RDDs让用户可以直接控制数据的共享。RDD具有可容错和并行数据结构特征,可以

定数据存储到硬盘还是内存、控制数据的分区方法并在数据集上进行种类丰富的操作。

目前提出的基于集群的内存存储抽象,比如分布式共享内存(Distributed Shared Memory|DSM)

键-值存储(Key-Value|Nosql),数据库(RDBMS)等提供了一个对内部状态基于细粒度更新的接口(例如

表格里面的单元)。而这样设计,提供容错性的方法:在主机之间复制数据,或者对各主机的更新情况

日志记录。但这两种方法对于数据密集型的任务来说代价很高,因为它们需要在带宽远低于内存的

群网络间拷贝大量的数据,同时还将产生大量的存储开销。但RDD提供一种基于粗粒度变换(如

map,filter,join)的接口,该接口会将相同的操作应用到多个数据集上。这使得他们可以通过记录用来

建数据集的变换(lineage),而不需存储真正的数据,进而达到高效的容错性。当一个RDD的某个分区

失的时候,RDD记录有足够的信息记录其如何通过其他的RDD进行计算,且只需重新计算该分区。

此,丢失的数据可以被很快的恢复,而不需要昂贵的复制代价。

-----------------------------------------------------------------------------------------------------------------------

  通常来讲,针对数据处理有几种常见模型,包括:Iterative Algorithms,Relational

Queries,MapReduce,Stream Processing。例如Hadoop MapReduce采用了MapReduces

型,Storm则采用了Stream Processing模型。RDD混合了这四种模型,使得Spark可以应用于各种大

据处理场景。

  RDD作为数据结构,本质上是一个只读的分区记录集合。一个RDD可以包含多个分区,每个分区就

一个dataset片段。RDD可以相互依赖。如果RDD的每个分区最多只能被一个Child RDD的一个分区

用,则称之为narrow dependency;若多个Child RDD分区都可以依赖,则称之为wide dependency。

同的操作依据其特性,可能会产生不同的依赖。例如map操作会产生narrow dependency,而join操

则产生wide dependency。

 Spark之所以将依赖分为narrow与wide,基于两点原因。

 首先,narrow dependencies可以支持在同一个cluster node上以管道形式执行多条命令,例如在执

了map后,紧接着执行filter。相反,wide dependencies需要所有的父分区都是可用的,可能还需要

用类似MapReduce之类的操作进行跨节点传递。

 其次,则是从失败恢复的角度考虑。narrow dependencies的失败恢复更有效,因为它只需要重新计

丢失的parent partition即可,而且可以并行地在不同节点进行重计算。而wide dependencies牵涉

RDD各级的多个Parent Partitions。下图说明了narrow dependencies与wide dependencies之间的

别:

  Linux

 Hwaddr 和 MacAddr

 noatime, nodiratime, discard

  expect 源码

### Lucene 4.**0** Feature

### Lucene 5.0 Feature

#### 5.3.1

LUCENE-6548: Some optimizations for BlockTree's intersect with very finite automata

LUCENE-6617: Reduce heap usage for small FSTs

LUCENE-6668: Optimized storage for sorted set and sorted numeric doc values in the case that there are few unique sets of values.

#### 5.5.4

##### BUGS

......

LUCENE-7440: Document id skipping (PostingsEnum.advance) could throw an ArrayIndexOutOfBoundsException exception on large index segments (>1.8B docs) with large skips.

LUCENE-7570: IndexWriter may deadlock if a commit is running while there are too many merges running and one of the merges hits a tragic exception

### Lucene 6.0 Feature

Optimizations

LUCENE-6891: Use prefix coding when writing points in each leaf block in the default codec, to reduce the index size

LUCENE-6901: Optimize points indexing: use faster IntroSorter instead of InPlaceMergeSorter, and specialize 1D merging to merge sort the already sorted segments instead of re-indexing

### Lucene 6.2.0 Feature

Optimizations

LUCENE-7330, LUCENE-7339: Speed up conjunction queries.

LUCENE-7351: Doc id compression for points.

