

前準備

- スライド: <http://bit.ly/1NvuNie> (Chrome 推奨)
 - PDF: <http://bit.ly/1OrSgCd>
-

- 無線LANを接続してください
- 最新版のソースを取得してください

```
# spark-hands-on-development ディレクトリ直下で実行してください
# 最新の`step1` ブランチをチェックアウト
git fetch
git checkout step1
```

- 質問や気になることなどはチャットへ投稿してください
 - <http://chat.tech-circle-10.mydns.jp/>
 - ID: guest / PW: guest でログインしてください

Apache Spark Hands-On Development @Tech-Circle #10

Twitterハッシュタグのランキングを実況する
Webアプリを開発してみよう

私は何者？

- 根来 和輝 Negoro Kazuki
- TIS株式会社 生産技術R&D室
- Typesafe Reactive Platform の有効性検証
 - Scala / Play / Slick / Akka



[@negokaz](https://twitter.com/negokaz)



[negokaz](https://github.com/negokaz)

Apache Spark とは？

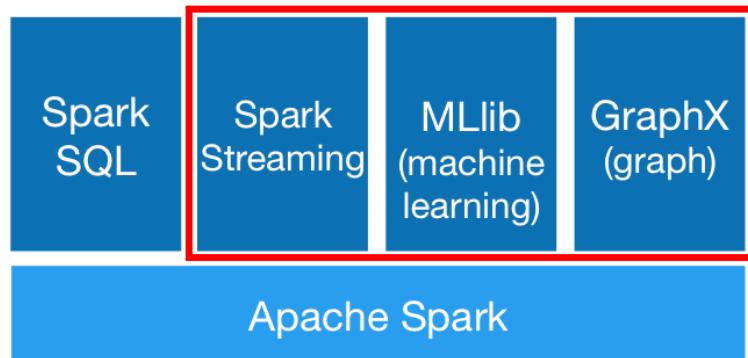


- ビックデータのための並列分散処理基盤
- 大量にあるデータの集計や分析ができる
- オープンソースで開発されている
- 最新のバージョンは 1.5.1 (2015/11 現在)

<http://spark.apache.org/>

Apache Spark とは？

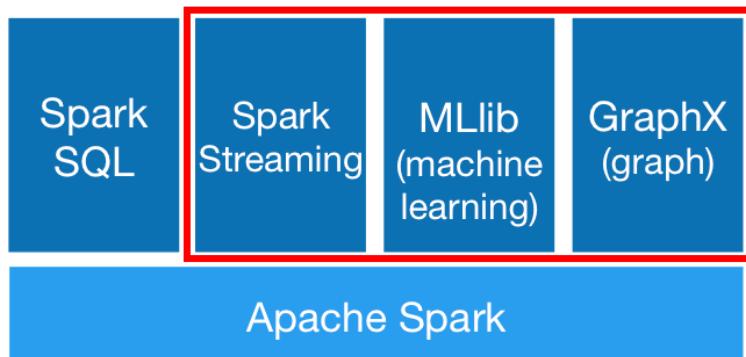
データを処理するための様々なライブラリを提供



- Spark Streaming
 - ストリーム処理
- MLlib
 - 機械学習
- GraphX
 - グラフ処理

Apache Spark とは？

データを処理するための様々なライブラリを提供

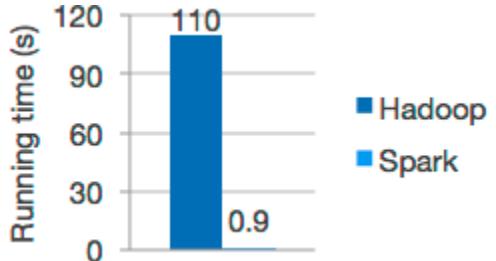


- Spark Streaming
 - ストリーム処理
- MLlib
 - 機械学習

それぞれのライブラリを組み合わせて
複雑な処理を実装できる。

● GraphX
◦ グラフ処理

Spark vs hadoop



- Hadoopは中間データをストレージに書き込む
 - スループットを高めることを重視
- Sparkは中間データをメモリに書き込む
 - レイテンシを低くすることを重視
- メモリ総量の数十倍のデータ量を扱う場合は
Hadoopが安定

