分散処理アプリ演習　第5回
Hadoop動作詳細

（株）NTTデータ

講義内容

1. HDFS動作詳細
   - HDFSメタデータ
   - HDFSチェックポイント
2. HDFSに関するポイント
   - DataNode操作
   - HDFSの持つ機能
3. MapReduce動作詳細
   - MapReduceジョブ実行フロー
   - Mapタスク/Reduceタスクの仕組み
   - タスクスケジューラ
4. MapReduceに関するポイント
   - データのローカリティの考え方
   - 投機的実行

1. HDFS詳細

振り返り - HDFSの構成とは（過去の講義より）

1. NameNode
   - HDFSのメタデータを管理する
2. DataNode
   - HDFSのデータの実体であるブロックを保存する
3. SecondaryNameNode
   - HDFSのメタデータのチェックポイントで利用する

- Hadoopに関する各種設定ファイル
  - /etc/hadoop/conf/ ディレクトリ内に設定ファイルが配置されている
  - core-site.xml：Hadoopの共通的な設定を記述する設定ファイル
  - hdfs-site.xml：HDFS関係の設定を記述する設定ファイル
  - mapred-site.xml：MapReduce関係の設定を記述する設定ファイル

HDFS上のファイルのアクセス

- HDFS上のデータは、以下のクラスを利用してアクセスする
  - 読み込み：org.apache.hadoop.fs.FSDataInputStream クラス
  - 書き込み：org.apache.hadoop.fs.FSDataOutputStream クラス

- HDFS上では、dfs.block.sizeで指定されたブロック単位でデータを保存する
  - DataNodeのローカルディスクにて、dfs.data.dir プロパティで指定したディレクトリに保存
  - DataNodeのローカルディスクでは、blk_<ランダムな数字> のファイル名でブロックを保存
  - DataNodeでは、ブロックと一緒に blk_<ランダムな数字>.meta ファイル(チェックサム用データ)も保存される

デモ：DataNode上でのデータの確認

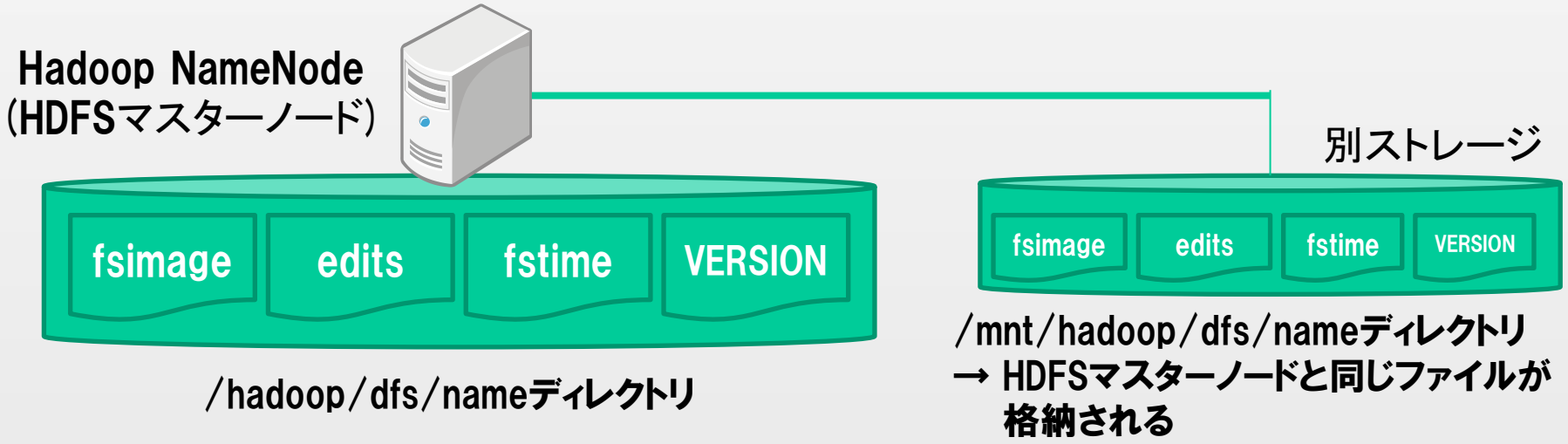
- 講師環境のDataNodeにアクセスして、データを確認する
  1. 講師の環境で、適当なサンプルファイルを作成
  2. hadoop fs -put コマンドでHDFS上に格納
  3. NameNodeのWeb画面でデータの配置を確認
  4. sshでDataNode（1台）にログイン
  5. DataNode用ディレクトリを参照して、データを確認

HDFSで持つメタデータ

1. fsimage
   - HDFSのパス、ファイルサイズ、権限(オーナ・グループ・パーミッション)、作成日時、更新日時を記録
   - SequenceFile形式で記録
2. edits
   - HDFS操作情報(create, delete, open, rename) を記録
   - SequenceFile形式で記録
3. fstime
   - fsimage更新日時を記録
4. VERSION
   - HDFSに関するバージョン情報
   - namespaceID (HDFSを識別する文字列、HDFSフォーマット時にランダムに生成)
   - layoutVersion (HDFSメタデータフォーマットのバージョン)