# Hadoop [2.7.4]

## HDFS

**dfshealth.html {**

**LastContact // 表明DataNode有多少秒的时间未向NameNode发送心跳**

**}**

**COMMANDS {**

**# snapshot**

**hdfs lsSnapshottableDir**

**hdfs dfs -deleteSnaphost**

**hdfs storagepolicies**

**}**

**CONFIG{**

**dfs.datanode.balance.bandwidthPerSec**

**[指定DataNode用于balancer的带宽, 1MB(default)]**

**dfs.permissions.enabled**

**[关闭权限检查]**

**dfs.datanode.socket.write.timeout**

**[(8\*60\*1000)]**

**dfs.client.socket-timeout**

**[60\*1000]**

**dfs.datanode.socket.reuse.keepalive**

**[peer read timeout (4\*1000)]**

**}**

### Block

#### BlockManager

BlockMap

CorruptReplicasMap

InvalidateBlocks

+ addToInvalidate(Block)

// Populating ReplQueue

#### BlocksMap

#### CorruptReplicasMap

Chunk

Packet

http://hadoop.apache.org/docs/current/hadoop-project-dist/hadoop-hdfs/ArchivalStorage.html

### Archival Storage

**Policies :**

* **Hot** - for both storage and compute. The data that is popular and still being used for processing will stay in this policy. When a block is hot, all replicas are stored in DISK.
* **Cold** - only for storage with limited compute. The data that is no longer being used, or data that needs to be archived is moved from hot storage to cold storage. When a block is cold, all replicas are stored in ARCHIVE.
* **Warm** - partially hot and partially cold. When a block is warm, some of its replicas are stored in DISK and the remaining replicas are stored in ARCHIVE.
* **All\_SSD** - for storing all replicas in SSD.
* **One\_SSD** - for storing one of the replicas in SSD. The remaining replicas are stored in DISK.
* **Lazy\_Persist** - for writing blocks with single replica in memory. The replica is first written in RAM\_DISK and then it is lazily persisted in DISK.

**Tools {**

mover

**}**

### HA

**CONFIG {**

**/-------------- hdfs-site.xml ----------------/**

**dfs.ha.fencing.methods**

**[sshfence|sshfence(username:port)|shell(../.sh)]**

**dfs.ha.fencing.ssh.private-key-files**

**[../.ssh/id\_rsa]**

**dfs.ha.fencing.ssh.connect-timeout**

**[]**

**}**

方案：

HDFS High Availability Using the Quorum Journal Manager   
High Availability With NFS   
HDFS Federation

HAState

早期的hadoop版本，NN是HDFS集群的单点故障点，每一个集群只有一个NN,如果这个机器或进程不可用，整个集群就无法使用。为了解决这个问题，出现了一堆针对HDFS HA的解决方案（如：Linux HA, VMware FT, shared NAS+NFS, BookKeeper, QJM/Quorum Journal Manager, BackupNode等）; 在HA具体实现方法不同的情况下，HA框架的流程是一致的, 不一致的就是如何存储和管理日志。在Active NN和Standby NN之间要有个共享的存储日志的地方，Active NN把EditLog写到这个共享的存储日志的地方，Standby NN去读取日志然后执行，这样Active和Standby NN内存中的HDFS元数据保持着同步。一旦发生主从切换Standby NN可以尽快接管Active NN的工作;

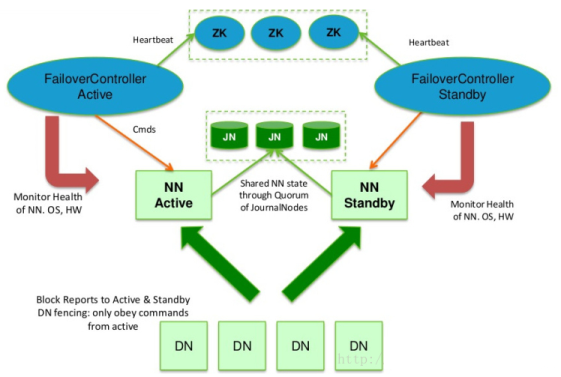
在 HDP2.4安装(五)：集群及组件安装 章节使用ambari创建cluster时，默认并未启用 hdfs ha, 可以通过 ambari 管理界面进行安装