Apache Sparkがスループットとレイテンシを両立させた仕組みと
最新動向をSparkコミッタとなったNTTデータ猿田氏に聞いた

Sparkの導入事例

- [Spark and Shark Bridges the Gap Between BI and Machine Learning](#)
 - Yahoo! 台湾法人
 - Hadoop MapReduce から Spark への移行
 - 日次レポート作成: 83% ↑ 機械学習: 7.5倍 ↑
- [Real-Time Recommendations using Spark](#)
 - 米 Comcast Labs.
 - 好みに合わせた番組のリアルタイムレコメンド
 - MLlib と Spark Streaming を組み合わせ

Sparkのユースケース

- 1台のサーバーではまかないきれないデータを集計・分析したい
 - クラスタの総メモリサイズに収まるデータ量
- 将来的にデータ量が増える可能性がある
- スループットよりもレイテンシを重視したい

より詳しく: [Apache Sparkが描く大規模インメモリ処理の世界](#)

Hands-On

これから開発するアプリ

Twitter Hashtag Ranking

1 #トレクル ×8 映像電伝虫発見！冬の島で見つけたのは……え？うさ…

映像電伝虫発見！冬の島で見つけたのは……え？うさぎっ!? <http://t.co/GDV81Yl7tt> #トレクル <http://t.co/EhhAW1wbGL>

映像電伝虫を捕まえました！巨人に海王類、そして、あの珍獣も？一味のメンバーはあなた次第!! <http://t.co/Fx4uKGRLBm> #トレクル <http://t.co/M8OkWwDXGd>

映像電伝虫発見！新たな船「メリー号」に乗り込む麦わらの一昧の姿を撮影しました！ <http://t.co/n4w0hQ3Yw6> #トレクル <http://t.co/tH0Wib7ysO>

映像電伝虫発見！捕らえられた海賊狩りを激写しました!!! <http://t.co/5BxEu97zVj> #トレクル <http://t.co/ZTp7IJGJ7d>

映像電伝虫発見！麦わらの海賊が処刑される寸前？ <http://t.co/cwyJwhiL9j> #トレクル <http://t.co/qNEpNdv6nS>

映像電伝虫発見！アーロン一味を激写しました！ <http://t.co/xxJp8ejbqg> #トレクル <http://t.co/SL9oiUzlo6>

映像電伝虫発見！新たな船「メリー号」に乗り込む麦わらの一昧の姿を撮影しました！ <http://t.co/yHlaB4KfLe> #トレクル <http://t.co/uN3YexFpTv>

ハロウィンキャンペーン！★10/13(0:00)～11/2(23:59)期間限定★期間中のトレクルはハロウィン仕様！！ <http://t.co/no7PzhXi99> #トレクル <http://t.co/4zNDBaXwhr>

2 #拡散希望 ×4 RT @siromomotoon: ギア交換しませんか 此方は画像参...

3 #相互フォロー ×4 【相互フォロー】募集中！！相互フォロー用に作成され.

4 #ふあばした人にやる ×4 RT @248017pippi: @nokonoko_39mmm 🎃名前➡よし...

5 #RTした人全員フォローする ×4 RT @ARASHI_ARAN0611: 初挑戦 ✨二宮和也の誕生日ま..

6 #パズドラ ×3 ▼【パズドラ】メイメイいっぱい捕まえてきたんだが逃..

ハッシュタグのランキングを実況するWebアプリ これから開発するアプリ

- たくさんTweetされた順にハッシュタグを表示

Twitter Hashtag Ranking

- 順位、Tweet数が確認できる

1 #トレクル x8 映像電伝虫発見！その鳥を見つけたのは、アースアコス。

- ハッシュタグをクリックするとTweetが見える

映像電伝虫捕まえました！巨人に海王類、そして、あの珍獣も？ 一味のメンバーはあなた次第!! <http://t.co/Fx4uKGRLBm> #トレクル <http://t.co/MBOkWwDXGd>

[Twitter - そもそも#ハッシュタグって何？](#)

映像電伝虫発見！捕らえられた海賊狩りを激写しました!!! <http://t.co/5BxEu97zVj> #トレクル <http://t.co/ZTp7IJGJ7d>

映像電伝虫発見！麦わらの海賊が処刑される寸前? <http://t.co/cwyJwhiL9j> #トレクル <http://t.co/qNEpNdv6nS>

映像電伝虫発見！アーロン一味を激写しました! <http://t.co/xxJp8ejbqg> #トレクル <http://t.co/SL9oUzlo6>

映像電伝虫発見！新たな船「メリー号」に乗り込む麦わらの一味の姿を撮影しました! <http://t.co/yHlaB4KfLe> #トレクル <http://t.co/uN3YexFpTv>

ハロウィンキャンペーン！★10/13(0:00)～11/2(23:59)期間限定★期間中のトレクルはハロウィン仕様！！<http://t.co/no7PzhXi99> #トレクル <http://t.co/4zNDBaXwhr>

2 #拡散希望 x4 RT @siromomotoon: ギア交換しませんか 此方は画像参...