HDFSメタデータの格納場所

- NameNode
  - dfs.name.dir（hdfs-site.xmlで記述）プロパティに設定
  - 複数のパスを記述することで、同じ情報を複数の領域に記録可能
    - DRBDやNFSなどと併用
  - 運用中は、NameNodeのヒープメモリでメタデータを保持しつづける
    - クライアントからのアクセスではメモリ上のデータを参照して結果を返す



（例: dfs.name.dirの値を”/hadoop/dfs/name,/mnt/hadoop/dfs/name”とした場合）

デモ：NameNodeでのメタデータの確認

- 講師環境のNameNodeにアクセスして、データを確認する
  1. sshでNameNode（1台）にログイン
  2. NameNode用ディレクトリを参照して、メタデータを確認

HDFSメタデータの更新

- HDFSメタデータは、起動時やチェックポイントなどのタイミングで更新される

NameNode

メタデータはNameNodeのメモリ上で管理

- NameNode起動時
- SecondaryNameNodeによるチェックポイント時
- savenamespaceコマンド実行時

- NameNode起動時
- SecondaryNameNodeによるチェックポイント時
- savenamespaceコマンド実行時

SecondaryNameNode

初期化のタイミング

- NameNode起動時
- SecondaryNameNodeによるチェックポイント時
- savenamespaceコマンド実行時

書き込みのタイミング

- HDFSクライアントからHDFS操作（read, write, create, rename, delete）が実行されたとき

デモ: NameNodeのメタデータ更新の確認

- 講師環境のNameNodeにアクセスして、データを確認する
  1. NameNode用ディレクトリを参照して、メタデータを確認
  2. クライアントより適当なファイルをHDFSに格納
  3. NameNode用ディレクトリを参照して、メタデータの更新を確認

HDFSメタデータのバックアップ

- メタデータの消失はHadoopクラスタの破壊と同じであるため、メタデータをバックアップする必要がある
- バックアップには以下の方法がある

1. SecondaryNameNodeによるチェックポイント

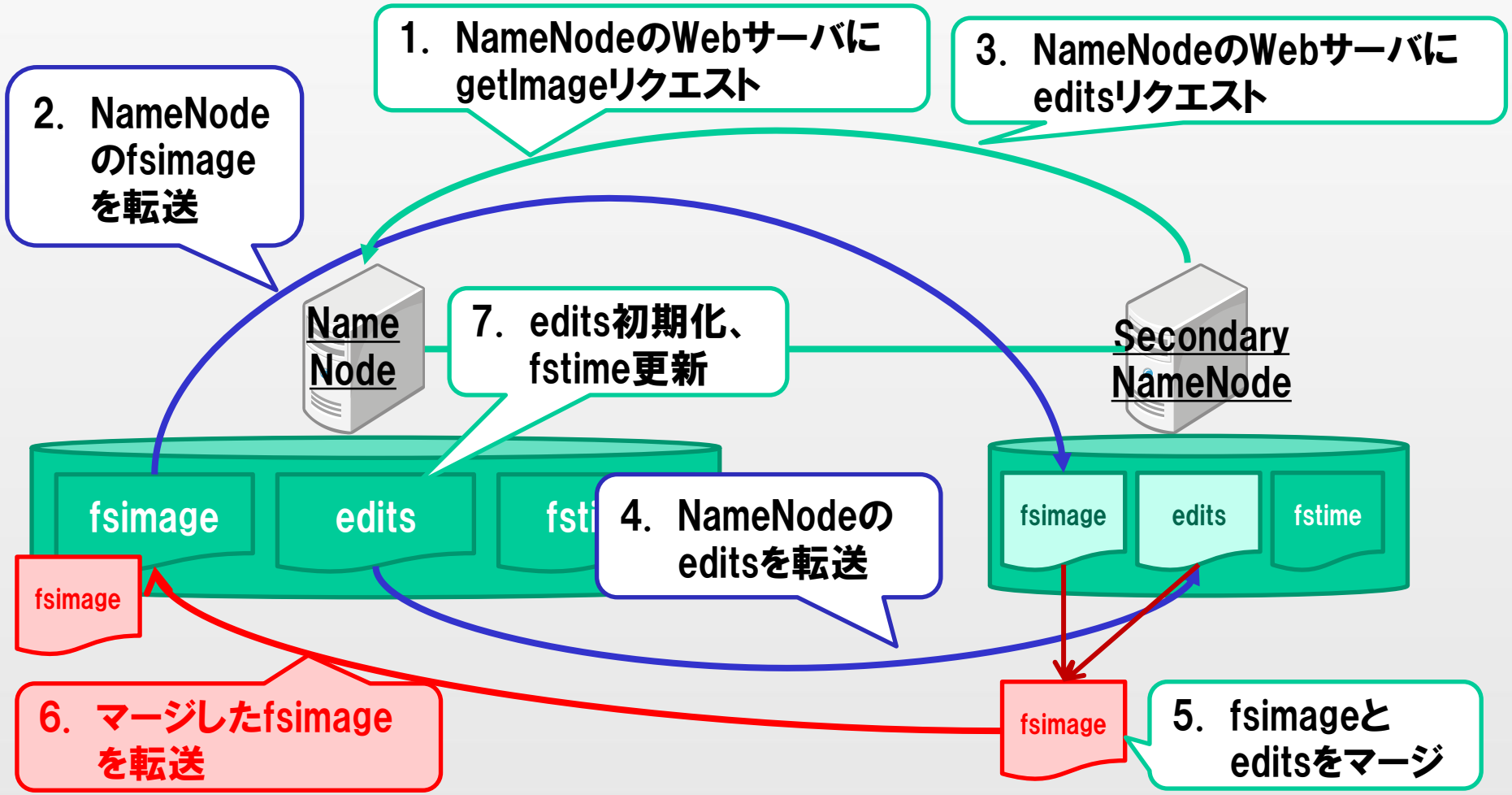
- SecondaryNameNodeにより、定期的にチェックポイント処理が実行されNameNodeのディスクにダンプされる（NameNodeのファイルを利用）
- チェックポイントのタイミングは以下のどちらかを満たした場合である
  - 前回のチェックポイントから60分が経過（fs.checkpoint.periodにて秒単位で指定）
  - editsが閾値（fs.checkpoint.sizeでバイト単位で指定、デフォルト64MB）以上
- secondarynamenodeコマンドにて強制的に実行することも可能