#### SPOF（single point of failure）方案回顾

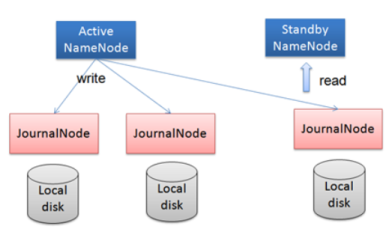
* Secondary NameNode：它不是HA，它只是阶段性的合并edits和fsimage，以缩短集群启动的时间。当NN失效的时候，Secondary NN并无法立刻提供服务，Secondary NN甚至无法保证数据完整性：如果NN数据丢失的话，在上一次合并后的文件系统的改动会丢失
* Backup NameNode (HADOOP-4539)：它在内存中复制了NN的当前状态，算是Warm Standby，可也就仅限于此，并没有failover等.它同样是阶段性的做checkpoint,也无法保证数据完整性.
* 手动把name.dir指向NFS（Network File System），这是安全的Cold Standby，可以保证元数据不丢失，但集群的恢复则完全靠手动
* Facebook AvatarNode：Facebook有强大的运维做后盾，所以Avatarnode只是Hot Standby，并没有自动切换，当主NN失效的时候，需要管理员确认，然后手动把对外提供服务的虚拟IP映射到Standby NN，这样做的好处是确保不会发生脑裂的场景。其某些设计思想和Hadoop 2.0里的HA非常相似，从时间上来看，Hadoop 2.0应该是借鉴了Facebook的做法

#### hadoop2.x ha 原理:

* hadoop2.x之后，Clouera提出了QJM/Qurom Journal Manager，这是一个基于Paxos算法实现的HDFS HA方案，它给出了一种较好的解决思路和方案:



* 基本原理就是用2N+1台 JN 存储EditLog，每次写数据操作有大多数（>=N+1）返回成功时即认为该次写成功，数据不会丢失了。当然这个算法所能容忍的是最多有N台机器挂掉，如果多于N台挂掉，这个算法就失效了。这个原理是基于Paxos算法
* 在HA架构里面SecondaryNameNode这个冷备角色已经不存在了，为了保持standby NN时时的与主Active NN的元数据保持一致，他们之间交互通过一系列守护的轻量级进程JournalNode
* 任何修改操作在 Active NN上执行时，JN进程同时也会记录修改log到至少半数以上的JN中，这时 Standby NN 监测到JN 里面的同步log发生变化了会读取 JN 里面的修改log，然后同步到自己的的目录镜像树里面，如下图：



* 当发生故障时，Active的 NN 挂掉后，Standby NN 会在它成为Active NN 前，读取所有的JN里面的修改日志，这样就能高可靠的保证与挂掉的NN的目录镜像树一致，然后无缝的接替它的职责，维护来自客户端请求，从而达到一个高可用的目的
* QJM方式来实现HA的主要优势：

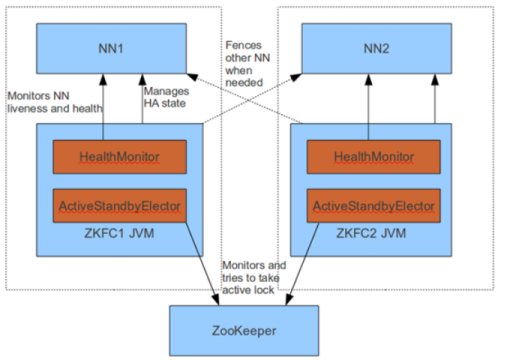
1. 不需要配置额外的高共享存储，降低了复杂度和维护成本
2. 消除spof
3. 系统鲁棒性(Robust:健壮)的程度是可配置
4. JN不会因为其中一台的延迟而影响整体的延迟，而且也不会因为JN的数量增多而影响性能（因为NN向JN发送日志是并行的）

#### hadoop2.x ha 详述

* datanode的fencing: 确保只有一个NN能命令DN.HDFS-1972中详细描述了DN如何实现fencing

1. 每个NN改变状态的时候，向DN发送自己的状态和一个序列号
2. DN在运行过程中维护此序列号，当failover时，新的NN在返回DN心跳时会返回自己的active状态和一个更大的序列号。DN接收到这个返回则认为该NN为新的active
3. 如果这时原来的active NN恢复，返回给DN的心跳信息包含active状态和原来的序列号，这时DN就会拒绝这个NN的命令

* 客户端fencing：确保只有一个NN能响应客户端请求，让访问standby nn的客户端直接失败。在RPC层封装了一层，通过FailoverProxyProvider以重试的方式连接NN。通过若干次连接一个NN失败后尝试连接新的NN，对客户端的影响是重试的时候增加一定的延迟。客户端可以设置重试此时和时间
* Hadoop提供了ZKFailoverController角色，部署在每个NameNode的节点上，作为一个deamon进程, 简称zkfc，示例图如下：



### DataNode

DataXceiver

..hdfs..server.datanode.DataXceiverServer

### FileSystem

+ get()

