分散処理アプリ演習　第6回
MapReduceプログラミング応用

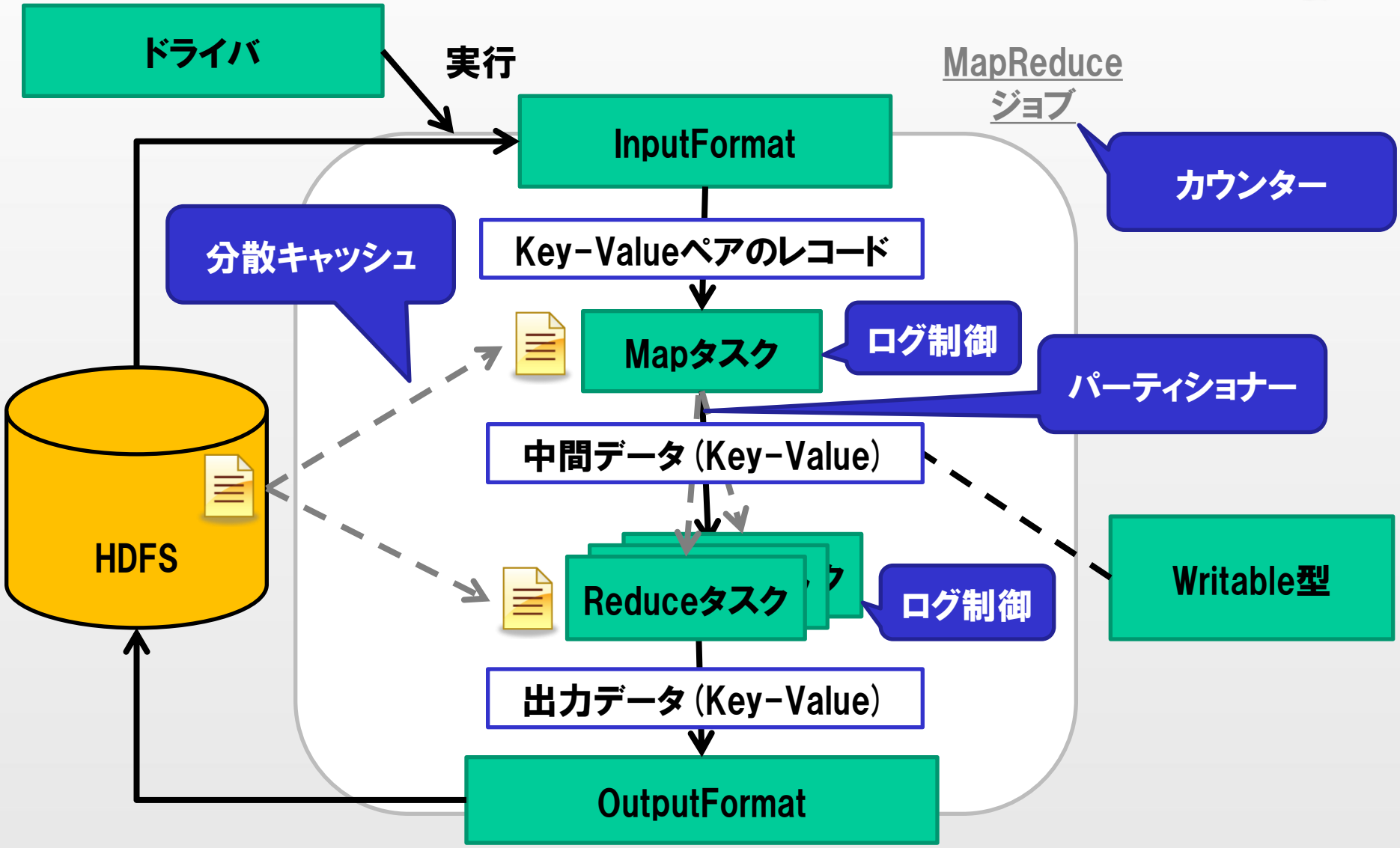
（株）NTTデータ

講義内容

1. MapReduceプログラミング応用のポイント
2. パーティショナー
3. 分散キャッシュ
4. カウンター
5. アプリケーションのログ制御
6. セカンダリーソート
7. 大量ファイルの扱い方
8. 演習：POSデータ分析アプリケーション開発

1. MapReduceプログラミング応用のポイント

MapReduceプログラミング応用ポイント



MapReduceプログラミング応用ポイント

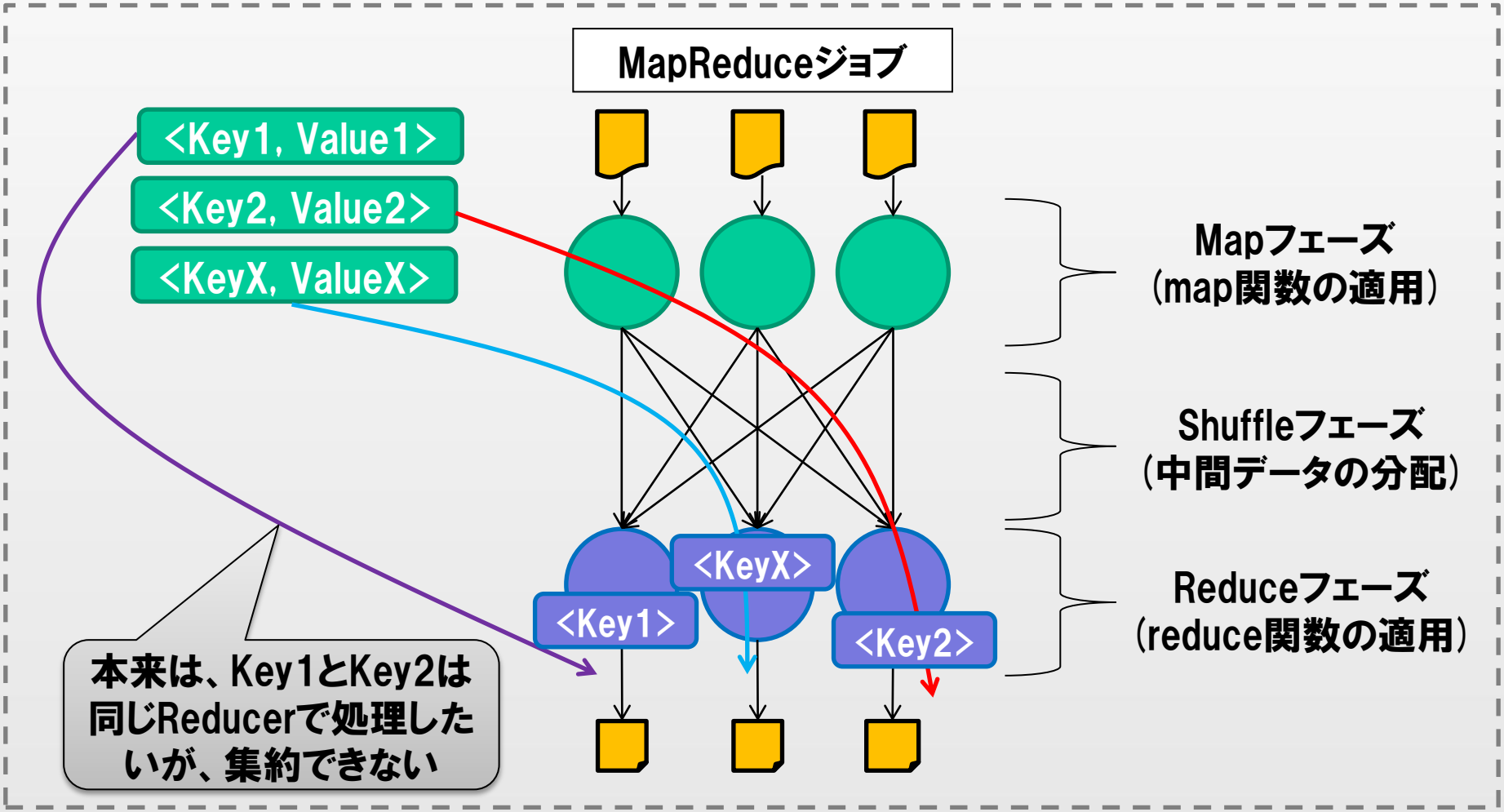
- MapReduceプログラミングでの応用ポイントは以下の通りである

項番	応用ポイント	概要
1	パーティショナー	Map処理結果をどのReduce処理と紐付けるか制御する方法
2	分散キャッシュ	MapReduce処理時に利用できるキャッシュを利用する方法
3	カウンター	MapReduceジョブの処理状況を確認する方法
4	アプリケーションログ制御	MapReduceアプリケーションを実行する場合に出力するログを制御する方法
5	セカンダリソート	Reduce処理時に、Keyだけでなく Valueも特定の条件で整列させる方法
6	大量ファイルの扱い方	MapReduceジョブで大量のファイルを扱う方法

2. パーティショナー

パーティショナーとは

- 特定の条件でデータを集めて処理したい場合に設定する
  - 標準のパーティショナーでは、意識的にデータを集約できない



【参考】Hadoop標準のパーティショナー

- HadoopはHashPartitionerによりKeyを振り分ける

```
public class HashPartitoner<K, V> extends Partitioner<K, V> {

  public int getPartition(K key, V, value,
                          int numReduceTasks) {
    return (key.hashCode() & Integer.MAX_VALUE) % numReduceTasks;
}
```

keyオブジェクトのハッシュを利用して
振り分け先を設定する

パーティショナーの設定

1. Partitionerインタフェースを実装
   - getPartitionメソッドを実装することで利用可能
2. Job.setPartitionerClassメソッドにて、利用するPartitionerクラスを定義

```
public class SampleJob extend Configured implements Tool {
  public int run(String[] args) throws Exception {
    Job job = new Job(getConf());
    // 途中省略
    job.setPartitionerClass(SamplePartitioner.class);
    // 途中省略
  }
}
```