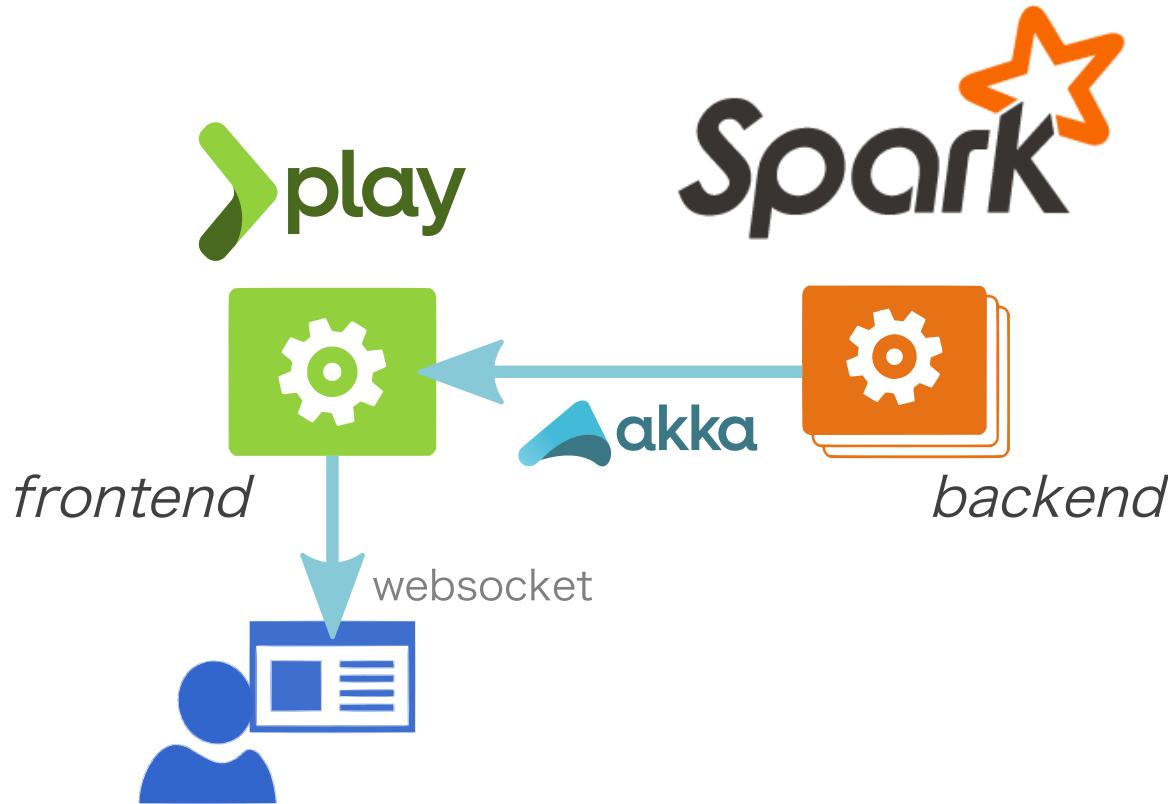
3 #相互フォロー x4 【相互フォロー】募集中！！相互フォロー用に作成され.

4 #ふあばした人にやる x4 RT @248017pippi: @nokonoko_39mmm 🎃名前よし...

5 #RTした人全員フォローする x4 RT @ARASHI_ARAN0611: 初挑戦✨二宮和也の誕生日ま..

6 #パズドラ x3 ▼【パズドラ】メイメイいっぱい捕まってきたんだが逃..