**PARAMs : {**

**hadoop.security.kerberos.ticket.cache.path**

**}**

#### DistributedFileSystem

#### Sender

### IO

#### BlockReader

#### RemoteBlockReader2

- bytesPerChunksum [512]

+ newBlockReader( )

#### BlockReaderFactory

+ build()

+ BlockReader getRemoteBlockReaderFromTcp()

+ getRemoteBlockReader()

#### DFSClient

+ static DFSInputStream open()

+ static DFSOutputStream create()

+ static LocatedBlocks getLocatedBlocks( String, long)

+ static callGetBlockLocations( ClientProtocol)

#### DFSInputStream

// 从NameNode 中获取多个文件块的信息

- LocatedBlocks

+ openInfo()

**PARAMs: {**

**dfs.client.retry.times.get-last-block-length [3]**

**dfs.client.retry.interval-ms.get-last-block-length [4000ms]**

**}**

+ readWithStrategy( ReaderStrategy, ...

currentNode = blockSeekTo()

+ DatanodeInfo blockSeekTo()

// blockReader created

+ DNAddrPair chooseDataNode( LocatedBlock)

+ readBuffer()

ByteArrayStrategy [IC,]

#### PacketReceiver

+ doRead()

|<--- int --->|<--- short -->|<-- bytes -->|

|- payload -|- header len --|-- protobuf -- |

#### DFSOutputStream

- String[] favoredNodes

- DataStreamer

+ static DFSOutputStream newStreamForCreate()

+ static ... createSocketForPipeline(DatanodeInfo, DFSClient)

**PARAMs : {**

**DFS\_CLIENT\_WRITE\_PACKET\_SIZE\_KEY**

**dfs.client.block.write.locateFollowingBlock.retries [5]**

**dfs.client.slow.io.warning.threshold.ms [30000ms]**

**}**

+ start()

+ write()

+ writeBytes()

DataStreamer [IC, ]

+ locateFollowingBlock

### ../hadoop/crypto/..

#### KeyProvider

### ../hadoop/net/..

### ../hadoop/ipc/..

#### Server

- int handlerCount // number of handler threads

- int readThreads // number of read threads

- int maxQueueSize // PARAMs -->

**PARAMs {**

**ipc.server.handler.queue.size [100]**

**ipc.server.read.threadpool.size [1]**

**}**

### ../hadoop/hdfs/protocol/..

#### ClientProtocol

// src -- file name

+ getBlockLocations(String src, long offset, long length)

+ create()

#### ClientDatanodeProtocol

#### DatanodeInfo

LocatedBlocks

#### LocatedBlock

- ExtendedBlock b

- long offset // offset of the first byte of the block in the file

- boolean corrupt //

- String[] storageIDs // cached storageId for each replica

- StorageType[] // cached storage type for each replica, if reported

- Datanodeinfo[] // List of cached datanode locations

+ DatanodeInfo[] getLocations

#### HdfsFileStatus

### BlockManagement

#### BlockManager

#### BlockIdManager

### SecondNameNode

作用：对主NN的一个补充，它会周期的执行对HDFS元数据的检查点。每个HDFS只有单个SecondNameNode节点

1.检查点

唤醒条件，

**CONFIG {**

**fs.checkpoint.size**

**[edits文件大小超过该值 (67108864|byte)]**

**dfs.namenode.checkpoint.period**

**[两次检查点创建间隔(3600|s)]**

**dfs.namenode.checkpoint.txns**

**[edits事物条数限制]**

**dfs.namenode.num.checkpoints.retained**

**dfs.namenode.num.checkpoints.retained**

**[]**

*****dfs.namenode.num.checkpoints.retained*****

--最多能保存的edits文件个数，默认为1,000,000. 官方解释是为防止standby namenode宕机导致edits文件堆积的情况，设置的限制，不是太明白。

*****dfs.ha.tail-edits.period*****

--standby namenode每隔多长时间去检测新的Edits文件。只检测完成了的Edits， 不检测inprogress的文件。

**}**

从NN下载fsimage和edits文件 （loadFSImage/loadFSEdits）

合并上传NN替换image文件

2. 防止edits文件过大，合并edits和fsimage

fsimage文件和edits文件，NN的目录树存储在fimage上，当NN启动时加载fimage到内存，edits文件记录fsimage变更，fimage和edits合并发生在NN启动的时候，

