春の嵐吹く リアルタイム分散処理 Storm

日々発生する大量なデータをリアルタイムに処理し続ける「ストリームデータ処理」に対するニーズが高まっています。同じビッグデータでもバッチ処理のHadoopとはまた違った解決方法が求められる分野です。本記事ではそのストリームデータ処理を実現するプロダクトとして、今、注目を集めている「Storm」について解説します。

Acroquest Technology㈱
鈴木 貴典 SUZUKI Takanori

Twitter社が公開したオープンソース

「Storm」は、Twitter社が公開しているオープンソースプロダクトであり、耐障害性に優れたリアルタイムの並列分散処理を簡単に実現するためのフレームワークです。もともとは、Twitterのつぶやきを解析するシステムを開発していたBackType社のメンバが、ビッグデータをリアルタイムに処理するためのプラットフォームとして、Stormの開発を進めていました。そのBackType社を、Twitter社が2011年7月に買収し、その後、オープンソースとして公開されることになりました。

今回は、Stormとはどのようなものなのか、Stormを利用してどのようなことができるのか、その概要を紹介します。

ビッグデータ×リアルタイム=ストリームデータ処理

Stormで実現するリアルタイム処理は、より正確に言うと、「ストリームデータ処理」というものに該当します。「ストリームデータ処理」とは、連続的に発生するデータ(これを、「ストリームデータ」と言います)をリアルタイムに処理をし続けることです(**図1**)。Twitterのつぶやきがそれに該当しますが、ほかにもネットワークで発生するトラフィックデータ、工場などで利用されるセンサーが生み出すデータ、気候や株価の変動情報なども、ストリームデータに該当します。

近年、ネットワークの普及やクラウド化が進んできたことにより、社会全体で扱うデータ量が爆発的に増加しています。そのため、その大量データをいかにリアルタイムに処理するか、ということで、ストリームデータ処理に対するニーズも高まっています。

Stormで何ができるのか?

Stormが生まれた背景

ビッグデータに対する処理と言えば、Hadoopが有名でしょう。Hadoopの登場によりビッグデータという言葉が広まった、といっても過言ではありません。しかしながら、Hadoopが対象とする分野は、基本的にはバッチ処理です。そのため、リアルタイムというニーズにはマッチングしにくい状況でした。

一方、従来のシステムでは、リアルタイムで分散処理を行う場合、MOM(Message Oriented Middleware)のようなメッセージング技術を利用して、メッセージのキューイングと非同期の処理を組み合わせて実現することが一般的でした。しかしながら、従来の技術では信頼性や拡張性を満たすためには、苦労する部分も多くありました。

Stormはそれらの課題を解決するために登場しました。Stormを利用することで、多大な苦労なく、大規模なリアルタイム分散システムを構築することが可能となります。

6つの特徴

Stormは次に示す6つの特徴があります(**図2**)。

①シンプルなAPI

StormのAPIは非常にシンプルです。複雑な分散処理などをとくに意識することなく、システムを開発可能です

②拡張性

Stormは複数のマシンで構成されるクラスタ上で並列分散的に動作します。これにより、膨大な数のメッセージに対しても低レイテンシを維持しつつ、スケールします。スケールするために必要なことは、マシンを増設して処理の並列数を増やすだけであり、プログラムを改修する必要はありません

③耐障害性

障害が発生し、データ処理中のノードがダウンした場合でも、Stormは必要に応じてタスクの再割り当てやノードの再起動を行います。そのため、処理が完全に停止してしまうようなことがありません[注1]

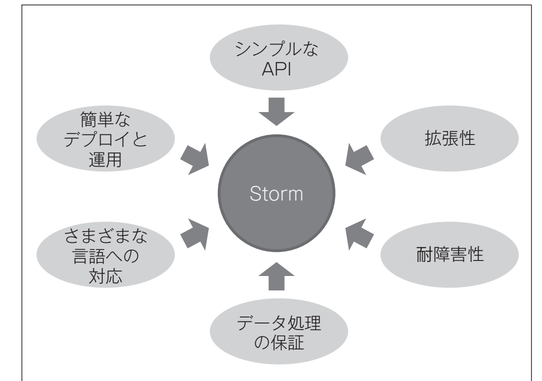
④データ処理の保証

何らかの理由により、データの処理に失敗したり、タイムアウトが発生したりした場合でも、Stormはそれを検知し、再処理するしくみを有しています。この機構により、すべてのメッセージが処理されることを担保できます