利用するPartitionクラスを宣言する

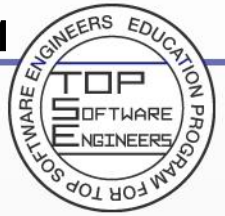
```
public class SamplePartitioner extends Partitioner<K, V> {

  // KEYの特定の要素 % Reduce処理数 を設定
  public int getPartition(K k, V v, int nReduces) {
    return k.getPosition() % nReduces;
  }
}
```

※ Reduce処理数より大きな値を設定するとデータを振り分けられなくなりFAILED扱いになるので注意

注意点：パーティショナー設定による影響

- Partitionerの設定によりデータが大きく偏る場合がある
  - 都道府県単位によるPartitioner
    - 東京、神奈川といったエリアのデータは大量、島根・鳥取はほとんど無い
  - 時間帯によるPartitoner
    - 日中帯のデータは大量、深夜のデータはほとんど無い
  - 国別によるPartitioner
    - 中国のデータは大量、バチカン市国のデータはほとんど無い

- 上記の事例では、他の要素を交えてパーティションするという工夫も必要である


TOP SOFTWARE ENGINEERS
EDUCATION PROGRAM FOR TOP SOFTWARE ENGINEERS

3. 分散キャッシュ

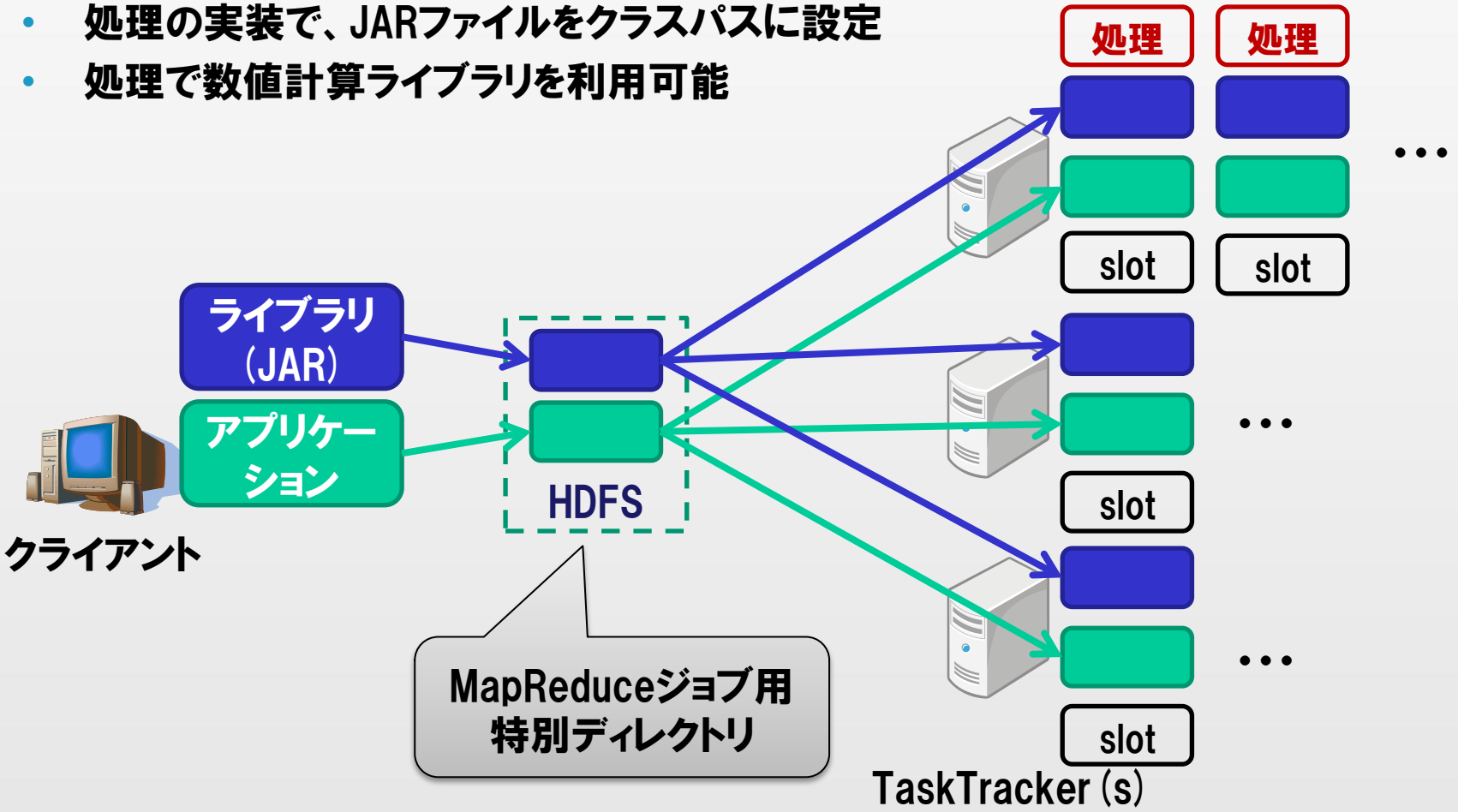
分散キャッシュとは

- 分散キャッシュ（DistributedCache）：Hadoop標準の分散キャッシュ機能
  - マスターデータのようなクライアントに保存してある小さなサイズのファイルを各処理に配布する仕組み
    - マスターデータとの結合のような処理を実装する場合に利用する
  - クライアントに配置している個別に作成したアーカイブやライブラリを処理で利用するために各TaskTrackerに配布する仕組み

- 利用シーンを次ページ以降で述べる

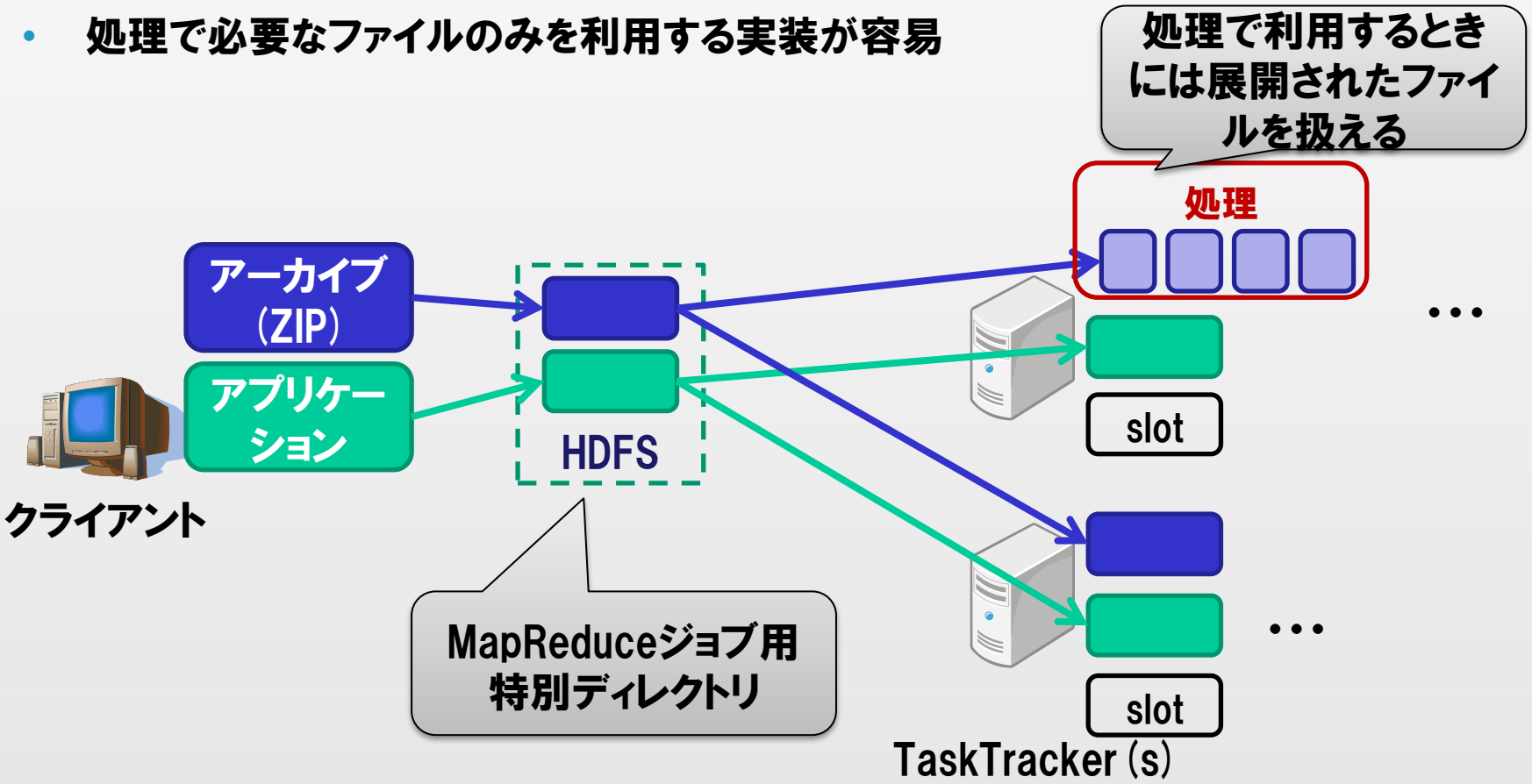
分散キャッシュの利用シーン

1. MapReduce処理で数値計算用ライブラリ(JARファイル)を利用したいが、Hadoopクラスタ自身を操作する権限は無い
   - アプリケーション実行時にDistributedCacheにてJARファイルを展開
   - 処理の実装で、JARファイルをクラスパスに設定
   - 処理で数値計算ライブラリを利用可能