構成



Sparkでプログラミング ①

静的データを処理する

準備

1. IntelliJ IDEA を起動
2. **SparkLogic.scala** を開く

- modules > backend > src > main > scala > com.example.tagrank.backend > spark
- Shift × 2 で SparkLogic.scala を入力すると簡単に検索できます

現在の実装

```
/**  
 * ① テキストファイルからツイートを読み込んで解析  
 */  
def analyzeRanking(sc: SparkContext, ssc: StreamingContext, receiver: ActorSelection): Unit = {  
  
    // tweets.txt の中にある全ツイートの RDD  
    val tweetsRDD: RDD[String] = sc.textFile(tweetsFilePath)  
  
    // Ranking に変換する RDD  
    val rankingsRDD: RDD[Ranking] =  
        tweetsRDD map { tweet: String =>  
            // String を Ranking ケースクラスに変換  
            Ranking("#hashTag", rank = 1, Array(tweet), sampleCount = 1)  
        }  
  
    // collect() を呼び出すことによって実際の処理が始まる  
    val rankings: Array[Ranking] = rankingsRDD.collect()  
  
    // フロントエンドに結果を渡す  
    receiver ! rankings  
}
```

現在の実装

```
/**  
 * ① テキストファイルからツイートを読み込んで解析  
 */  
def analyzeRanking(sc: SparkContext, ssc: StreamingContext, receiver: ActorSelection): Unit = {  
  
    // tweets.txt の中にある全ツイートの RDD  
    val tweetsRDD: RDD[String] = sc.textFile(tweetsFilePath)  
  
    // Ranking に変換する RDD  
    val rankingsRDD: RDD[Ranking] =  
        tweetsRDD map { tweet: String =>  
            // String を Ranking ケースクラスに変換  
            Ranking("#hashTag", rank = 1, Array(tweet), sampleCount = 1)  
        }  
  
    // collect() を呼び出すことによって実際の処理が始まる  
    modules/backend/src/main/resources/tweets.txt  
  
    // フロントエンドに結果を渡す  
    receiver ! rankings  
}
```

- テキストファイルのデータからRDDを作成

- **modules/backend/src/main/resources/tweets.txt**

RDD ? - Resilient Distributed Datasets

- 耐障害性のある分散データセット
- イミュータブル
- データはパーティションに分割され
各ノードに分散して配置される
- 入力元のRDDと処理内容の情報を持っているため
壊れても復旧可能

[Apache Spark - Resilient Distributed Datasets \(RDDs\) \(Scala API\)](#)

現在の実装

```
/**  
 * ① テキストファイルからツイートを読み込んで解析  
 */  
def analyzeRanking(sc: SparkContext, ssc: StreamingContext, receiver: ActorSelection): Unit = {  
  
    // tweets.txt の中にある全ツイートの RDD  
    val tweetsRDD: RDD[String] = sc.textFile(tweetsFilePath)  
  
    // Ranking に変換する RDD  
    val rankingsRDD: RDD[Ranking] =  
        tweetsRDD map { tweet: String =>  
            // String を Ranking ケースクラスに変換  
            Ranking(#hashTag", rank = 1, Array(tweet), sampleCount = 1)  
        }  
  
    // collect() を呼び出すことによって実際の処理が始まる  
    val rankings: Array[Ranking] = rankingsRDD.collect()  
  
    // フロントエンドに結果を渡す  
    receiver ! rankings  
}
```

現在の実装

- `map()` で `String` ⇒ `Ranking` に変換する

```
/**  
 * Ranking: hashTag, rank, sampleTweets,  
 */  
  
def sampleCount[T](implicit Tag: String, receiver: ActorSelection): Unit = {  
  
    // tweets.txt の中にある全ツイートの RDD  
    val tweetsRDD: RDD[String] = sc.textFile(tweetsFilePath)  
  
    // Ranking に変換する RDD  
    val rankingsRDD: RDD[Ranking] =  
        tweetsRDD map { tweet: String =>  
            // String を Ranking ケースクラスに変換  
            Ranking("#hashTag", rank = 1, Array(tweet), sampleCount = 1)  
        }  
  
    // collect() を呼び出すことによって実際の処理が始まる  
    val rankings: Array[Ranking] = rankingsRDD.collect()  
  
    // フロントエンドに結果を渡す  
    receiver ! rankings  
}
```

現在の実装

```
/**  
 * ① テキストファイルからツイートを読み込んで解析  
 */  
def analyzeRanking(sc: SparkContext, ssc: StreamingContext, receiver: ActorSelection): Unit = {  
  
    // tweets.txt の中にある全ツイートの RDD  
    val tweetsRDD: RDD[String] = sc.textFile(tweetsFilePath)  
  
    // Ranking に変換する RDD  
    val rankingsRDD: RDD[Ranking] =  
        tweetsRDD map { tweet: String =>  
            // String を Ranking ケースクラスに変換  
            Ranking("#hashTag", rank = 1, Array(tweet), sampleCount = 1)  
        }  
  
    // collect() を呼び出すことによって実際の処理が始まる  
    val rankings: Array[Ranking] = rankingsRDD.collect()  
  
    // フロントエンドに結果を渡す  
    receiver ! rankings  
}
```