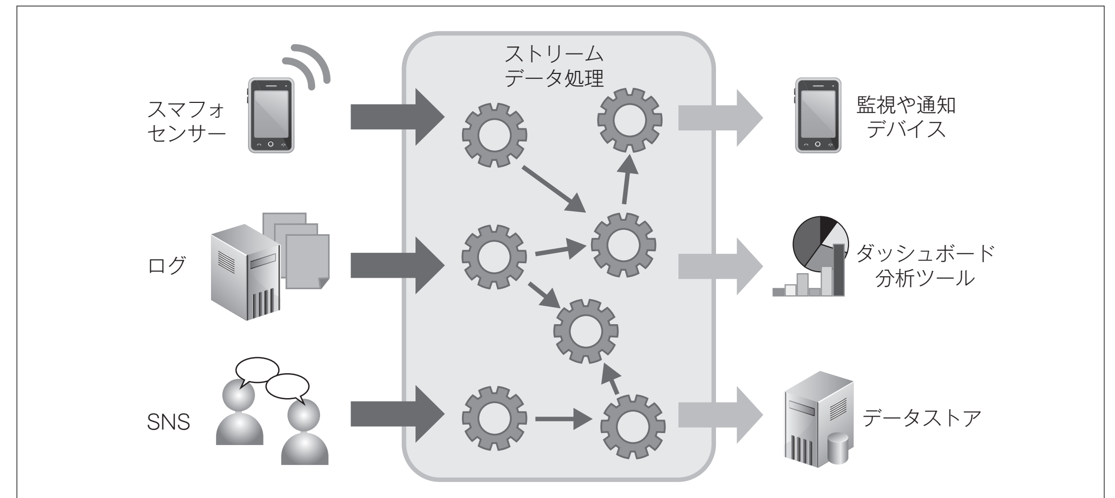
⑤さまざまな言語への対応

Storm自体はClojure[注2]で実装されていますが、ユーザが開発するアプリケーション部分は、いろいろな言語で開発できます。Java、Scala、Ruby、

■図2 Stormの6つの特徴



■図1 ストリームデータ処理



注1) 処理のタイミング次第では、同一メッセージが重複して処理されることがあります。
注2) LISP系の言語の方言の1つ。関数型プログラミングのスタイルでのインタラクティブな開発を支援し、マルチスレッドプログラムの開発を容易化する汎用言語です。Java仮想マシン上で動作します。http://clojure.org/

Python、Perl、JavaScript、および、PHPなどの多くの言語をサポートしています

⑥簡単なデプロイと運用

Stormは簡単にデプロイし、動作させることが可能です。システム構成もわずかな設定で変更できます。また、Amazon EC2などのクラウド環境でも動作させられます

Stormでとくに興味深いのは、耐障害性やデータが完全に処理されることをサポートしている点にあるでしょう。このようなしくみが標準で備わっているため、ミッションクリティカルな分野にも適用しやすくなっています。

Stormでできること

ここではStormで実現できる、主な機能を紹介します。

①継続的な並列処理

Stormの基本的な機能です。通常はストリームデータであるメッセージを一時的にキューに保存し、そのメッセージを継続的、かつ、並列的に処理し続けます

②メッセージのグルーピング

メッセージは特定のルールに従って、グルーピングして処理できます。グルーピングの方法としては、ランダム(Shuffle grouping)、指定されたフィールドの値が一致するもの(Fields grouping)、全メッセージを処理するもの(All grouping)などがあります。Stormが標準で提供する7種類のグルーピングのほか、独自のグルーピングを実装することも可能です

③トランザクション

前述のとおり、Stormはメッセージが必ず処理されることを保証します。しかしながら、障害が発生し、メッセージが再送された場合、重複して処理される可能性があります。たった一度だけ処理をしたい場合は、トランザクション機能(Transactional topologies)を利用します。トランザクション機能を利用した場合、Stormで処理されるメッセージには、トランザクションIDが自動で付与されます。このトランザクションIDを利用することで、重複する処理を判別することができます