### JournalNode

**CONFIG {**

**dfs.namenode.shared.edits.dir**

**[设置一组 journalNode 的 URI 地址，active NN 将 edit log 写入这些JournalNode，而 standby NameNode 读取这些 edit log，并作用在内存中的目录树中。如果journalNode有多个节点则使用分号分割。该属性值应符合以下格式qjournal://host1:port1;host2:port2;host3:port3/journalId]**

**dfs.journalnode.edits.dir**

**[JournalNode 所在节点上的一个目录，用于存放 editlog 和其他状态信息]**

**}**

### NameNode

**START\_OPT {**

**..**

**BOOTSTRAPSTANDBY**

**}**

**CONFIG {**

**dfs.ha.automatic-failover.enabled**

**[开启NN故障自动切换 ()]**

**dfs.namenode.plugins**

**}**

Initialize ()

{

...

NNRpcServer start

FSNamesystem start

NNHttpServer start

ServicePlugin start

}

#### NameNodeRpcServer

- FSNamesystem

- RPC.Server

serviceRpcServer // listen to requests from DataNodes

clientRpcServer // listen to requests from clients

+ allowSnapshot()

+ createSnapshot()

+ setStoragePolicy()

+ create()

#### FSDirectory

- String blockPoolId

+ BlockInfoContiguous addBlock()

#### FSNamesystem

**PARAMs {**

**}**

+ loadFromDisk()

+ LocatedBlock getAdditionalBlock()

+ LocatedBlock storeAllocatedBlock()

+ DatanodeStorageInfo[] getNewBlockTargets()

+ FileState analyzeFileState()

+ startFileInt(

EnumSet<CreateFlag> flag,

long blockSize,

)

+ strartFileInternal()

+ setNewINodeStoragePolicy

+ internalReleaseLease

+ saveAllocatedBlock

#### INodeDirectory

+ addSnapshot(int id, String name)

#### INode

##### INodeFile

- BlockInfoContiguous[] // blocks

+ recordModification

#### INodesInPath

- long header // storagePolicyId

- INode[] // inodes

- int snapshotId

- boolean isSnapshot

[.snapshot ]

HDFS快照是一个只读的基于时间点文件系统拷贝。快照可以是整个文件系统的也可以是一部分。常用来作为数据备份，防止用户错误和容灾。

HDFS实现了：

* snapshot 创建的时间 复杂度为O(1)，但是不包括INode 的寻找时间
* 只有当修改snapshot时，才会有额外的内存占用，内存使用量为O(M),M 为修改的文件或者目录数
* 在datanode 上面的blocks 不会复制，做snapshot 的文件是记录了block的列表和文件的大小，但是没有数据的复制
* snapshot 并不会影响HDFS 的正常操作：修改会按照时间的反序记录，这样可以直接读取到最新的数据。快照数据是当前数据减去修改的部分计算出来的。

**COMMAND {**

hdfs lsSnapshottableDir

hdfs snapshotDiff <path> <fromSnapshot> <toSnapshot>

**}**

+ INodesInPath resolve()