分散キャッシュの利用シーン

2. マスターデータ（サイズ:小、ファイル数:多数）がZIPファイルとして用意されているが、このマスターデータを簡単に利用したい
   - サイズ小のファイルをHDFSに保存 → 効率悪いリソース利用方法
   - DistributedCacheの機能により、ZIPファイルはTaskTrackerで展開可能
   - 処理で必要なファイルのみを利用する実装が容易



分散キャッシュの利用方法

- API：DistributedCacheクラスを利用する
  - 1. MapReduceジョブ設定クラス （Jobクラスでの設定）

```
public class SampleJob extend Configured implements Tool {
  private static String linkName = "link";

  public int run(String[] args) throws Exception {
    Configuration config = getConf();
    Path cacheFile = new Path(args[0]);
    // 事前にcacheFileをHDFSを利用して格納する
    FileSystem fs = FileSystem.get(config);
    if (!fs.exists(cacheFile)) {
      fs.copyFromLocalFile(cacheFile, cacheFile);
    }
    // DistributedCacheの設定
    URI uri = new URI(cacheFile + "#"+ linkName);
    DistributedCache.addCacheFile(uri, config);
    DistributedCache.createSymlink(config);
    // 以下省略
  }
}
```

分散キャッシュの利用方法

- API：DistributedCacheクラスを利用する
  - 2. Map処理（Reduce処理）での設定

```
public class SampleMap extends Mapper<K, V, K, V> {
  private static HashMap<X, Y> m = new HashMap<X, Y>();
  private static String linkName = "link";
  public void setup(Context c)
    throws IOException, InterruptedException {
    // DistributedCacheで配布したファイルを読み込む
    Scanner s = new Scanner(new File(linkName));
    s.useDelimiter("¥n");
    // ファイル内のデータをHashMapに格納する
    while (s.hasNext()) {
      String str = s.next();
      m.put(X.getX(str), Y.getY(str));
    }
  }
  public void map(K k, V v, Context c)
    throws IOException, InterruptedException {
    // HashMap mの値を利用してMap処理を実行する
  }
}
```

分散キャッシュの利用方法

- hadoopコマンドにて、-files , -libjars , -archives をアプリケーションの実行と一緒に指定する
  - -files <カンマ区切りでパスを指定>
    - DistributedCacheにより各スレーブノードにファイルを展開
  - -libjars <カンマ区切りでパスを指定>
    - 各タスク実行時のクラスパスに指定したJARファイルも組み込む
  - -archives <カンマ区切りでパスを指定>
    - タスク実行時に、アーカイブファイルを展開して各スレーブノードにファイルを展開
    - アーカイブ対象は、zip、tar、tgz、tar.gzが利用できる

```
## sample.txt をMapReduceアプリケーション (sample.jar) で利用する方法
$ hadoop jar sample.jar -files sample.txt input output

## ライブラリ mylibrary.jar をMapReduceアプリケーションで利用する方法
$ hadoop jar sample.jar -libjars mylibrary.jar input output

## sample.tgz をMapReduceアプリケーションで利用する方法
$ hadoop jar sample.jar -archives sample.tgz input output
```

注意点：分散キャッシュの不適切な利用方法

- 処理を実行するChildプロセスのヒープメモリに展開できないサイズのマスターデータを利用する場合
  - ChildプロセスでOutOfMemoryErrorが発生
    - 必要なデータのみをメモリに展開するなどAPレベルでの制御が必要
    - ヒープメモリサイズを拡大して利用
    - 結合方法をReduceでのジョインに変更

4. カウンター

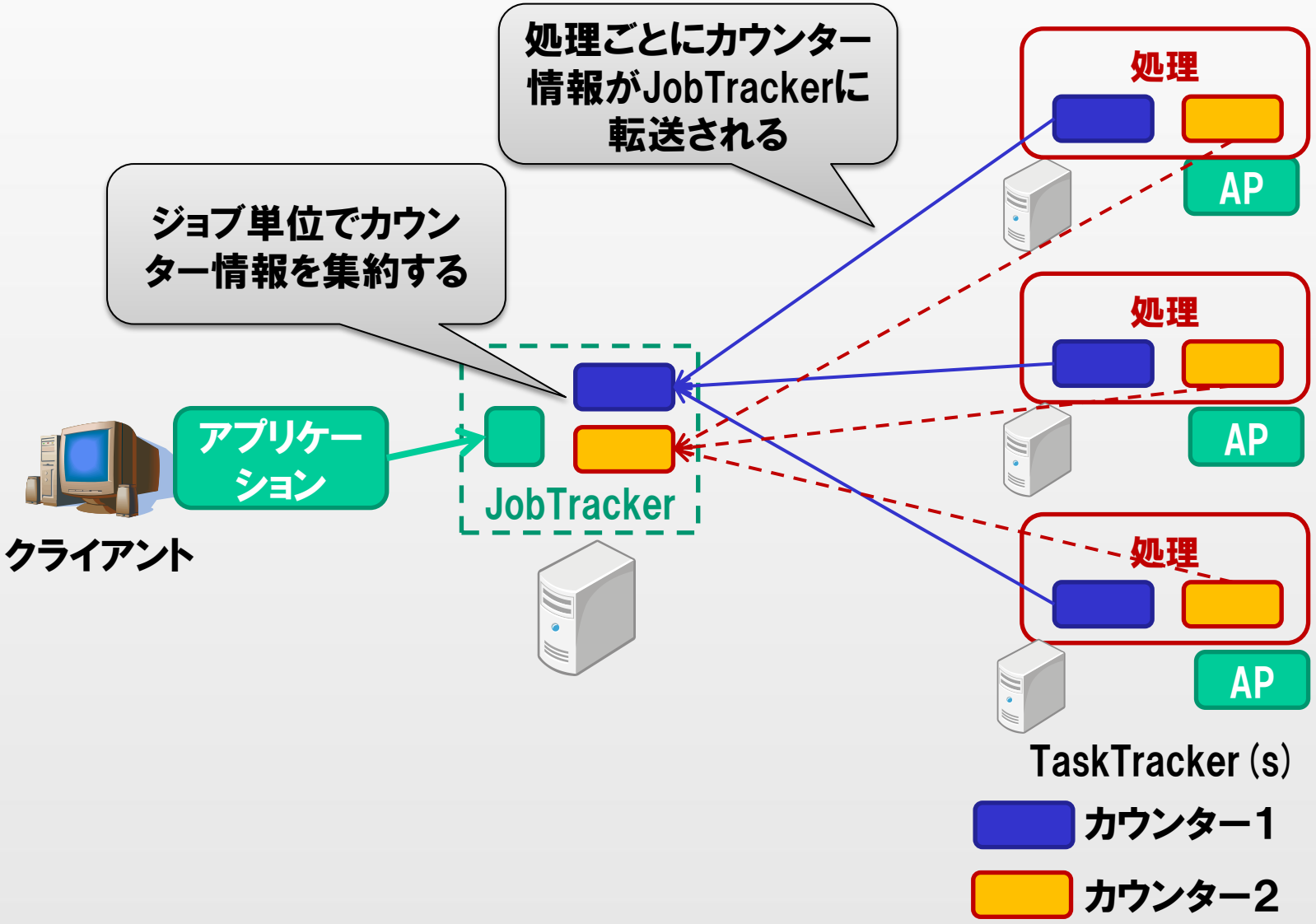
カウンターとは

- MapReduceジョブを実行したときに得られる統計情報
- long型の整数値で情報が記録される

- カウンターの構成は、以下の通りである
  - カウンター
    - 値が記録される
  - カウンターグループ
    - 複数のカウンターを特定の内容で集約したもの

- カウンター情報は、TaskTracker-JobTracker間のハートビートよって転送される
- カウンター情報は、TaskTrackerでの1つのMap処理/Reduce処理ごとに記録され、JobTrackerでMapReduceジョブ単位に集約される

カウンター情報の流れ


処理ごとにカウンター情報がJobTrackerに転送される
ジョブ単位でカウンター情報を集約する
処理
AP
処理
AP
処理
AP
アプリケーション
JobTracker
クライアント
TaskTracker (s)
カウンター1
カウンター2

Hadoop標準のカウンター

- Hadoop標準カウンターは以下の通りである
  - Job Counters：MapReduceジョブに関するカウンター
    - Map処理の起動方法(Data-local, Rack-localなど)
    - Map処理やReduce処理にスロットを利用した時間
  - FileSystemCounters：ファイル操作に関するカウンター
    - HDFS入出力データ量
    - TaskTrackerのローカルディスク操作に関する入出力データ量
  - Map-Reduce Framework ： MapReduceジョブとしてのカウンター
    - Map/Reduceでの入出力レコード数
    - Map/Reduceでの入出力データ量
    - Map処理、Reduce処理中にメモリ->ディスクへ書き出したレコード数