Stormの基本アーキテクチャ

Stormのプロセス構成、および、アプリケーション構成について説明します。

プロセス構成

Stormは複数マシンにまたがって動作させられます。それら全体を「クラスタ」と呼びます。Stormのクラスタは基本的にはMaster-Slave構成をとり、具体的には図3のような構成となります。

◎Nimbus

クラスタ内におけるMasterノードであり、SupervisorやWorkerプロセスの管理を行います

◎Supervisor

クラスタ内におけるSlaveノードであり、タスク(後述のSpoutやBoltが該当します)のアサイン待ち受けや、Workerプロセスの起動／停止を行います

◎Worker

タスクを実行するプロセスになります

◎Zookeeper

ZookeeperはApache Foundationのプロダクトの1つであり、Hadoopのサブプロジェクトの1つとして開発されました。StormではNimbusとSupervisor間の協調管理に用いられています

アプリケーション構成

Storm上で動作するアプリケーションは、Topology(トポロジ)と呼ばれ、図4のようなイメージになります。Topologyは次の要素から構成されます。

◎Tuple

Stormで処理されるメッセージのことです。デフォルトではinteger、long、short、byte、string、double、float、boolean、byte配列などをサポートします

◎Stream

途切れずに連続するTupleを意味します。どのようなTupleが流れるかはグルーピングより決定されます

◎Spout

Streamのソースとなるもので、外部からデータを取得したり、受け付けたりして、Tupleを生成／送出します。Stormの処理の起点となるものです

◎Bolt

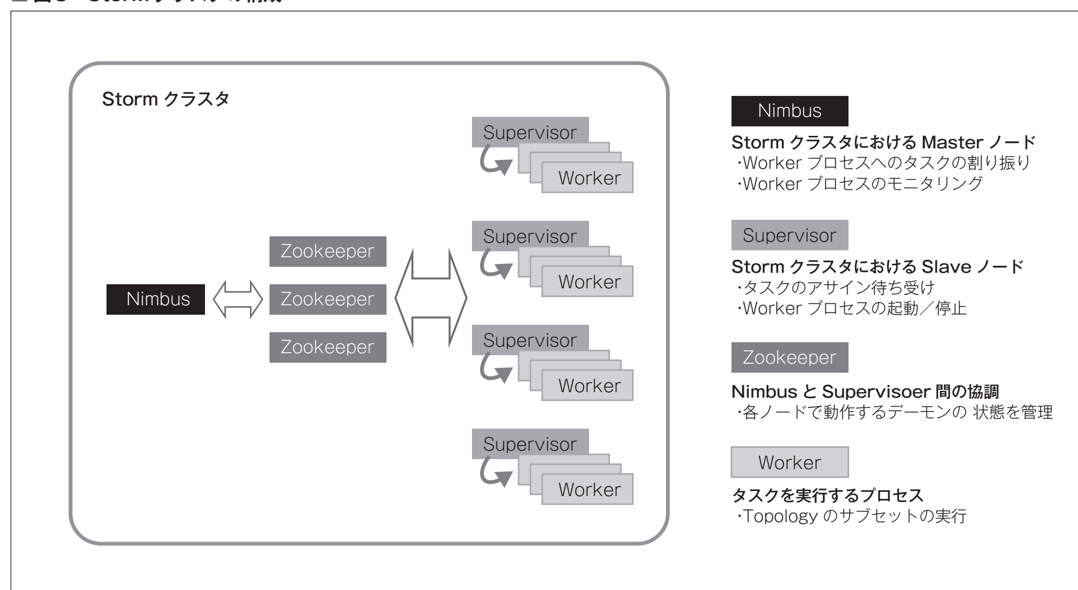
Streamの変換処理を行います。単一または複数のStreamからTupleを受信し、加工したうえで、新たなStreamにメッセージを送信します

Topologyはある一連の業務フローに該当するものです。Topology同士はStormクラスタ内で複数存在することが可能であり、それぞれのTopologyは独立して動作します。