#### LeaseManager

**class Lease**

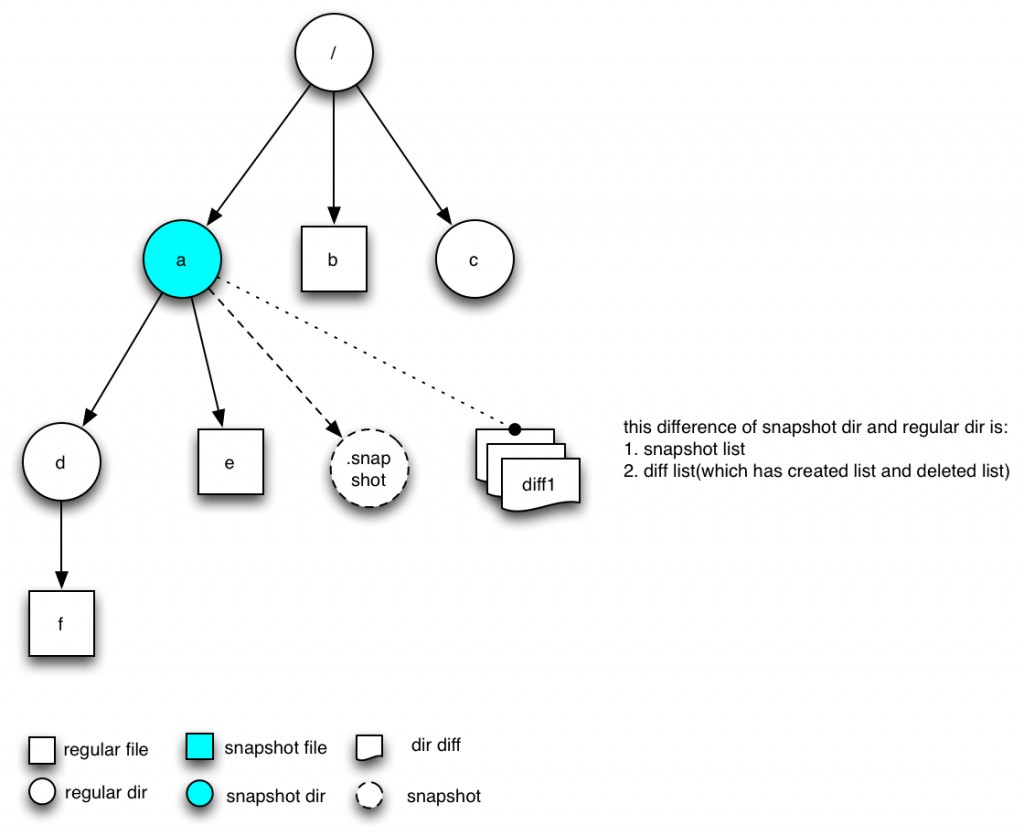
String holder // 客户端

long lastUpdate // 最后更新时间

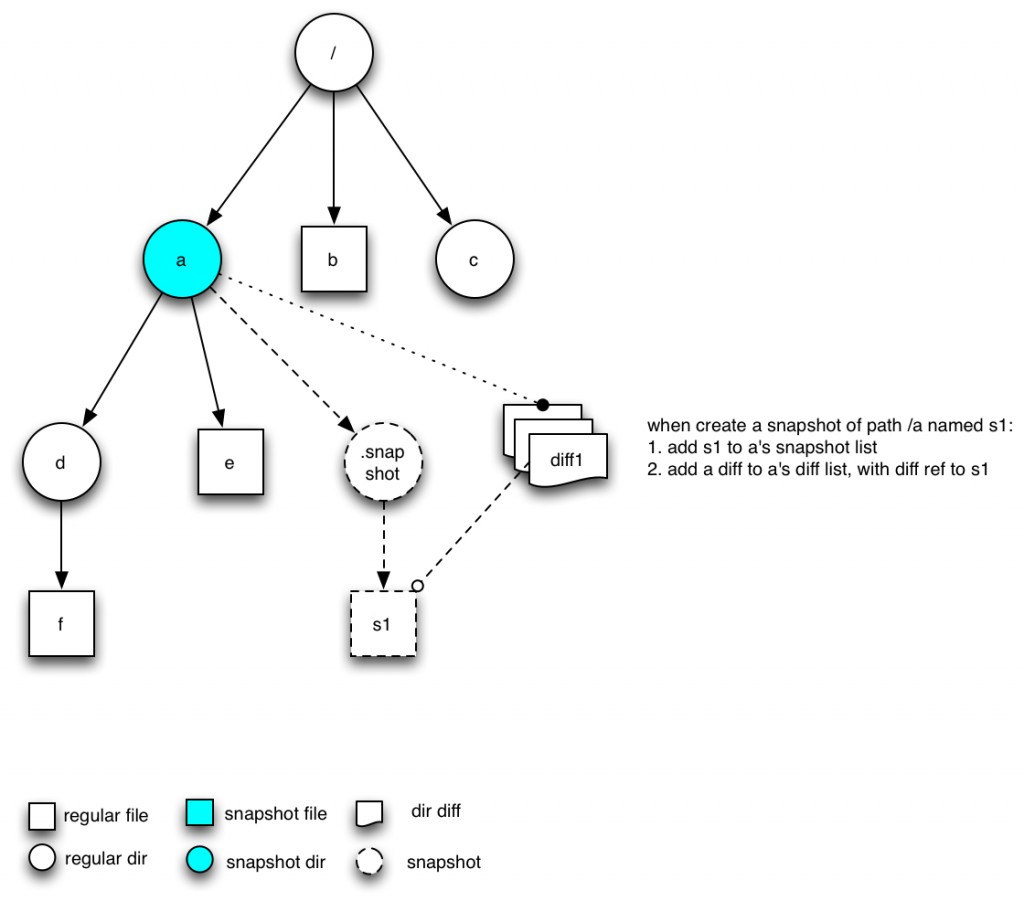
**class Moniter** // checks for the leases that have expired