2. savenamespaceコマンドによるチェックポイント

- dfsadmin -savenamespaceコマンドでNameNodeメモリ上のHDFS構造をfsimageファイルにダンプする（NameNodeのメモリ上のデータを利用）
- 本コマンドの実行は、HDFSはセーフモードの状態であることが前提

SecondaryNameNodeによるチェックポイント

- SecondaryNameNodeによるチェックポイントは以下のように実行される
  - ノード間はHTTPにて通信される



SecondaryNameNodeによるチェックポイントの注意点

- SecondaryNameNodeでのチェックポイントは、以下に注意する
  - 大量ファイルの更新・操作が頻発する場合edits閾値超えによるチェックポイントが多発、fsimageがGBを超えるサイズとなると通信時間が増加しチェックポイントが遅延する
    - editsチェックポイントのサイズを変更（64MBより大きな値に変更）
    - NameNode-SecondaryNameNodeの通信路をHadoop処理用の通信路と別に用意
  - SecondaryNameNodeもNameNodeと同様にメタデータをメモリ上で管理するため、NameNodeと同等のヒープメモリが必要
    - 他ノードと共存させる場合は、メモリ容量に注意すること
    - JavaVMのオプションもNameNodeと同じ設定にすること

デモ: SecondaryNameNodeの確認

- 講師環境のSecondaryNameNodeにアクセスして、チェックポイントについて確認する
  1. SecondaryNameNodeにsshでアクセス
  2. SecondaryNameNodeのログファイルを参照し、チェックポイント内容を確認
  3. SecondaryNameNodeのデータ格納領域を参照し、データを確認

2. HDFSに関するポイント

HDFSに関するポイント

- fsck - HDFSメタデータのチェックについて
- HDFSデータノードのメンテナンス
- HDFSのゴミ箱機能
- クオータ

fsck - HDFSメタデータのチェック

- fsckはHDFS上で管理されているファイルとブロックの関係にて、ブロックが欠損または十分なレプリケーション数を保持していないかを確認するために実行する

```
## HDFS上のすべての領域に対してfsckを実行する場合
$ hadoop fsck /
```

- fsckでは、ブロック異常の確認に加えて、HDFS修復（異常ブロックを持つファイルの削除や/lost+foundへの移動）も実行できる

```
## HDFS上で欠損があるブロックをもつファイルを削除する場合
$ hadoop fsck / -delete
```

- 各ファイルごとにブロックIDやサイズ、レプリケーション数、格納場所（ラック情報を含めた格納場所）も確認できる

```
## /user/hogehoge ディレクトリ以下のブロック情報や格納情報を確認する場合
$ hadoop fsck /user/hogehoge -files -blocks -locations -racks
```

fsck - HDFSメタデータのチェック

- fsckは、NameNodeのメモリ上で管理されている情報を確認して結果を出力する仕組みである
  - ブロック情報（ブロックID、格納先DataNode名、ブロックサイズ）
  - ディレクトリ、ファイル情報（ブロック数、レプリケーション数など）

- fsckは、コマンドで実行された内容がNameNodeのWeb機能によって処理される
  - Webブラウザ上から確認することも可能

```
## NameNode (nn:50070) で、/user/hogehogeを対象にfsckを実行する場合
http://nn:50070/fsck?path=/user/hogehoge
※ 要webuserでのアクセス権限
```

- fsckは、NameNodeのメモリ情報の参照、HDFSアクセスに関するログ出力とNameNodeにとって負荷が高い処理となる
  - 大規模なMapReduceジョブを実行中にfsckを実行しないこと

【参考】DataNodeのメンテナンス

- HadoopクラスタのDataNodeをメンテナンスする場合は、HDFSのサービスを継続した状態で作業可能である
  - メンテナンス：ソフトウェアアップグレード、ハードウェア交換 など

- DataNodeを制御する手順は以下の通りである
  1. NameNodeのhdfs-site.xmlに以下のプロパティを記載する
     - dfs.hosts.exclude プロパティ
     - 設定値：メンテナンス対象のDataNodeサーバー覧を記述したファイルのパスを記述
  2. 1. で指定したファイルに、切り離すDataNodeサーバ名を記述する
  3. NameNodeで以下のコマンドを実行する

```
$ hadoop dfsadmin -refreshNodes
```