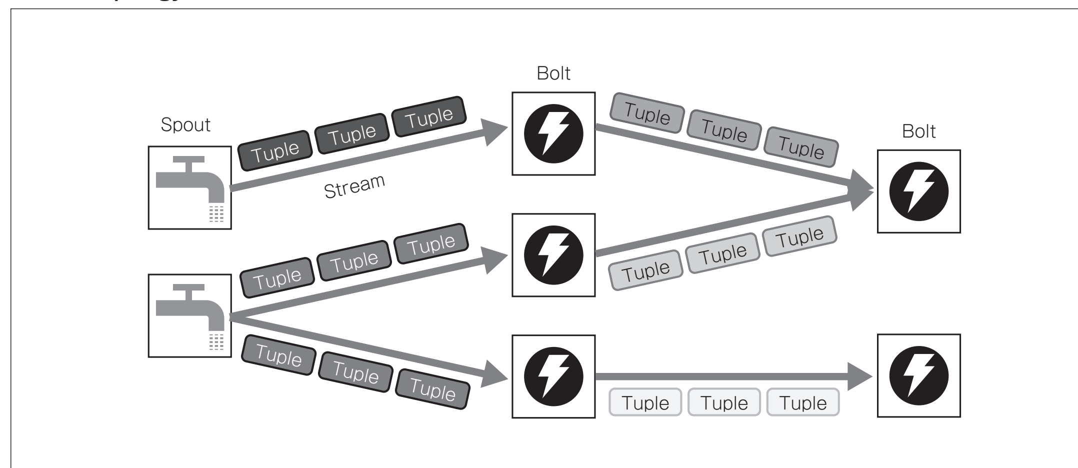
基本アーキテクチャ

Stormを利用したストリームデータ処理を行うシステムは、基本的には図5のようなアーキテクチャとなります。

■図3　Stormクラスタの構成



■図4　Topology



①イベント受付

処理対象のイベントは一時的にメッセージキューに保存するのが一般的です。これは大量のイベントを受信した場合でも、イベントを消失することなく処理を継続するためです

②イベント処理

次にメッセージキューに一時的に保存されたイベントをStormが取得し(Spout)、業務に応じた処理(Bolt)を分散して行います

③結果の処理

Storm自体は処理結果を表示する画面などは提供していません。そのため、Stormで処理をした結果はユーザへ通知されたり、RDBやKVS(NoSQL)などに保存されたりします

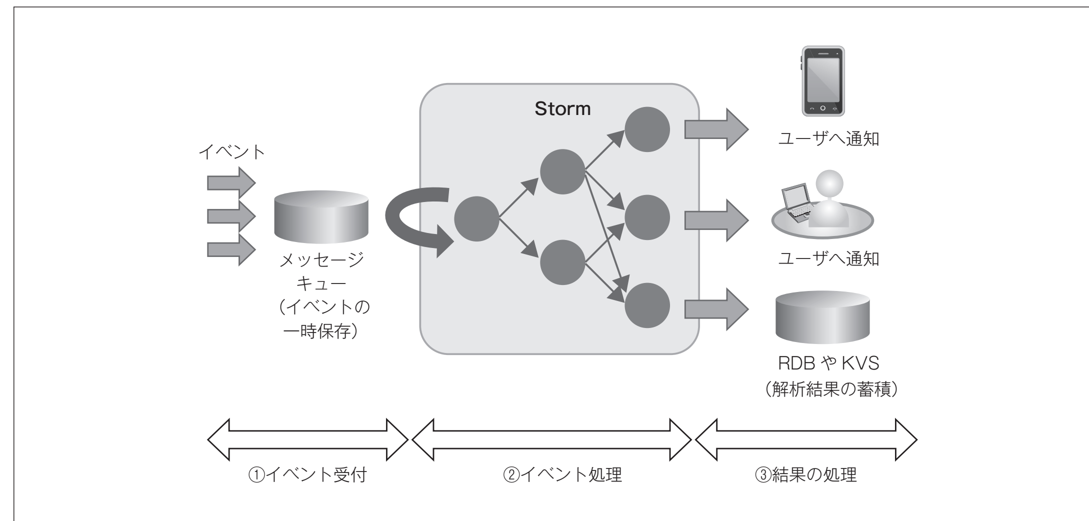
Storm自体はデータを永続化する機能や、ユーザに処理結果を通知するためのUIなどは、提供していません。それらが必要な場合は、システムに合わせて開発することが必要となります。