カウンターの値確認方法

- カウンター情報は以下の方法で確認できる
  1. JobTrackerのWebページより、各MapReduceジョブの情報を参照する
     - ジョブ単位の画面で、JobTrackerで集約されたカウンター情報を確認できる
     - タスク単位の画面で、タスクごとのカウンター情報も確認できる
  2. JobTrackerをログディレクトリに記録されるMapReduceジョブ実行履歴を参照する
     - JobTrackerの${HADOOP_LOG_DIR}/history/ 以下に記録される
  3. hadoop job -counter コマンドを実行する
     - hadoop job -counter <MapReduceジョブID> <カウンターグループ> <カウンター名>
  4. Counter用APIを利用する
     - org.apache.hadoop.mapred.Counters クラスを利用する

個別に利用するカウンター

- MapReduceアプリケーションにカウンター情報を埋め込むことも可能である

```
public class SampleMapper extend Mapper<K, V, K, V> {

  public void map(K key, V value, Context ctxt) {
    // 途中省略
    Counter counter = ctxt.getCounter(XCounter.RECORD);
    counter.increment(1);
    // 以下省略
  }

  public enum XCounter {
    RECORD,
  }
}
```

Counter情報を探す。存在しない場合は新規にカウンターとして登録される（初期値=0）

incrementメソッドの引数の値の分カウントされる

Counterで利用できる属性は、enumにて定義されていること

個別に利用するカウンター

- クライアントからカウンター情報を取得することも可能である

```
public class SampleJob extend Configured implements Tool {

  public int run(String[] args) throws Exception {
    Job job = new Job(getConf());
    // 途中省略・MapReduceジョブ実行
    job.waitForCompletion(true);
    // カウンター情報の取得
    long c = job.getCounters().
                findCounter(XCounter.RECORD).getValue();
    // 以下省略
  }

}
```

戻り値はlong型である

Jobクラスのインスタンスにて
カウンター情報を取得できる。
MapReduceジョブ実行前には
nullとなる

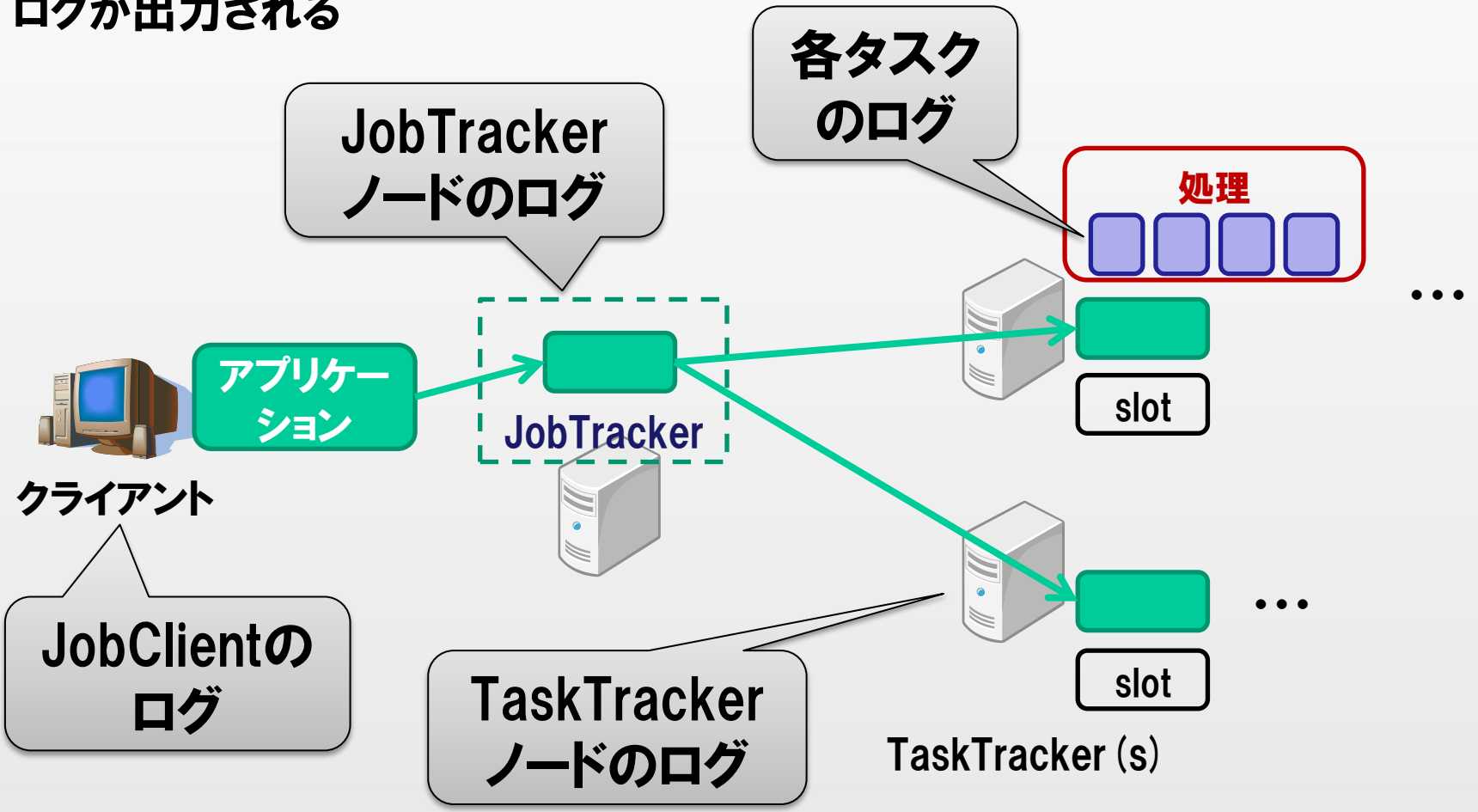
注意点：カウンターの扱い方

- 個別カウンターを利用する場合は、以下の点に注意する
  1. カウンターには、long型のみ設定可能である
     - 小数は利用不可
     - 負数も扱わない（incrementメソッドにて設定はできるが）
  2. 各タスク間でカウンター情報を共有するような実装はしない
     - カウンター値の変化を考慮したアプリケーションの実装は困難
     - カウンター情報取得に関して、JobTracker側でボトルネックになる

5. アプリケーションのログ制御

MapReduceジョブ実行でのログ出力

- Hadoopの各ノードの動作やMapReduceアプリケーションの実行では、さまざまなログが出力される



※ HDFSでは、NameNode、DataNode、DFSClientのログが出力される

MapReduceジョブ実行時のログ出力箇所

- JobClient（コマンドラインにてジョブを実行するサーバ）
  - MapReduceジョブ設定、進捗状況を出力
  - 通常は、画面上に表示
  - 各タスクのステータスに応じて、エラー情報を表示
- TaskTracker
  - Map処理/Reduce処理内に設定したログを記録
  - Task試行IDごとにログ領域が生成
  - 以下の3つのファイル内に記録
    - stdout：標準出力（System.out.printXXX）にて出力した情報を記録
    - stderr：標準エラー出力（System.err.printXXX）にて出力した情報を記録
    - syslog：Log4JやLOGクラス経由で出力した情報を記録
- JobTracker
  - タスクのステータスに応じて、エラー情報をJobTrackerログに記録

MapReduceジョブに関するログの確認

1. JobTrackerのWeb画面より各MapReduceジョブの情報をたどり確認
   - デフォルト：http://<JobTrackerのアドレス>:50030/

2. JobTracker/TaskTrackerのログディレクトリを確認
   - /var/log/hadoop/ 以下にログファイルが出力される

3. hadoop job -events コマンドを実行して確認

ジョブID job_201201010000_0001 の 10つの情報を確認する場合
```
$ hadoop job -events job_201201010000_0001 1 10
SUCCEEDED attempt_201201010000_0001_m_000001_0 http://<処理を実行したTaskTrackerのアドレス> :50060/tasklog?(以下省略)
・・・
```

→ 指定したタスクのログの出力先のHTTPアドレスを出力する

注意点：不適切なログ出力設定

- 入力されるレコードごとにINFOレベルで出力するような設定は×
  - ログ領域のディスク容量が一杯になると、MapReduceジョブは起動しなくなる
    - FAILED扱いとなり全てのジョブは停止（よくはまりやすいポイント）

```
public class SampleMapper extend Mapper<K, V, K, V> {

  public void map(K key, V value, Context ctxt) {
    // 途中省略

    // mapメソッド内でSystem.out.printlnを実行
    // 入力レコード分System.out.printlnが実行される
    System.out.println(value.toString());
    System.out.println(hogehoge.toString());

    // 以下省略
  }

}
```