  4. 切り離すDataNodeに含まれるブロックの再配置が実行される
  5. ブロックの再配置が完了したら、NameNodeによってDataNodeが切り離される
  6. DataNodeはプロセス停止となる
  7. 復旧時は、2. の記述を削除し、3を実行した上でDataNodeを起動する

【参考】HDFS上のデータのゴミ箱

- WindowsやLinuxと同様に、HDFSにもゴミ箱機能が提供されている
- ゴミ箱機能を利用することで、操作ミスによってHDFS上のファイルが即座に削除されることを防ぐ

- ゴミ箱機能を利用する場合は、core-site.xml に以下を設定する
  - fs.trash.interval プロパティ
  - 設定値：ゴミ箱に保存しておく期間（分単位）

- ゴミ箱機能を有効にした場合、削除コマンドを実行するとゴミ箱領域にデータが移動される

```
## hogehogeユーザの場合 /user/hogehoge/.Trash にデータが移動される
$ hadoop fs -rm /user/hogehoge/fugafuga.txt
```

- ゴミ箱をスキップしてファイルを削除する場合は、-skipTrashを付与する

```
## hogehoge.txt をゴミ箱をスキップして削除する場合
$ hadoop fs -rm -skipTrash hogehoge.txt
```

【参考】クオータ

- ユーザ単位でHDFS利用に関するクオータを設定できる
  1. HDFS上に格納できるファイル数（ディレクトリも含む）
  2. HDFS上に保存できる容量
- 設定方法
  1. HDFSに格納できるファイル数に関するクオータ

```
## /user/hogehoge以下に100000ファイル（ディレクトリ含む）クオータを設定
## クオータ設定数以上のファイル（ディレクトリ）を格納する場合は、
## NSQuotaExceededExceptionが返される
$ hadoop dfsadmin -setQuota 100000 /user/hogehoge
```

  2. HDFS上に保存できる容量に関するクオータ

```
## /user/hogehoge以下に64MBファイル（ディレクトリ含む）クオータを設定
## クオータ設定サイズ以上のファイル（ディレクトリ）を格納する場合は、
## DSQuotaExceededExceptionが返される
$ hadoop dfsadmin -setSpaceQuota 67108864 /user/hogehoge
```

  - 容量に関するクオータは、HDFSファイルサイズ×レプリケーション数で計算されることに注意

3. MapReduce詳細

振り返り – MapReduceの構成とは（過去の講義より）

1. JobTracker
   - Jobクライアントから MapReduceジョブを受け付け、タスクに分割
   - 分割されたタスクを各TaskTrackerに配布
   - TaskTrackerの死活確認

2. TaskTracker
   - JobTrackerから割り当てられたタスクを実行するためにChildプロセスを生成
   - Childプロセスで実行されているタスクの監視
   - JobTrackerへのハートビート通信（死活・タスク実行状況）

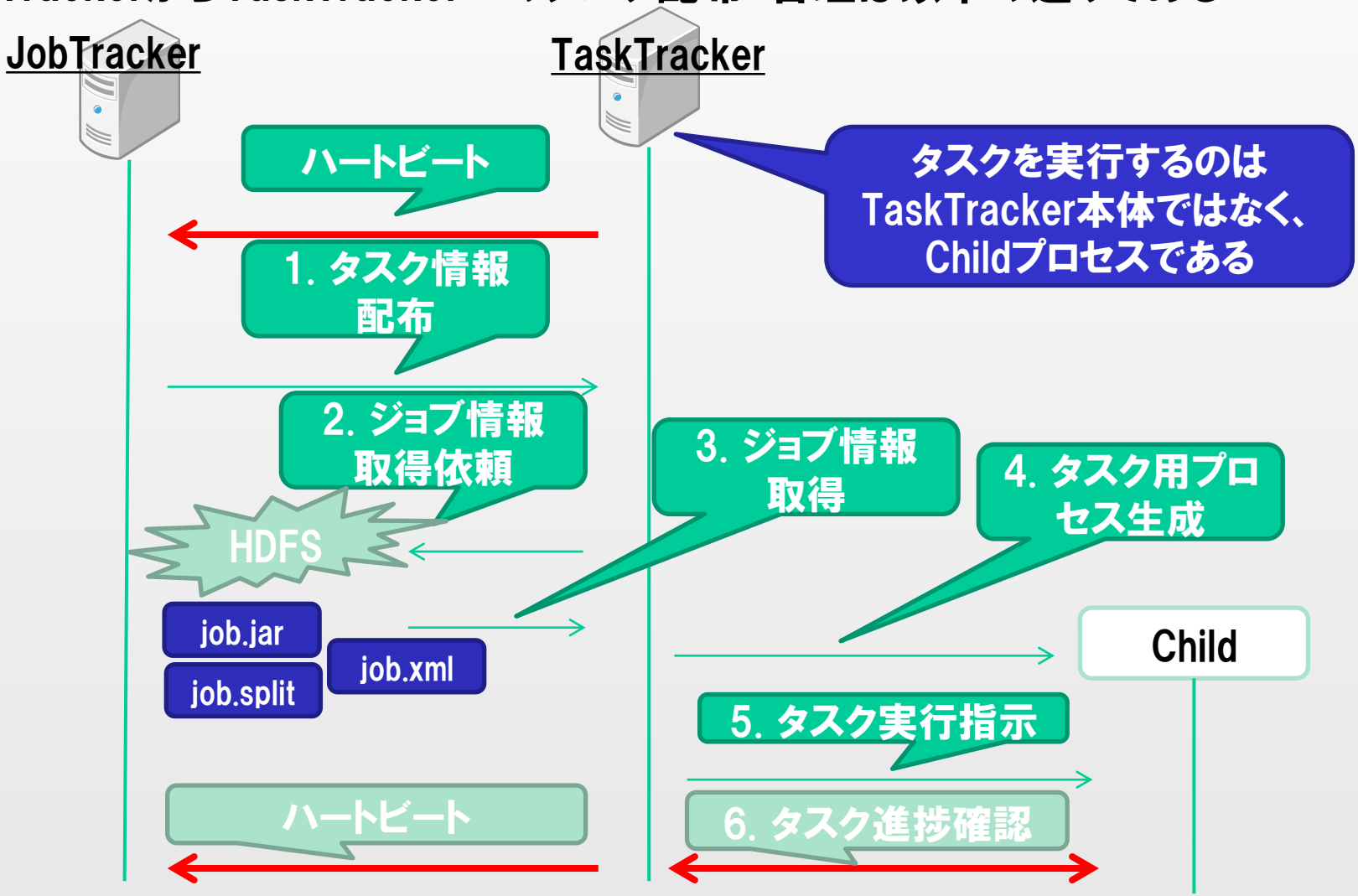
3. Jobクライアント（JobClient）
   - MapReduceジョブをJobTrackerに転送
   - ジョブの進捗状況を確認