Hadoopとの比較

冒頭で、StormはHadoopでは困難なリアルタイム処理を実現するために開発された、と書きましたが、HadoopとStormを比較した場合のそれぞれの特徴を**表1**に示します。

アーキテクチャとしてはHadoopに似た構成となっている部分もあり、どちらも分散処理を行うフレームワークですが、最大の違いはバッチ処理なのか、リアルタイム処理なのか、というところにあります。

■ 図5　Stormを利用したシステムのアーキテクチャ



■ 表1　HadoopとStormの比較

	Hadoop	Storm
対象	バッチ処理	リアルタイム処理
処理技術	MapReduce	ストリームデータ処理
特性	・巨大で有限のジョブ ・たくさんのデータを一度だけ処理する ・長い待ち時間	・小さい無限のジョブ ・無限のストリームデータを連続して処理する ・短い待ち時間
ノード	マスター：JobTracker スレーブ：TaskTracker	マスター：Nimbus スレーブ：Supervisor
処理単位	Job	Topology

ただし、HadoopとStormは競合するものではありません。システムによっては、両者を組み合わせて処理を行うケースもあります。重要なことは、導入するシステムの特性を見極め、その特性に応じて使い分けることと言えるでしょう。

サンプルプログラム

次に、サンプルプログラムを用いてStormの実装内容を解説します。今回のサンプルプログラムは、文章を解析してそこに登場する単語数をカウントするものです。Stormのサンプルプログラムとして公開されているものがベースになっていますが、もともとSpout内部で文章を生成していた部分を、実際のシステムに近いイメージで処理するように、メッセージキューのOSSであるKestrelと連携して処理する構成に変更しています(**図6**)。

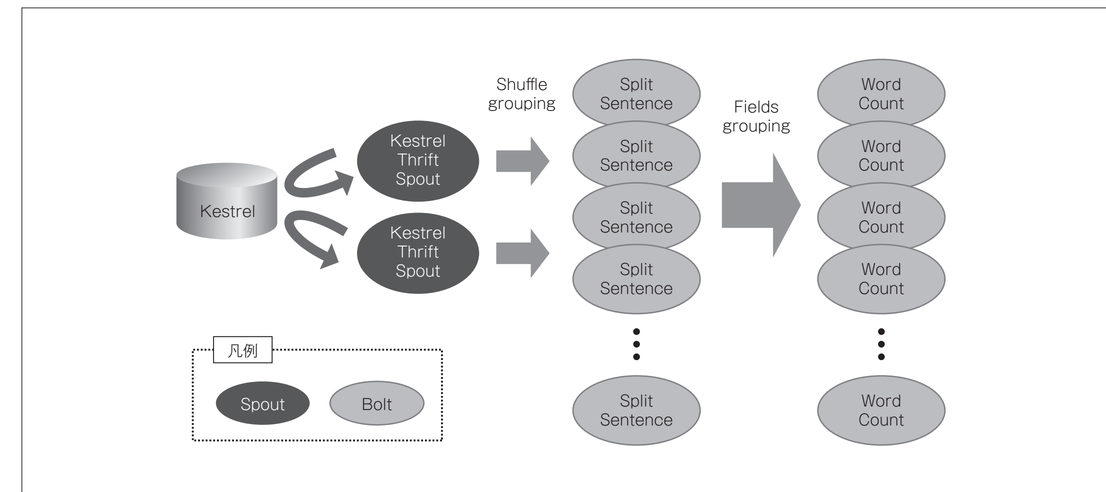
それぞれのコンポーネントの処理内容は、**表2**のようになります。以降では、各コンポーネントの実装について説明します。

データを取得する ーKestrelThriftSpoutー

Stormが提供するBaseRichSpoutを継承してSpoutクラスを実装します(**リスト1**)。KestrelThriftSpoutではおもに次の流れで処理を行います。

①Kestrelへ接続する
②Kestrelからデータを取得する
③Boltへデータを送信する

■ 図6　サンプルプログラムの処理イメージ



■ 表2　コンポーネントの処理内容

コンポーネント	処理内容
KestrelThriftSpout (Spout)	メッセージキューのKestrelから文章データ(今回は1行あたり約100バイトの英文データ)を取得する 取得した文章データを1行ずつBoltへ送信する
SplitSentenceBolt (Bolt)	1行の英文の文章を単語単位に分割する
WordCountBolt (Bolt)	分割された単語をカウントする