現在の実装

```
/**  
 * ① テキストファイルからツイートを読み込んで解析  
●*/RDD の collect() を呼び出すことで RDD  
に定義した処理が初めて実行される  
def analyzeRanking(sc: SparkContext, ssc: StreamingContext, receiver: ActorSelection): Unit = {  
    // ツイートをテキストファイルから読み込む  
    val tweets: RDD[String] = sc.textFile(tweetTs5iPath)  
    // テキストファイルの読み込み  
    val rankingsRDD: RDD[Ranking] = tweets.map { tweet: String =>  
        // String を Ranking ケースクラスに変換  
        Ranking("#hashTag", rank = 1, Array(tweet), sampleCount = 1)  
    }  
  
    // collect() を呼び出すことによって実際の処理が始まる  
    val rankings: Array[Ranking] = rankingsRDD.collect()  
  
    // フロントエンドに結果を渡す  
    receiver ! rankings  
}
```

現在の実装

```
/**  
 * ① テキストファイルからツイートを読み込んで解析  
 */  
def analyzeRanking(sc: SparkContext, ssc: StreamingContext, receiver: ActorSelection): Unit = {  
  
    // tweets.txt の中にある全ツイートの RDD  
    val tweetsRDD: RDD[String] = sc.textFile(tweetsFilePath)  
  
    // Ranking に変換する RDD  
    val rankingsRDD: RDD[Ranking] =  
        tweetsRDD map { tweet: String =>  
            // String を Ranking ケースクラスに変換  
            Ranking("#hashTag", rank = 1, Array(tweet), sampleCount = 1)  
        }  
  
    // collect() を呼び出すことによって実際の処理が始まる  
    val rankings: Array[Ranking] = rankingsRDD.collect()  
  
    // フロントエンドに結果を渡す  
    receiver ! rankings  
}
```

現在の実装

```
/**  
 * ① テキストファイルからツイートを読み込んで解析  
 */  


- フロントエンドへ結果を返す



```
def analyzeRanking(sc: SparkContext, ssc: StreamingContext, receiver: ActorSelection): Unit = {
 // tweets の RDD を作成
 val tweetsRDD: RDD[String] = sc.textFile(tweetsFilePath)
 // Ranking ケースクラスを作成
 val ranking: RDD[Ranking] =
 tweetsRDD.map(tweet: String =>
 // String を Ranking ケースクラスに変換
 Ranking("#hashTag", rank = 1, Array(tweet), sampleCount = 1)
)

 // collect() を呼び出すことによって実際の処理が始まる
 val rankings: Array[Ranking] = ranking.collect()

 // フロントエンドに結果を渡す
 receiver ! rankings
}
```


```

アプリケーションの起動

```
### Mac OS X###
cd spark-hands-on-development
./activator backend/run
./activator run
```

```
### Windows ###
cd spark-hands-on-development
activator backend/run
activator run
```

- ▶ ブラウザで <http://localhost:9000/> を開く

コンソール上で動きを確認

- 毎回ブラウザで確認するのは手間なので
コマンドを用意しておきました

```
### Mac OS X ###
./activator backend/print
```

```
### Windows ###
activator backend/print
```

- ▶ Ctrl + C で終了

日本語のツイートを抽出 - 使うもの

- RDD#filter() を使う
 - filter() には 「String を引数にとって Boolean を返す関数」を渡す
 - true になる要素だけの RDD が生成される
 - def containsJapaneseChar(s: String): Boolean を用意しておきました
- tweetsRDD に filter() を適用してみる

[Scala API - RDD](#)

日本語のツイートを抽出 - 実装

```
// tweets.txt の中にある全ツイートの RDD  
val tweetsRDD: RDD[String] = sc.textFile(tweetsFilePath)  
  
// 日本語を含むツイートを抽出する RDD  
val japaneseTweetsRDD: RDD[String] =  
    tweetsRDD.filter(containsJapaneseChar)  
  
// Ranking に変換する RDD  
val rankingsRDD: RDD[Ranking] =  
    tweetsRDD map { tweet: String =>  
        // String を Ranking ケースクラスに変換  
        Ranking("#hashTag", rank = 1, Array(tweet), sampleCount = 1)  
    }
```