MapReduceジョブ実行のフロー 1

- JobクライアントとJobTrackerのやりとりは以下の通りである

MapReduceジョブ実行のフロー 2

- JobTrackerからTaskTrackerへのタスク配布・管理は以下の通りである



【参考】MapReduceとHDFSの関係

- MapReduceジョブ実行に必要な情報はHDFSで保存される
  - JobTrackerやTaskTrackerは、HDFS上に保存したファイルを個別に取得する
  - MapReduceジョブ実行に必要なファイルは、mapred.submit.replicationプロパティによってレプリケーション数10で配布される

- MapReduceジョブ実行前にHDFS上のブロック情報を確認し、結果をjob.splitファイルに記録する（データのローカリティを考慮したタスク割り当て）
  - タスク実行に関するデータ取得の通信を削減するため

- MapReduceジョブ処理結果は、自サーバで実行しているDataNodeでブロックを保持し、他DataNodeにレプリケーションする
  - 処理結果配布に関して、データ転送の通信量を削減するため

デモ: MapReduce処理での動作確認

- 講師環境で、Hadoopのexampleパッケージに含まれるsleepを動作させて、MapReduceジョブ実行に必要な情報を確認する

```
## sleepジョブの実行（rootで実行）
$ hadoop jar /usr/lib/hadoop/hadoop-examples.jar sleep -mt 600000 -m 1

## JobTrackerのデータ領域を参照
$ hadoop dfs -ls /mapred/staging/root/.staging/
```

ジョブとタスクの関係

- ジョブ：Jobクライアントより投入されるジョブ
  - ジョブの識別は、ジョブIDによって実施される
  - ジョブIDは、JobTrackerの起動時間をベースに生成されている

```
## JobTrackerが 2012年2月29日 10時20分に起動、3番目のMapReduceジョブの場合
job_201202291020_0003
```

- タスク：ジョブに含まれるMap処理とReduce処理のこと
  - Map処理：入力となるブロック（ファイル）ごとに実行される
  - Reduce処理：クライアントが指定した数に応じて実行される
  - タスクの識別は、タスクIDによって実施される
  - タスクIDは、ジョブIDに基づいて生成される

```
## IDがjob_201202141818_0007であるMapReduceジョブの3番目のMap処理の場合
task_201202141818_0007_m_000003
## IDがjob_201201302020_0120であるMapReduceジョブの10番目のReduce処理の場合
task_201201302020_0120_r_000010
```

タスク試行回数

- MapReduceジョブで、処理されるタスクは、1つのタスクに付き最大4回まで再試行が可能である
  - 各タスクを実行しているノードに異常が発生し、タスクが継続できなくなった場合でもMapReduceジョブ全体として継続させるための仕組み

- タスク試行に関しては、タスク試行ID (TaskAttemptID) で管理される

```
## タスクIDがtask_201201011234_0100_m_000000の場合の1回目のタスク試行ID
attempt_201201011234_0100_m_000000_0
```

- タスクの再試行回数は、以下のプロパティで設定できる
  - JobクライアントのMapred-site.xmlかConfiguration.setメソッドで設定する
  - Mapタスク再試行回数：mapred.map.max.attempts
  - Reduceタスク再試行回数：mapred.reduce.max.attempts

- タスクの再試行は、MapReduceアプリケーション自体にバグがある場合でも指定した回数分再試行されることに注意

TaskTrackerのブラックリスト化

- JobTrackerは、タスクの実行状況を確認し、FAILEDが多いTaskTrackerにタスクを割り当てないように動作する（ブラックリスト）
  - 異常が多いTaskTrackerにタスクを割り当てて、FAILEDとなる場合の処理時間の長期化を抑えるため