●openメソッド

最初に、Kestrelへ接続を行うため、Spoutのopenメソッドをオーバーライドします(**リスト2**)。このメソッドはSpoutが起動時に呼び出されます。ここでは、Kestrelとの接続を行うためのクラスであるKestrelClientInfoクラスをnewしています。

●nextTupleメソッド

次に、Kestrelからデータを取得するために、nextTupleメソッドをオーバーライドします(**リスト3**)。tryEachKestrelUntilBufferFilledメソッド内(本稿では具体的な処理は割愛します)で、Kestrelからデータを取得し、一時的にemitBufferに英文データを格納しています。

その後で、collectorのemitメソッドを呼び出すことでデータをBoltに送出しています。collectorはStormのSpoutやBolt間でTupleを受け渡すためのクラスで、openメソッド中で初期化されています。

注意点としては、emitするオブジェクト(Tuple)はシリアライズ可能であること、また、一意のIDを指定して送出する必要がある点があります。

Spoutでは、上記の他に、Boltでの処理が成功/失敗した際に呼び出される、ack/failメソッドを必要に応じて実装します。

■ リスト1　KestrelThriftSpoutクラス

```
public class KestrelThriftSpout extends BaseRichSpout {

  @Override
  public void open(Map conf,
                     TopologyContext context,
                     SpoutOutputCollector collector) {
       (中略)
  }

  @Override
  public void nextTuple() {
       (中略)
  }

  @Override
  public void ack(Object msgId) {
       (中略)
  }

  @Override
  public void fail(Object msgId) {
       (中略)
  }

}
```

■ リスト2　openメソッド

```
@Override
public void open(Map conf,
                   TopologyContext context,
                   SpoutOutputCollector collector) {
  this.collector = collector;

  (中略)

  int numTasks = context.getComponentTasks(context.getThisComponentId()).size();
  int myIndex = context.getThisTaskIndex();
  int numHosts = this.hosts.size();

  if (numTasks < numHosts) {
    for (HostInfo host: this.hosts) {
      this.kestrels.add(new KestrelClientInfo(host.host, host.port));
    }
  } else {
    HostInfo host = this.hosts.get(myIndex % numHosts);
    this.kestrels.add(new KestrelClientInfo(host.host, host.port));
  }
}
```

単語単位に分割する ―SplitSentenceBolt―

今回はSplitSentenceBolt、および、WordCountBoltの2種類のBoltを実装していますが、最初にSplitSentenceBoltについて説明します。SplitSentenceBoltはStormが提供するBaseRichBoltを継承して、Boltクラスを実装します(**リスト4**)。SplitSentenceBoltでは、おもに次の流れで処理を行います。

①Boltを初期化する
②Tuple(1行の英文)を受信する
③1行の英文を単語に分割し、次のBoltへ送出する

●prepareメソッド

Boltを初期化するためのprepareメソッドをオーバーライドします(**リスト5**)。ここではcollectorをインスタンス変数に設定するだけです。

●executeメソッド

executeメソッドの引数であるTupleは、英文の1行のデータです。そのデータを単語単位に分割します。その分割された単語は次のBoltへ送出するために、Spoutのときと同様に、collectorのemitメ

■ リスト3　nextTupleメソッド

```
@Override
public void nextTuple() {
  if (this.emitBuffer.isEmpty()) {
    tryEachKestrelUntilBufferFilled();
  }

  EmitItem item = this.emitBuffer.poll();
  if (item != null) {
    if (this.immediateAck) {
      this.collector.emit(item.tuple);
    } else {
      this.collector.emit(item.tuple,
                          item.sourceId);
    }
  }

  // Sleep Interval
  Utils.sleep(this.tupleEmitInterval);
}
```

■ リスト4　SplitSentenceBoltクラス

```
public class SplitSentenceBolt extends BaseRichBolt implements IRichBolt {

  @Override
  public void prepare(Map stormConf,
                      TopologyContext context,
                      OutputCollector collector) {
    (中略)
  }

  @Override
  public void execute(Tuple input) {
    (中略)
  }

  @Override
  public void declareOutputFields(OutputFieldsDeclarer declarer) {
    (中略)
  }
}
```