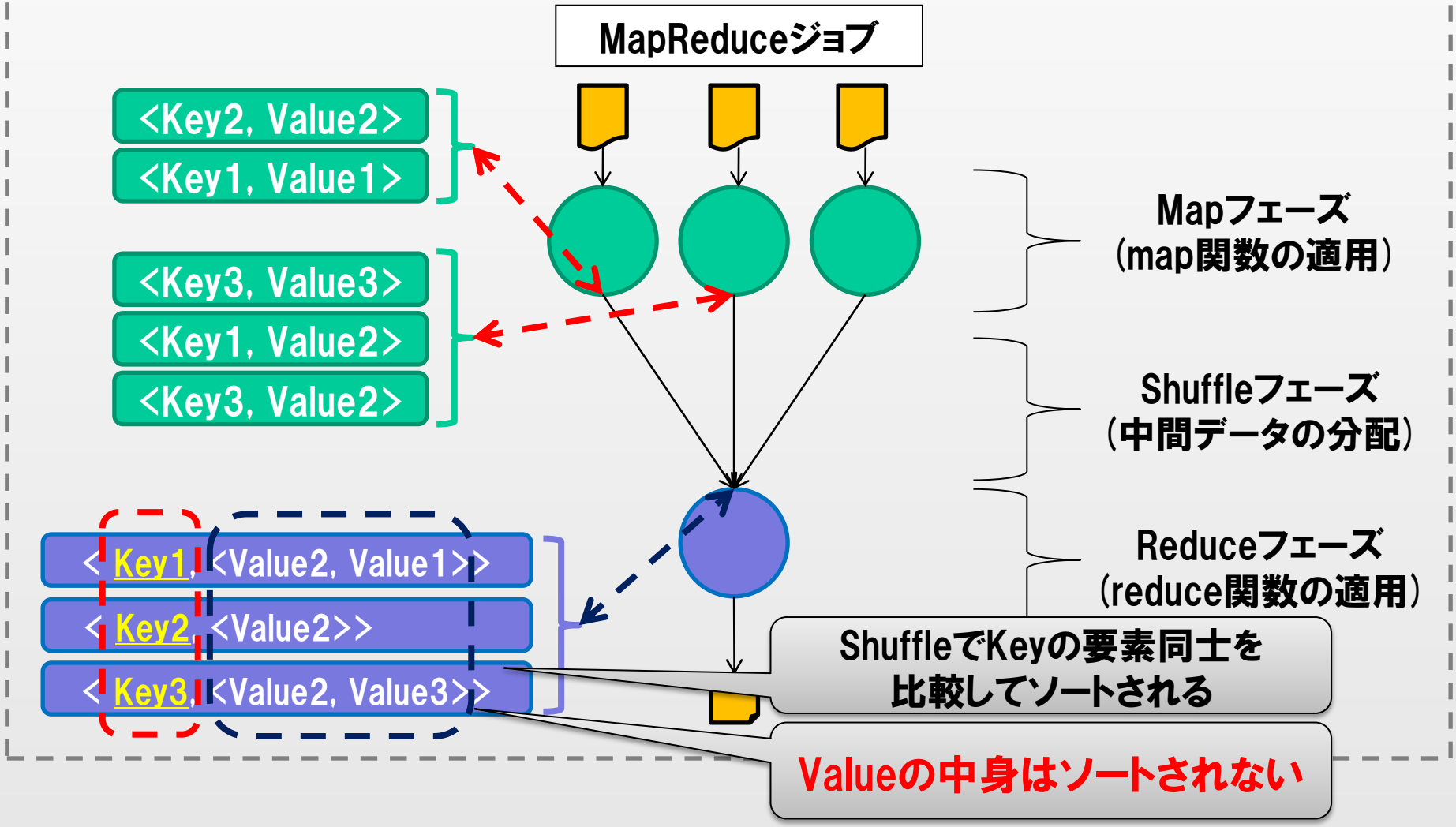
大量レコードが存在する場合、処理を実行する
TaskTrackerのログ領域を圧迫する
※ ログ領域が一杯になると処理が実行できない


EDUCATION PROGRAM FOR TOP SOFTWARE ENGINEERS
TOP
SOFTWARE
ENGINEERS

6. セカンダリソート

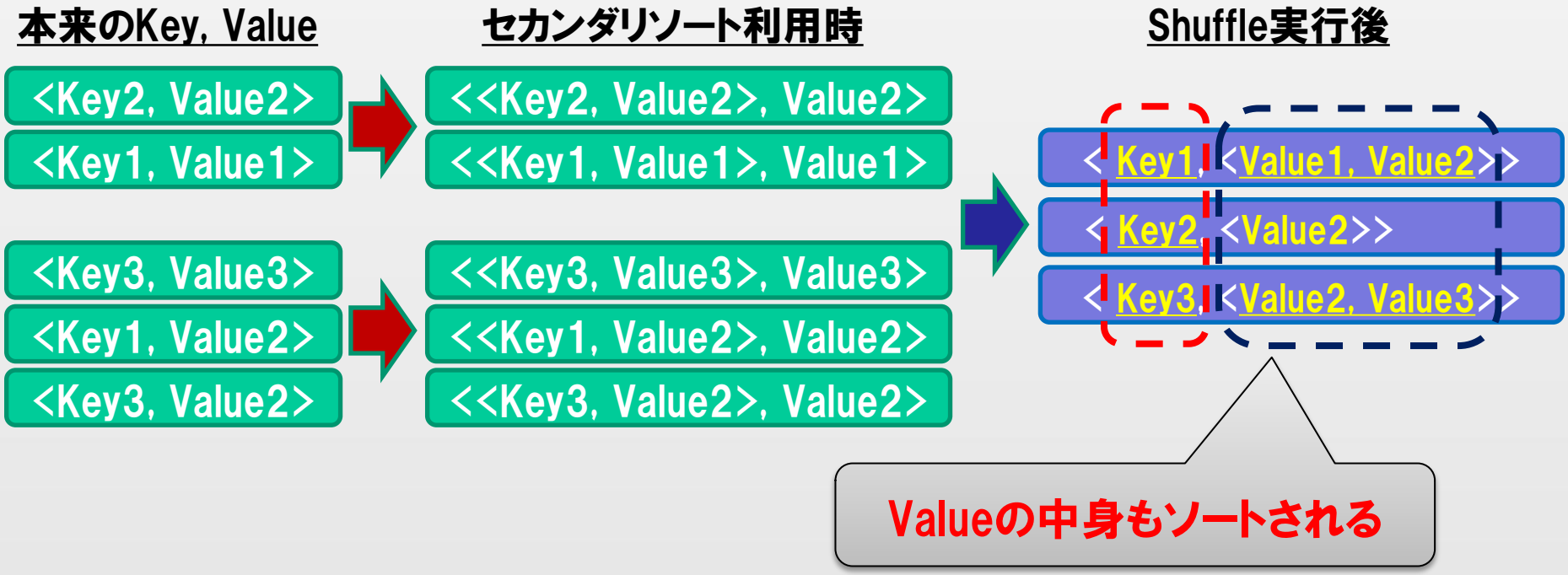
MapReduceの持つソート機能について

- Hadoop MapReduceフレームワークで、自動的にソートされるものは以下の通り



セカンダリソートの役割

- Reduceで処理するデータでValueにて以下のような処理を実施する場合は、事前にソートされているほうが望ましい
  - Valueの最小値（最大値）のみを抽出する
  - 特定の条件に含まれているValueのみを処理したい
  - 出力結果が特定の要素でソートされている
- HadoopにてValueをソートする手法として、セカンダリソートを利用する



セカンダリソートの実現方法

1. 従来のKeyに加え、ソート対象となるValueを組み合わせた複合キーを用意する
   - 複合キーでのcompareToメソッドにて、ソート対象Valueを並び替える実装を記述する
2. パーティショナーにてKeyのみ利用するgetPartitionメソッドを用意する
   - 標準のパーティショナーでは、複合キーのすべてを利用するため
3. キー比較用Comparatorクラスにて、複合キーのKeyのみを扱うcompareメソッドを用意する
4. Value比較用Comparatorクラスを用意する
5. Job.setPartitionerメソッドにて、2.で定義したPartitionerクラスを宣言する
6. Job.setGroupingComparatorClassメソッドにて、3.で定義したComparatorクラスを宣言する

注意点：セカンダリソートの利用について

- セカンダリソートで扱うComparatorの実装方法について注意する
  - もともと、HadoopのKeyのソートはシリアライズされているデータ同士を比較
    - シリアライズ-デシリアライズの手間を削減
  - compareToメソッドとcompareを組み合わせる仕組みが重要

```
// Hadoop標準のLongWritableクラスのComparatorを参照する
public class LogWritable implements WritableComparable {
  // 途中省略
  public static class Comparator extends WritableComparator {
    public Comparator() {
      super(LongWritable.class);
    }
    public int compare(byte[] b1, int s1, int l1,
                       byte[] b2, int s2, int l2) {
      long thisValue = readLong(b1, s1);
      long thatValue = readLong(b2, s2);
      return (thisValue<thatValue ? -1 :
                (thisValue==thatValue ? 0 : 1));
    }
  }  // 以下省略。readLongメソッドはWritableComparatorクラスのメソッドである
}
```

7. 大量ファイルの扱い方

大量のファイル操作が引き起こす影響

- Hadoopの処理（タスク）数は、以下の条件によって決まる
  - Map処理数 = MapReduceで利用するファイルのブロック数
  - Reduce処理数 = 各自で定義

- Map処理数は以下のように算出できる（ブロックサイズ=64MB）
  - 1MBの200個のファイル -> Map処理数：200
  - 200MBの1個のファイル -> Map処理数：4（4つのブロック数）

- HadoopクラスタのMapスロット数が少ない場合、200個のファイルを処理するために、そのつどMap用Javaプロセス起動やファイル読み込みが発生し、効率が悪い
  - ファイル数が増えるほどMap処理数が増え、処理時間が長期化する