- JobTrackerがTaskTrackerをブラックリストにする方針は、以下の2つである
  - MapReduceジョブ単位で、FAILEDになった回数が閾値を超える場合
    - mapred.max.tracker.failuresプロパティで設定、デフォルト4
    - ジョブ単位のブラックリストTaskTrackerの総数がクラスタ全体の1/4未満の場合に、ジョブ単位のブラックリストに割り当てられる
    - 他のMapReduceジョブには影響しない
  - MapReduceジョブ全体で、FAILEDになったタスクの割合が閾値を越える場合
    - mapred.max.tracker.blacklistsプロパティで設定、デフォルト4
    - TaskTracker全体のFAILED回数を確認し、平均値より50％高い場合に、MapReduce環境としてのブラックリストに適用される
    - ブラックリストになったTaskTrackerは、24時間TaskTrackerとして利用できない

タスクスケジューラ

- MapReduceジョブの各タスクをTaskTrackerに割り当てる操作は、タスクスケジューラによって実行される
  - JobTracker内に、taskScheduler.assignTasksメソッドを呼び出し、別途設定したタスクスケジューラに応じたタスクスケジューリングを実現する

- タスクを割り当てる方法として、利用者（Jobクライアント）から設定できる内容は以下の通りである
  - MapReduceジョブの優先度
    - 5段階：VERY_LOW，LOW，NORMAL，HIGH，VERY_HIGH
  - タスク割り当てに利用するキュー情報
    - mapred.job.queue.nameプロパティで設定可能
    - 存在しないキューを利用することはできない
      - キューは、mapred.queue.namesプロパティで設定

タスクスケジューラ設定

- JobTrackerのmapred-site.xmlに利用したいスケジューラを記述する
  - プロパティ名：mapred.jobtracker.taskScheduler
  - 記述しない場合のデフォルトは、(JobQueueTaskSchedulerが利用される)

```
<configuration>
  ・・・途中省略・・・
  <!-- スケジューラでJobQueueTaskSchedulerを設定する場合 -->
  <property>
    <name>mapred.jobtracker.taskScheduler</name>
    <value>org.apache.hadoop.mapred.JobQueueTaskScheduler</value>
  </property>
  ・・・途中省略・・・
</configuration>
```

Hadoopですぐ利用できるタスクスケジューラ

- Hadoopでは、以下のスケジューラがすぐに利用可能である

1. JobQueueTaskScheduler
   - Hadoopデフォルトのスケジューラ
   - 1つのキュー (default) にてタスクをスケジューリングする
2. CapacityTaskScheduler
   - Hadoopのcontributeなスケジューラ
   - 複数のキューにタスク割り当て比率を設定してタスク割り当てをスケジューリングする
   - 割り当て比率に応じてスケジューリングする
3. FairScheduler
   - Hadoopのcontributeなスケジューラ
   - 複数のプール (キューとは異なる) にタスク割り当て比率を設定してスケジューリングする
   - プールの比率に応じて、実行するジョブに公平にタスクを割り当てる

タスクスケジューラ利用のポイント

- タスクスケジューラを利用するポイントは以下の通りである
  - 1人の利用者(目的が1つ)のみでMapReduceジョブを実行する場合
    - JobQueueTaskSchedulerを利用する
    - MapReduceジョブの優先度に従いタスクを割り当てられる
  - 処理スロット数が潤沢(3桁以上のスロット数)で厳密にリソースを割り当てたい場合
    - CapacityTaskSchedulerを利用する
    - リソース割り当てに関する割合を設定して、他のキューのジョブの影響を受けないようにする
  - 複数の利用者で大規模なMapReduceジョブが実行されてリソースをほぼ占有していても小規模なMapReduceジョブも継続して実行して欲しい場合
    - FairSchedulerを利用する
    - タスク割り当て設定に従い、プールに公平にタスクを割り当てる
    - 大規模な処理は、プリエンプションでタスク割り込みも可能