ソッドを呼び出しています(**リスト6**)。

ここでTupleが処理されることを保証するための機構について、簡単に説明しておきます。TupleがStormのTopology内を移動していく途中で、処理が失敗した場合は、Spoutに通知されるようにする必要があります。Tupleは、並列分散で処理されるため、Tupleの生成元の参照を含めておくことで、Spoutへの通知を行えるようにします。この方法を「アンカリング」と呼びます。

実際にこの「アンカリング」を行っているのは、collector.emit()のステートメントになります。元のTupleを引数に含めていますが、そうすることでTupleの発生元をトレースすることが可能となります。さらに、collector.ack()のステートメントが実行され、各Tupleの処理結果が通知されます。もし、Bolt内での処理に失敗した場合は、ackメソッドではなく、failメソッドを呼び出すことが必要となります(処理されるTupleは、必ずackかfailが実行される必要があります)。

●declareOutputFieldsメソッド

declareOutputFieldsメソッドでは、Boltで送出するTupleの内容にフィールド名を付与します(**リスト7**)。今回の例では、「new Values(targetWord)」と、フィールドはtargetWordの1つだけであるため、「new Fields("word")」と1つのフィールド名を指定していますが、複数指定することも可能です。フィールド名はBoltがメッセージを受信する際のグルーピングで利用されたり、次で処理を行うBoltでフィールド名を指定して値を取得するのに利用されたりするのに必要となります。

■ リスト5　prepareメソッド

```
@Override
public void prepare(Map stormConf,
                      TopologyContext context,
                      OutputCollector collector) {
  this.collector = collector;
}
```

■ リスト6　executeメソッド

```
@Override
public void execute(Tuple input) {
  // 文章を単語単位に分割する
  String sentence = input.getStringByField("str");
  String[] words = StringUtils.split(sentence);

  // 単語単位にTupleに分割し、次のBoltに送信する
  for (String targetWord : words) {
    this.collector.emit(input, new Values(targetWord));
  }

  this.collector.ack(input);
}
```

■ リスト7　declareOutputFieldsメソッド

```
@Override
public void declareOutputFields(
                OutputFieldsDeclarer declarer) {
  declarer.declare(new Fields("word"));
}
```

単語をカウントする ―WordCountBolt―

WordCountBoltはBaseBasicBoltを継承しています(**リスト8**)。SplitSentenceBoltが継承していたBaseRichBoltクラスとの違いは、prepareメソッドを必要としない点です。WordCountBoltでは、おもに次の流れで処理を行います。

①Tuple(1単語)を受信する
②単語の出現回数をカウントする

●executeメソッド

SplitSentenceBoltでは、"word"というフィールド名を付与してTupleを送出していたので、ここではそのフィールド名を使ってTupleから単語データを取得し、カウントしています(**リスト9**)。同じ単語であれば、カウントアップされていきます。

●declareOutputFieldsメソッド

WordCountBoltでは、「new Values(word, count)」と2つのフィールドを指定しているため、"word"、"count"という2つのフィールド名を指定しています(**リスト10**)。

全体をつなげる ―WordCountTopology―

最後に、これまで作成したSpoutやBoltを連携させるためのTopologyクラスを実装します(**リスト11**)。Topologyを実装する際に意識するのは、並列

■ リスト8　WordCountBoltクラス

```
public class WordCountBolt extends BaseBasicBolt {

  /** 単語出現回数カウンタ */
  Map<String, Integer> counts = new HashMap<String, Integer>();

  @Override
  public void execute(Tuple input, BasicOutputCollector collector) {
    (中略)
  }

  @Override
  public void declareOutputFields(OutputFieldsDeclarer declarer) {
    (中略)
  }
}
```

■ リスト9　executeメソッド

```
@Override
public void execute(Tuple input, BasicOutputCollector collector) {
  // 単語出現回数カウンタからカウンタを取得
  String word = input.getStringByField("word");
  Integer count = this.counts.get(word);

  if (count == null) {
    count = 0;
  }
  count++;

  // 結果を単語出現回数カウンタに反映
  this.counts.put(word, count);

  collector.emit(new Values(word, count));
}
```