- HDFS上でのファイル管理方法には注意が必要である

Hadoopでの大量ファイル利用方法 1

- 大量ファイルをMapReduceで扱う対策：その1
- HDFS上に格納する前に、ファイル自体をマージして格納する

- ファイルフォーマットや格納日付が同じであるような場合、ファイルをマージしながらHDFSに格納する方法がある
  - catコマンドとHDFS putコマンドの利用

```
## ローカルのファイルの確認
$ ls
sampleA.txt sampleB.txt sampleC.txt
## HDFS上にsample.txtと1つのファイルに集約
$ cat *.txt | hadoop fs -put - sample.txt
## 確認
$ hadoop fs -ls sample.txt
```

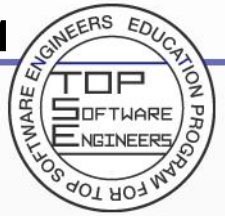
  - 複数のファイル格納用アプリケーションの作成

Hadoopでの大量ファイル利用方法 2

- 大量ファイルをMapReduceで扱う対策：その2
  - HadoopのMapReduce処理では、通常は1つのファイル＝1つ以上のMap処理であるが、2つ以上のファイルを1つのMap処理で割り当てることも可能である

- 手法として、CombineFileInputFormatを利用する

- HadoopのMapReduceフレームワークには、CombineFileInputFormat抽象クラスが用意されている
- CombineFileInputFormatを実装することで、複数ファイルを1つのMap処理で扱うことができるようになる


SOFTWARE ENGINEERS EDUCATION PROGRAM FOR TOP
TOP
SOFTWARE
ENGINEERS

8. 演習

演習テーマ

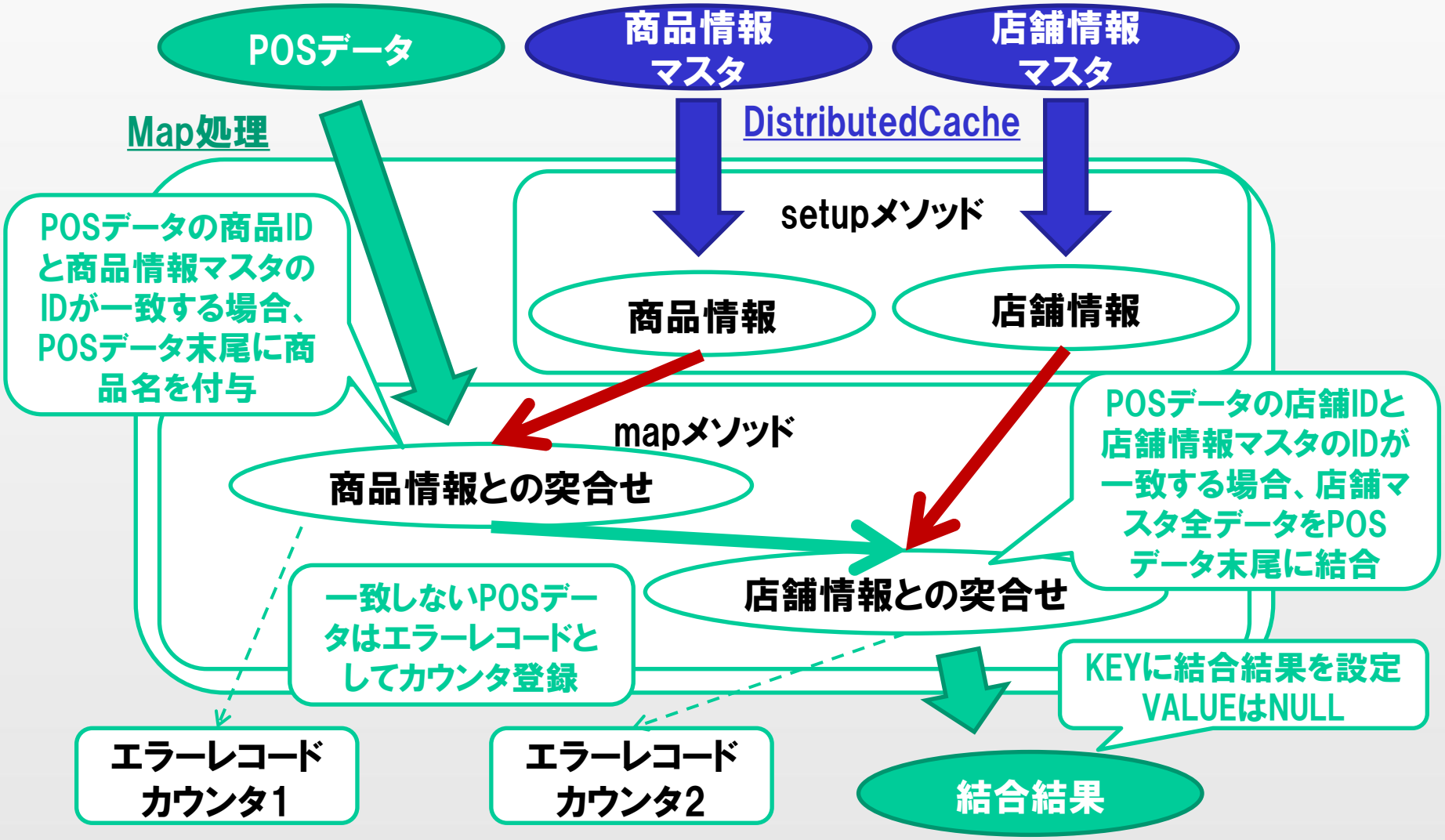
- 本講義では、以下の2つのMapReduceジョブについて演習する

- 演習1：POSデータとマスターデータの結合処理
  - 課題1：商品情報マスタのDistributedCacheの設定
  - 課題2：商品情報マスタから商品名を抽出
  - 課題3：商品情報マスタの商品IDとレコード内の商品IDの結合処理

- 演習2：POSデータ集計処理
  - 課題4：Partitionルールの実装

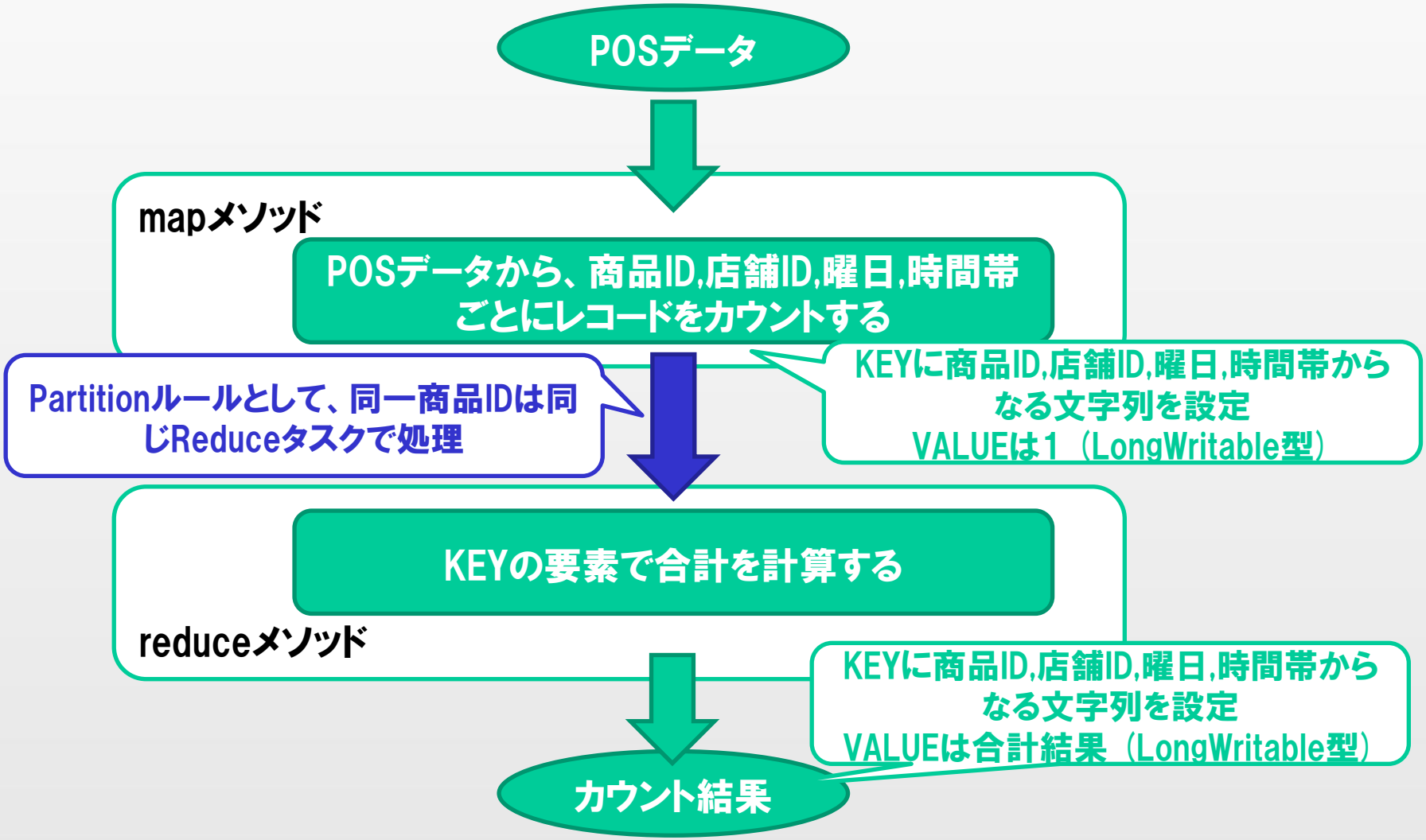
演習1: POSデータとマスターデータの結合処理

- POSデータとマスターデータの結合処理の流れを以下に示す



演習2: POSデータ集計処理

- POSデータ集計処理の流れを以下に示す



演習環境 - データ類

- 演習で利用する資材は、以下の通り配置されている
  - /root/hadoop_exercise/06/ （ローカル環境）以下にデータは格納されている

- POSデータ：posdata.tar.gz（pos-dataディレクトリに格納）
  - POSデータは、以下のコマンドを実行してHDFS上に配置すること

```
## POSデータを展開する
$ cd /root/hadoop_exercise/06/pos-data/
$ tar xvzf posdata.tar.gz
## POSデータをHDFS上に格納する
$ hadoop fs -mkdir hadoop_exercise/06/pos-data
$ hadoop fs -put posdata.csv hadoop_exercise/06/pos-data/
## POSデータをHDFS上に配置されているか確認する
$ hadoop fs -ls hadoop_exercise/06/pos-data/
```