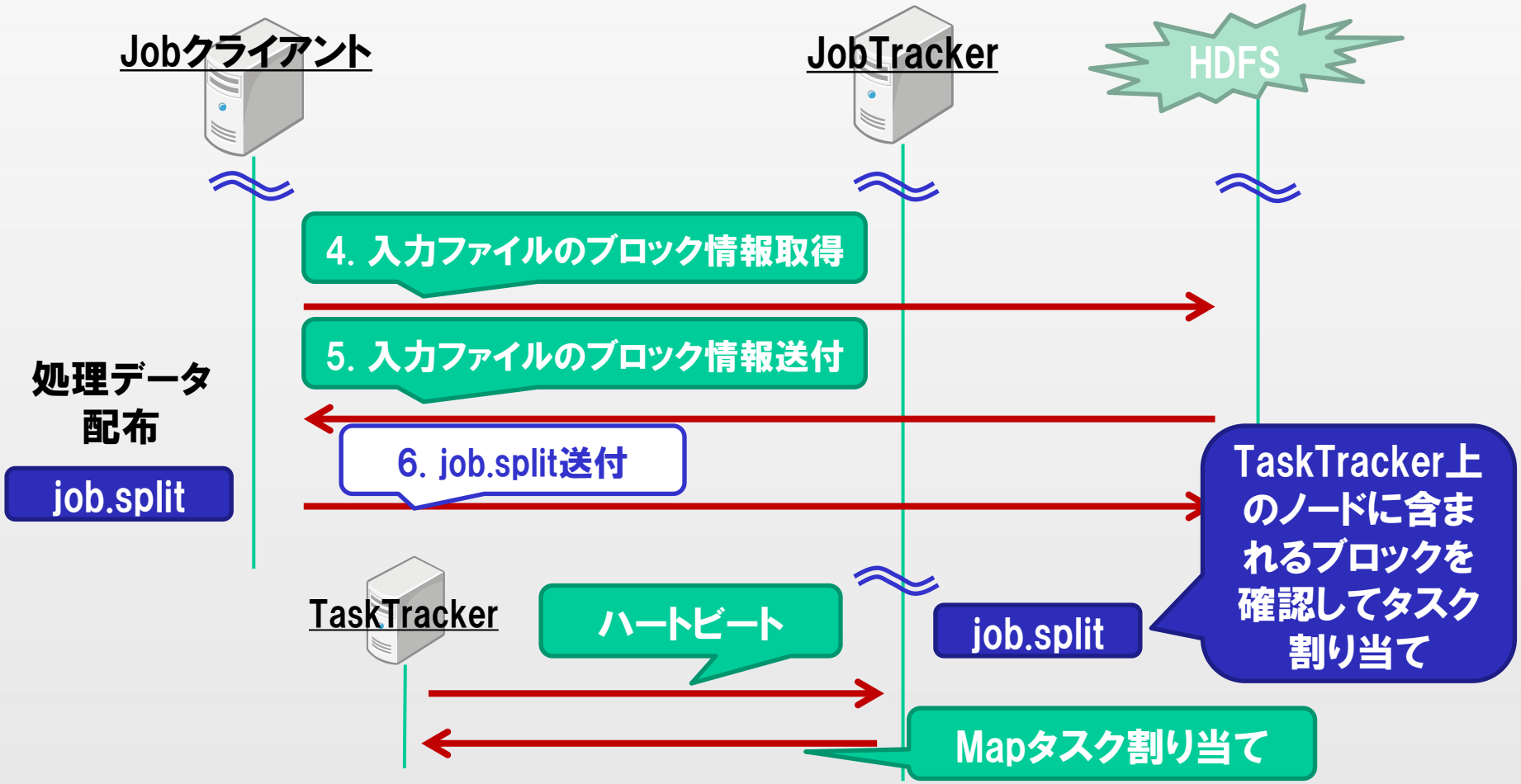
4. MapReduceに関するポイント

MapReduceに関するポイント

- データのローカリティを意識したMap処理の割り当て
- 中間データの扱い方
- 投機的実行

データのローカリティを意識したMap処理割り当て

- Hadoopは、”データが存在する場所で処理”が特長である
- これを実現する仕組みとして、Map処理の割り当て方法がある



データのローカリティに関するMap処理確認方法

- 処理に関するローカリティは、以下の手段で確認することができる

1. JobClientに出力されるメッセージでの確認

```
## データが自DataNodeに含まれるTaskTrackerでのMapタスク数
12/02/20 18:50:40 INFO mapred.JobClient:     Data-local map tasks=1
## データが自ラックのDataNodeに含まれるTaskTrackerでのMapタスク数
12/02/20 18:50:40 INFO mapred.JobClient:     Rack-local map tasks=1
```

2. MapReduceジョブのWebインタフェースでの確認

- MapReduceジョブ画面のカウンター情報にて、”Job Coutners”グループに”Data-local map tasks”や”Rack-local map tasks”カウンター情報が記録

中間データの扱い方

- Map処理結果となるデータは、中間データと呼ばれる
  - Map処理を実行したTaskTrackerのローカルディスクに出力される
  - 中間ファイルはHDFS上に出力されない
  - MapReduceジョブ完了後に、TaskTrackerにて自動的に消去される

- 中間データは、Hadoopのシリアライゼーション機能によって、シリアライズ化された状態で記録される
  - アプリケーション開発者は、シリアライズを意識した実装をする必要はない

- 中間データを出力するローカルディスクの容量が一杯で出力できない場合は、Map処理はFAILEDとして失敗する

中間データの確認方法

- MapReduceジョブを実行するときに中間データは以下のように確認できる
  1. JobClientに出力されるメッセージによる確認

```
## MapReduceジョブを実行した場合に確認できるメッセージ
# Map output bytes : Map処理結果である中間データの総サイズ
# Map output records : Map処理結果である中間データの件数
12/02/20 18:50:40 INFO mapred.JobClient: Map output bytes=125805034
12/02/20 18:50:40 INFO mapred.JobClient: Map output records=4850017
```