数とグルーピングとの指定です。

並列数はSpout／BoltをTopologyに登録する際に指定するだけです。パフォーマンスを向上させたければ、この数値を変更するだけで可能になります。グルーピングについて、SplitSentenceBoltはランダムにデータを受け取るshuffleGrouping、WordCountBoltは単語ごとにデータを受け取る必要があるためfieldsGroupingを指定しています。

今回のTopologyの処理でサーバ（CPU：Xeon E3-1230 3.2GHz 4Core、メモリ：32GB）2台で分散するようにして性能を測定したところ、次のような結果になりました。

◎スループット（Tuple／秒）

- KestrelThriftSpout：3,302
- SplitSentenceBolt：3,302
- WordCountBolt：545,223

■リスト10　declareOutputFieldsメソッド

```
@Override
public void declareOutputFields(
                OutputFieldsDeclarer declarer) {
  declarer.declare(new Fields("word", "count"));
}
```

■リスト11　WordCountTopologyクラス

```
public class WordCountTopology {
  public static void main(String[] args)
      throws Exception {
 （中略）

  // Topology を作成する
  TopologyBuilder builder = new TopologyBuilder();

  // Add Spout(KestrelThriftSpout)
  KestrelThriftSpout kestrelSpout
    = new KestrelThriftSpout(kestrelHosts,
                             kestrelQueueName,
                             new StringScheme());
  builder.setSpout("KestrelSpout",
                   kestrelSpout, 2);

  // Add Bolt(KestrelThriftSpout -> SplitSentence)
  builder.setBolt("SplitSentence",
                  new SplitSentenceBolt(), 20)
    .shuffleGrouping("KestrelSpout");

  // Add Bolt(SplitSentence -> WordCount)
  builder.setBolt("WordCount",
                  new WordCountBolt(), 10)
    .fieldsGrouping("SplitSentence",
                    new Fields("word"));

  if (isLocal) {
    LocalCluster cluster = new LocalCluster();
    cluster.submitTopology("WordCount",
        conf, builder.createTopology());
  } else {
   （中略）
  }
}
```

今回の実装で、並列分散処理を意識した内容はほとんどありません。そのような簡単な実装で、数千〜数十万のイベントを処理するようなシステムを簡単に実現できるようになるのは、Stormの効果であると言えます。

今後の動向

Stormはリアルタイムの並列分散処理を実現するには非常に強力なフレームワークですが、一方、それ単体では実現できる処理はシンプルなものです。そのため、StormとほかのOSSを連携するSpoutやBoltが提供されつつあります。

- Spout：Kestrel、RabbitMQ、Kafka、JMS、Redis、Scribe
- Bolt：HBase、Cassandra、Mongo

Hadoopもそれ単体ではMapReduceというシンプルな処理しか提供していませんでしたが、エコシステムという周辺プロダクトが整備されたことにより、非常に利便性が向上しました。SpoutやBoltは疎結合であるため、同様に周辺プロダクトが整うことで、より効率的にシステムを開発できるようになっていくことが期待されます。

また、その他にも、CEP（複合イベント処理）や機械学習といった、より高度な機能の実現が検討されています。このような機能が提供されれば、適用範囲もより広がるでしょう。海外では、すでにStormを利用した商用システムもいくつか公開されています。今後、日本でも、そのようなシステムが増えることが期待されます。SD

Stormの日本語情報は、まだまだ少ないですが、筆者が所属するAcroquest Technologyでは、若手エンジニアが、ブログでStormの情報を公開しています。

◎Taste of Tech Topics
`http://acro-engineer.hatenablog.com/`

本家のサイトでも公開されていないような、独自に調査や評価した内容も掲載しています。ぜひ、参考にしてみてください。

●参考情報

◎Storm本家サイト
`http://storm-project.net/`

◎Storm-Installer
`https://github.com/acromusashi/storm-installer`
Stormの環境を簡単に構築できるように、インストーラを公開しています。

◎本記事のサンプル
`https://github.com/acromusashi/storm-example-wordcount`
本書で利用したStormのサンプルプログラムです。