- 商品情報マスタデータ：goodsmaster.csv（master-dataディレクトリに格納）
- 店舗情報マスタデータ：storemaster.csv（master-dataディレクトリに格納）

演習環境 - ソースコード

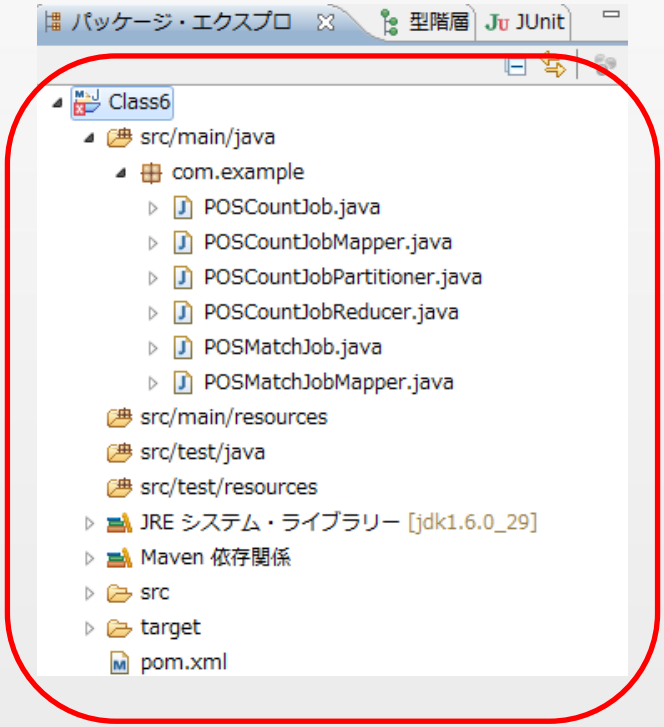
- 演習で利用する資材は、以下の通り配置されている
  - /root/workspace/Class6/ 以下にデータは格納されている

- POSデータとマスターデータの結合処理用クラス（赤字は実装対象クラス）
  - POSMatchJob：POSデータ結合処理MapReduceジョブクラス
  - POSMatchJobMapper：POSデータ結合処理Map処理クラス

- POSデータ集計処理用クラス
  - POSCountJob：POSデータ集計処理MapReduceジョブクラス
  - POSCountJobMapper：POSデータ集計処理Map処理クラス
  - POSCountJobPartitioner：POSデータ集計処理Partitionerクラス
  - POSCountJobReducer：POSデータ集計処理Reducerクラス

演習環境 – Eclipseプロジェクト

- EclipseのClass6プロジェクト内に、必要な資材がそろっている

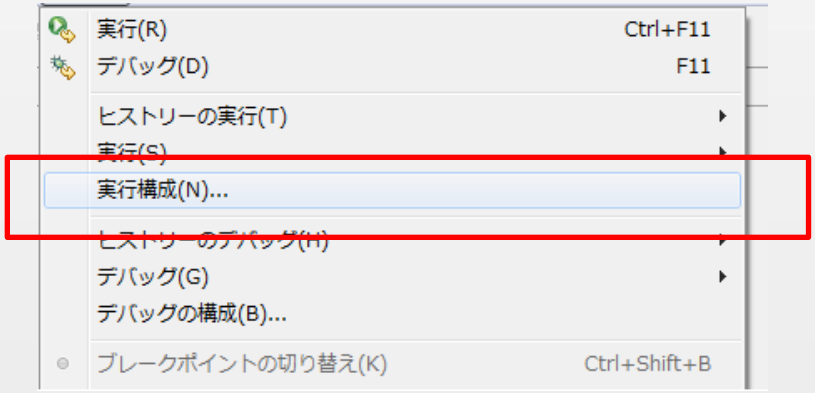


「Class6」プロジェクト内の「src/main/java」ディレクトリ内に演習対象のソースコードが格納されている

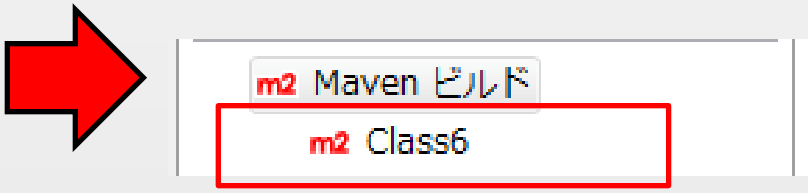
実行用JARファイル作成方法

- 作成したMapReduceジョブをコンパイルし、class6-exercise-0.1.jarという名前でjarパッケージにまとめる

Eclipseのメニューから
[実行]→[実行構成]を選択



出現したダイアログボックスの左側にある「Maven ビルド」から、「Class6」を選択し、ダブルクリック



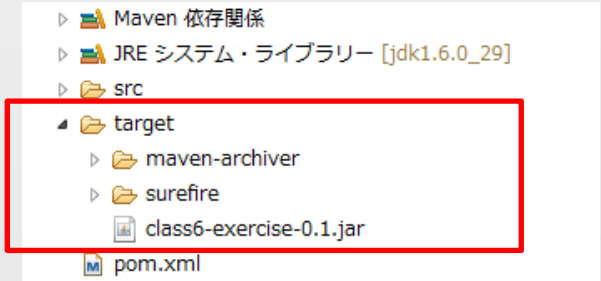
※ Eclipse上での操作である

実行用JARファイル作成方法（続き）

- コンパイル/パッケージングが始まる。ここでコンパイルエラーなどがある場合はEclipseのコンソールに表示される

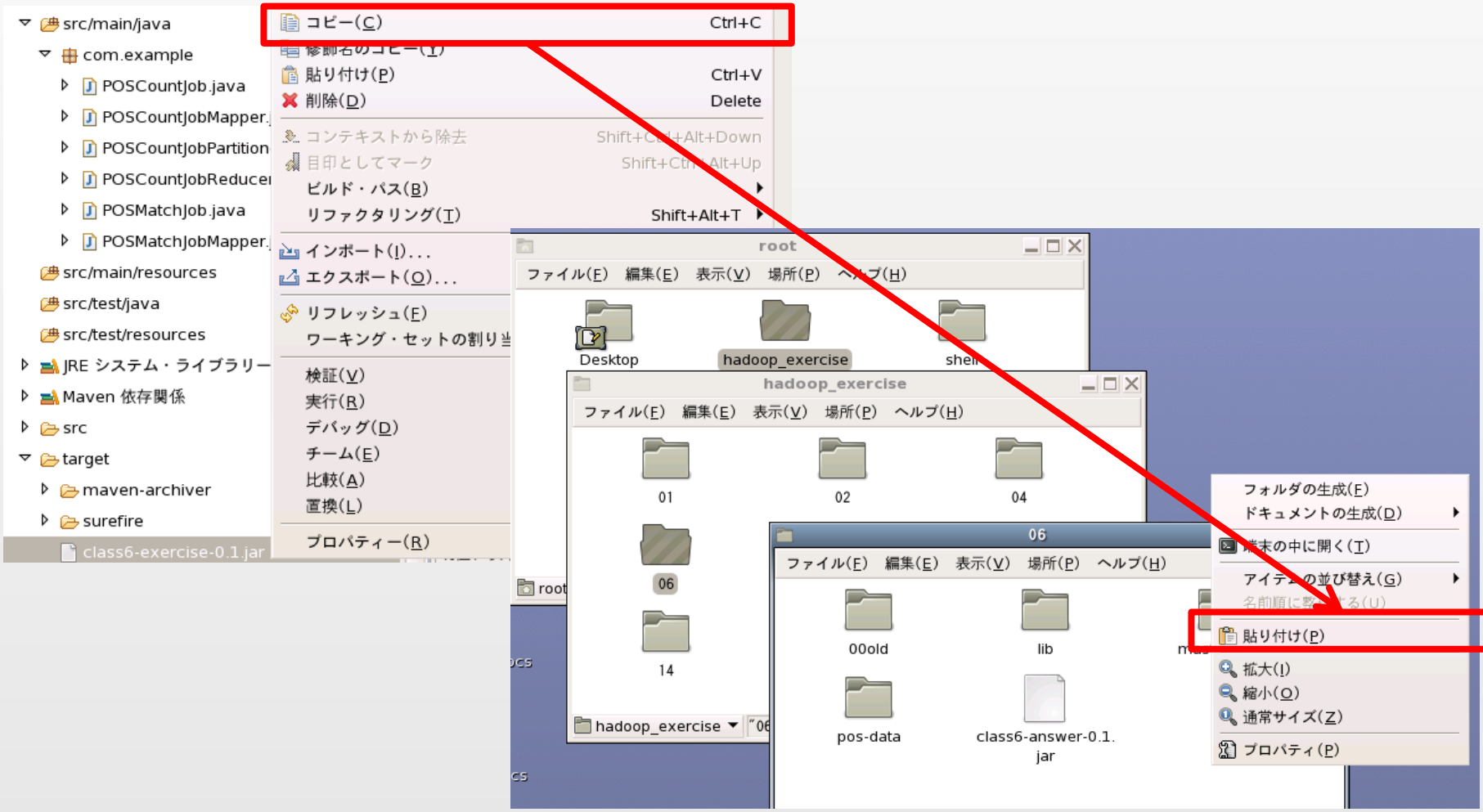
```
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time: 2.109s
[INFO] Finished at: Mon Mar 05 14:20:34 JST 2012
[INFO] Final Memory: 7M/17M
[INFO] ------------------------------------------------------------------------
```