#### ../snapshot

1.allowSnapshot



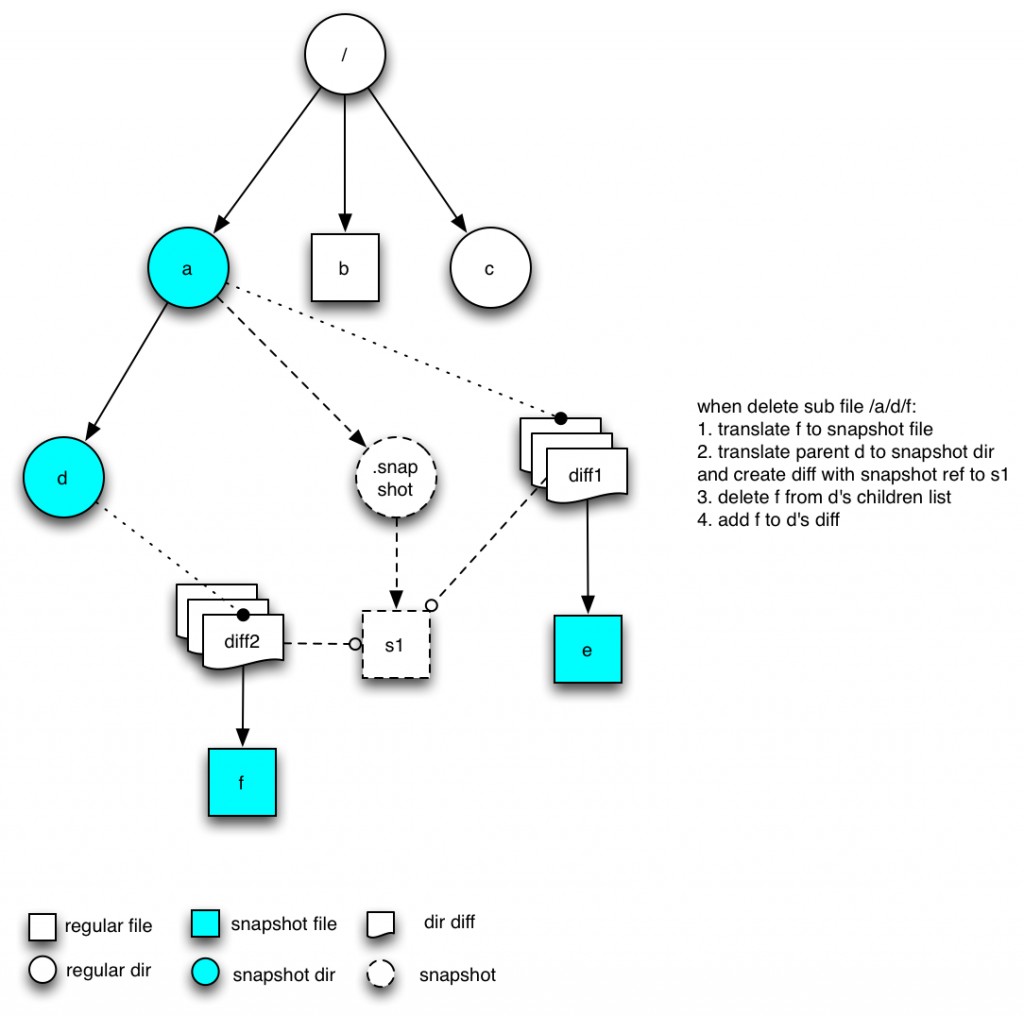
2. createSnapshot



3. 删除e

不论是删除一个文件还是一个目录，只要是直接子节点，都会将节点转换为快照版本.例如e会变成INodeFileWithSnapshot，在a的DirectoryDiff中ChildDiff中deleted列表中将会包含e,而在a的正常节点下会被删除。目录节点的处理同样。

1. 删除子孙节点

处理这种节点的原则是：先将孙子节点转变为Snapshot版本，然后将父节点变为snapshot版本，同时将孙子节点版本加入到直接父节点的diff列表中。为了能够通过同一个snapshot找到当时的文件，需要将新的diff指向到老的snapshot版本上。图中d节点是INodeDirectoryWithSnapshot（不是INodeDiretorySnapshottable, 本身不允许在d上创建snapshot)

##### DirectorySnapshottableFeature

- DirectoryDiffList

+ createDiff()

+ addSnapshot(INodeDirectory snapshotRoot, int id, String name)

snapshotRoot.updateModificationTime(); // save

**ChildrenDiff [IC, ]**

**DirectoryDiff [IC,]**

**DirectoryDiffList [IC,]**

##### INodeFileWithSnapshot

### DataNode

### HistoryServer

### ../hadoop/hdfs/protocolPB

#### ClientNamenodeProtocolTranslatorPB

+

+ addBlock( ..