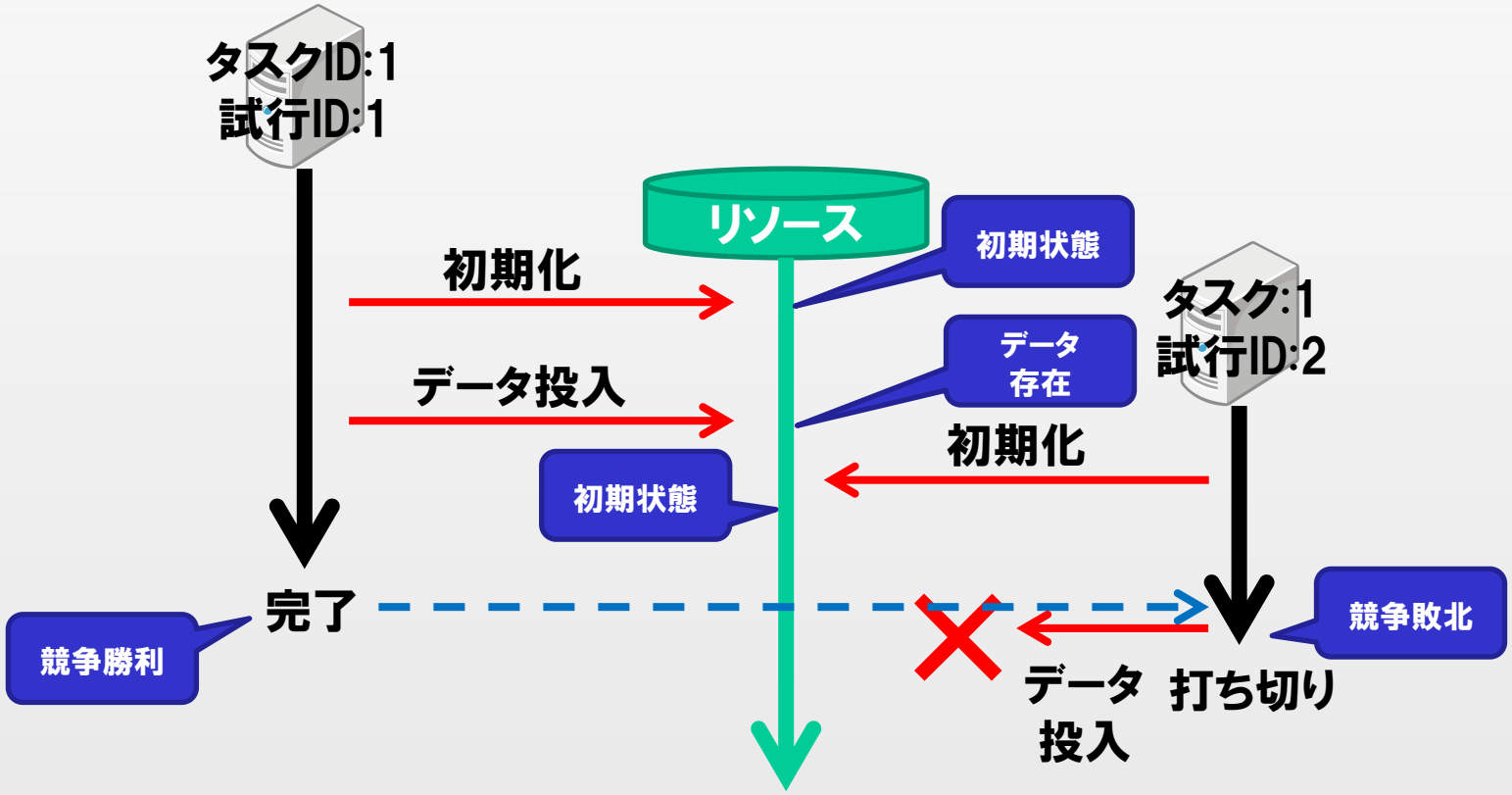
  2. JobTrackerでのMapReduceジョブに関するWebページにて確認
     - Map-Reduce Frameworkカウンターグループのカウンター値で確認できる
       - Map output bytes（Map列の値）
       - Map output records（Map列の値）
     - 合わせて、FileSystemCountersグループの次の情報も確認するとよい
       - FILE_BYTES_WRITTEN（Map列の値）
       - 中間データに加え、実際にTaskTrackerに出力したデータ量を示す

投機的実行の仕組み

- MapReduceジョブの進捗が進み終盤になるとTaskTrackerに空きスロットが発生することになる
- 空きリソースの有効活用として、処理が完了していないタスクと同じタスクを別のTaskTrackerに割り当てることがある
  - この挙動を**投機的実行**と呼ぶ
- 投機的実行により、処理を競争させてより早く処理が完了させることを目標とする
  - 競争に勝ったタスクを処理結果として扱う
  - 競争に負けたタスクはその瞬間に終了となる
- 投機的実行は、以下のプロパティによって実行するかどうか設定できる
  - Mapタスク： mapred.map.tasks.speculative.execution プロパティ
  - Reduceタスク： mapred.reduce.tasks.speculative.execution プロパティ
  - デフォルト値：どちらもtrue（投機的実行は有効）

投機的実行により陥りやすいポイント

- 処理内部で、他リソースへの操作や参照を実行する場合、同一タスクIDによる操作のため、リソース競合など異常を引き起こしやすい



リソースは初期化されたままとなる
（必要なデータは記録されない）

まとめ

本講義で学んだ内容

- HDFS詳細
  - HDFSメタデータ
  - HDFSチェックポイント
- HDFSに関するポイント
  - DataNode操作
  - HDFSの持つ機能
- MapReduce詳細
  - MapReduceジョブ実行フロー
  - Mapタスク/Reduceタスクの仕組み
  - タスクスケジューラ
- MapReduceに関するポイント
  - データのローカリティの考え方
  - 投機的実行