- ビルド成功すると、「Class6」プロジェクト内の「target」ディレクトリに、class6-exercise-0.1.jarが作成される



実行方法

- ビルドして生成されたJARファイルをVNC上の操作で、以下のディレクトリにコピーする（コピー先ディレクトリ：/root/hadoop_exercise/06/）



実行方法

- MapReduceアプリケーションは以下のように実行する

```
-- class6-exercise-0.1.jar を/root/hadoop_exercise/06/に
-- コピーした状態で利用するアプリケーションとする
-- POSデータ結合処理を実行する場合
$ cd /root/hadoop_exercise/06/
$ hadoop jar class6-exercise-0.1.jar ¥
 com.example.POSMatchJob ¥
 hadoop_exercise/06/pos-data ¥
 hadoop_exercise/06/match ¥
 master-data/goodsmaster.csv ¥
 master-data/storemaster.csv

-- POSデータ集計処理を実行する場合
$ cd /root/hadoop_exercise/06/
$ hadoop jar class6-exercise-0.1.jar ¥
 com.example.POSCountJob ¥
 hadoop_exercise/06/pos-data ¥
 hadoop_exercise/06/count
```

確認方法

- MapReduceジョブ実行結果は、以下のコマンドで確認する

```
## POSデータ結合処理結果をlsコマンドにて確認する
$ hadoop fs -ls hadoop_exercise/06/match

## POSデータ結合処理結果の中身をtailコマンドで確認する
## 処理結果ファイル(part-m-00000)の中身を確認する
$ hadoop fs -tail hadoop_exercise/06/match/part-m-00000
```

演習で扱うPOSデータ

- POSデータは、セッション状態によって記録される情報が異なる
  - フィールドはカンマ区切り，改行コードはLF，文字コードはUTF-8
- 共通で記録される情報は以下の通りである

項番	項目名	データ型	桁数	概要
1	取引ID	文字列	28	店舗全体での取引ID（登録日 + 登録時間 + 登録店舗ID + セッションID）
2	登録日	文字列	8	YYYYMMDDで登録
3	曜日	文字列	1	0: 日曜 ～ 6：土曜
4	登録時間	文字列	6	HHMMSSで登録
5	セッションID	文字列	6	1取引区分のID
6	商品コード	文字列	8	上位2桁：商品区分コード，下位6桁：商品番号
7	商品名	文字列	32	登録している文字列
8	金額	整数	-	商品販売価格
9	消費税額	整数	-	合計額 × 0.05（切捨て）
10	数量	整数	-	デフォルト1，複数一括で購入している場合は2以上

演習で扱うPOSデータ

- セッションの最後には共通情報の後ろに以下の情報が記録される

項番	項目名	データ型	桁数	概要
11	登録店舗ID	文字列	8	上位2桁：CS 直営店，FC：フランチャイズ店，SC：特別店 中位2桁：エリアコード 下位4桁：店舗ID
12	登録端末ID	文字列	4	数字4桁 0000 ～ 9999 まで。
13	購入者性別	文字列	1	1：男性，2：女性，3：不明
14	購入者年齢層	文字列	1	1： 10代以下，2：20代，3：30代，4：40代，5：50代，6：60代以上
15	担当者ID	文字列	8	店舗担当者のID
16	合計額	整数	-	セッションIDの金額の和
17	合計消費税含額	整数	-	合計額 × 0.05（切捨て）
18	支払方法	文字列	1	1：現金，2：電子マネー，3：クレジットカード で区分
19	電子マネーID	文字列	1	1～6まで
20	電子マネー番号	文字列	16	任意の16桁の数字で構成。
21	電子マネー承認番号	文字列	6	000000～999999で昇順に付与。登録店舗IDと登録日とカード電子マネー番号で特定が可能
22	カード会社ID	文字列	2	01 ～ 12 まで。
23	カード番号	文字列	16	最大16桁の数字で構成。
24	カード承認番号	文字列	6	000000～999999で昇順に付与。登録店舗IDと登録日とカード番号で特定が可能
25	イベントID	文字列	1	0：無し，1: スポーツ系，2：コンサート系，3：祭り系，4: 教育系，5：その他
26	備考	文字列	64	何も無い場合は---とする

演習で扱う商品情報マスタ

- 商品情報マスタのフォーマットは以下の通りである
  - フィールドはカンマ区切り，改行コードはLF，文字コードはUTF-8

項番	項目名	データ型	桁数	概要
1	商品コード	文字列	8	上位2桁：商品区分コード，下位6桁：商品番号 商品区分コード：10：飲料、20：パン、30：お菓子、40：お弁当類、50：加工食品、60：お土産
2	商品名	文字列	32	登録している文字列
3	会社区分	文字列	1	1：自社製造品，2：他社製造品
4	会社コード	文字列	8	他社製造品の場合は製造会社コードを付与
5	仕入れ価格	整数型	-	
6	販売額	整数型	-	
7	有効区分	文字列	1	0：無効、1：有効
8	登録日	文字列	8	YYYYMMDDで設定
9	失効日	文字列	8	YYYYMMDDで設定
10	備考	文字列	64	必要に応じて設定する

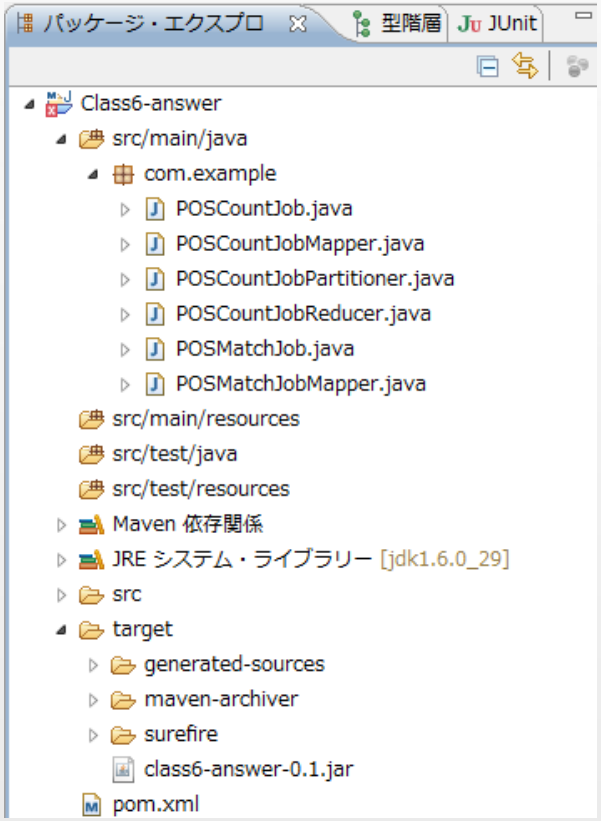
演習で扱う店舗情報マスタ

- 店舗情報マスタのフォーマットは以下の通りである
  - フィールドはカンマ区切り，改行コードはLF，文字コードはUTF-8

項番	項目名	データ型	桁数	概要
1	登録店舗ID	文字列	8	上位2桁：CS 直営店，FC：フランチャイズ店，SC：特別店 中位2桁：エリアコード 下位4桁：店舗ID
2	店舗名	文字列	64	店舗名
3	店舗形態	文字列	1	1：独立店舗、2：商業施設内店舗、3：交通機関内店舗、4：教育機関内店舗、5：その他
4	開店時間	文字列	4	HHMMで記入。24時間の場合0000を記入
5	閉店時間	文字列	4	HHMMで記入。24時間の場合2359を記入
6	店舗郵便番号	文字列	7	7桁
7	店舗住所	文字列	128	住所
8	店舗責任者ID	文字列	8	店長ID
9	店舗電話番号	整数型	11	IP電話
10	店舗FAX番号	整数型	10	電話
11	店舗アカウント名	文字列	12	最大12文字
12	開店日	文字列	8	YYYYMMDDで入力
13	閉店日	文字列	8	YYYYMMDDで入力
14	有効区分	文字列	1	0：無効、1：有効

回答例

- Eclipse上のClass6-answerプロジェクトに回答例を示す



まとめ

本講義で学んだ内容

- パーティショナー
- 分散キャッシュ
- カウンター
- アプリケーションのログ制御
- セカンダリーソート
- 大量ファイルの扱い方
- 演習：POSデータ分析アプリケーション開発