ExtendedBlock

DatanodeInfos[]

..

String[] // favoredNodes

)

+ waitForWritable()

../hadoop/net/SocketIOWithTimeout

### 配置文件

#### hdfs-site.xml

 NameNode在启动的时候 ****首先****进入****安全模式，****如果datanode丢失的block达到一定的比例（由hdfs-site.xml文件中dfs.safemode.threshold.pct决定，默认0.999f），则系统会一直处于安全模式状态即只读状态 ； 否则没有其他情况影响，一般情况下，系统会自动离开安全模式

## YARN

**CONFIG{**

**/-------------------- HA -----------------------/**

**/------------------------------------------------/**

**yarn.nodemanager.resource.memory-mb**

**yarn.nodemanager.aux-services.mapreduce\_shuffle.class**

**[(org.apache.hadoop.mapred.ShuffleHandler)]**

**yarn.nodemanager.aux-services**

**NodeManager上运行的附属服务。需配置成mapreduce\_shuffle，才可运行MapReduce程序**

**yarn.resourcemanager.address**

**[客户端通过该地址向RM提交应用程序(8032)]**

**yarn.resourcemanager.scheduler.address**

**[AM通过该地址向RM申请资源、释放资源等(8030)]**

**yarn.resourcemanager.resource-tracker.address**

**[ResourceManager 对NodeManager暴露的地址.。NodeManager通过该地址向RM汇报心跳，领取任务等]**

**yarn.scheduler.minimum-allocation-mb**

**[单个任务可申请的最小物理内存量 1024MB(default)]**

**yarn.scheduler.maximum-allocation-mb**

**[单个任务可申请的最大物理内存量 8192MB(default)]**

**yarn.scheduler.minimum-allocation-vcores**

**[单个任务可申请的最小虚拟CPU个数 1(default)]**

**yarn.scheduler.maximum-allocation-vcores**

**[单个任务可申请的最多虚拟CPU个数 32(default)]**

**yarn.nodemanager.resource.cpu-vcores**

**yarn.nodemanager.vmem-check-enabled**

**[虚拟内存检查]**

**yarn.nodemanager.pmem-check-enabled**

**[物理内存检查]**

**yarn.resourcemanager.am.max-attempts**

**[am重试次数]**

**yarn.nodemanager.local-dirs**

**[中间结果存放位置]**

**yarn.nodemanager.delete.debug-delay-sec**

**[执行结束后延迟n(min)删除本地文件及日志]**

**Yarn.resourcemanager.recovery.enabled**

**[RM挂了相应的正在运行的任务在RM恢复后不能重新启动]**

**yarn.resourcemanager.store.class**

**[org.apache.hadoop.yarn.server.resourcemanager.recovery.FileSystemRMStateStore，**

**基于Hadoop文件系统的实现。还可以为**

**org.apache.hadoop.yarn.server.resourcemanager.recovery.ZKRMStateStore]**

**}**

### ../resourcemanager/..

#### ResourceManager

- RMContextImle

+ serviceStart()

### ../client/..

#### AbstractService

- AMRMClient

- HeartbeatThread

- CallbackHandlerThread

+ serviceStart()

##### AMRMClientAsync

+ onContainersCompleted

+ onContainersAllocated

#### TimelineClient

## HIVE

### SQL语法

#### 数据类型

array

maps

**struct**

Eg: CREATE TABLE test (id INT, info struct<name:STRING, age:INT>)

> ROW FORMAT DELIMITED FIELDS TERMINATED BY','

> COLLECTION ITEMS TERMINATED BY ':';

**FIELDS TERMINATED BY** // 字段与字段之间的分隔符

**COLLECTION ITEMS TERMINATED BY** // 一个字段各个item的分隔符

UDF

HiveServer2

Hive中定义schema

add JAR

create temporary function yhighlight as 'cn.net.ycloud.ydb.handle.fun.YHighlightSummary';

## SPARK