- tweetsRDD.filter() に containsJapaneseChar を渡す

日本語のツイートを抽出 - 実装

```
// tweets.txt の中にある全ツイートの RDD  
val tweetsRDD: RDD[String] = sc.textFile(tweetsFilePath)  
  
// 日本語を含むツイートを抽出する RDD  
val japaneseTweetsRDD: RDD[String] =  
    tweetsRDD.filter(containsJapaneseChar)  
  
// Ranking に変換する RDD  
val rankingsRDD: RDD[Ranking] =  
    japaneseTweetsRDD map { tweet: String =>  
        // String を Ranking ケースクラスに変換  
        Ranking("#hashTag", rank = 1, Array(tweet), sampleCount = 1)  
    }
```

- Ranking へは japaneseTweetsRDD から変換するように書き換えておく

ハッシュタグを抽出 - 設計

```
Ranking("#hashTag", rank = ???, Array(tweet), sampleCount = ???)
```

ハッシュタグとそのハッシュタグを含んでいる
ツイートの配列のペアを作らないといけない

ハッシュタグを抽出 - 設計

```
// japaneseTweetsRDD: RDD[String]  
"ツイートA #hashTag1"  
"ツイートB #hashTag1 #hashTag2"
```

▼ ハッシュタグとツイートのペアを作成

```
( "#hashTag1", "ツイートA #hashTag1")  
( "#hashTag1", "ツイートB #hashTag1 #hashTag2")  
( "#hashTag2", "ツイートB #hashTag1 #hashTag2")
```

▼ 同じハッシュタグでグループ分け

```
( "#hashTag1", [ "ツイートA #hashTag1", "ツイートB #hashTag1 #hashTag2" ] )  
( "#hashTag2", [ "ツイートB #hashTag1 #hashTag2" ] )
```

▼ Rankingに変換

```
Ranking( "#hashTag1", rank = ???, [ "ツイートA #hashTag1", "ツイートB #hashTag1 #hashTag2" ], sampleCount = ???)  
Ranking( "#hashTag2", rank = ???, [ "ツイートB #hashTag1 #hashTag2" ], sampleCount = ???)
```

ハッシュタグを抽出 - 使うもの

```
// japaneseTweetsRDD: RDD[String]  
"ツイートA #hashTag1"  
"ツイートB #hashTag1 #hashTag2"
```

▼ ハッシュタグとツイートのペアを作成

```
( "#hashTag1" , "ツイートA #hashTag1" )  
( "#hashTag1" , "ツイートB #hashTag1 #hashTag2" )  
( "#hashTag2" , "ツイートB #hashTag1 #hashTag2" )
```

ハッシュタグを抽出 - 使うもの

```
// japaneseTweetsRDD: RDD[String]
"ツイートA #hashTag1"
"ツイートB #hashTag1 #hashTag2"
```

▼ ハッシュタグとツイートのペアを作成

```
( "#hashTag1", "ツイートA #hashTag1")
( "#hashTag1", "ツイートB #hashTag1 #hashTag2")
( "#hashTag2", "ツイートB #hashTag1 #hashTag2")
```

- `RDD#flatMap()`, `RDD#map()` を使う
 - `for`式でシンプルに記述できる
- `def pickHashTags(s: String): Set[String]`
を用意しておきました

[Scala API - RDD](#)

ハッシュタグを抽出 - 実装

```
// flatMap, map を使った場合
val hashTagTweetPairRDD: RDD[(String, String)] =
  japaneseTweetsRDD flatMap { tweet =>
    pickHashTags(tweet) map { hashTag =>
      (hashTag, tweet)
    }
  }

// for式を使った場合
val hashTagTweetPairRDD: RDD[(String, String)] =
  for {
    tweet   <- japaneseTweetsRDD
    hashTag <- pickHashTags(tweet)
  } yield (hashTag, tweet)
```

```
// Ranking に変換する RDD
val rankingsRDD: RDD[Ranking] =
  hashTagTweetPairRDD map { case (hashTag, tweet) =>
    Ranking(hashTag, rank = 1, Array(tweet), sampleCount = 1)
  }
```

ハッシュタグでグループ分け - 使うもの

```
( "#hashTag1", "ツイートA #hashTag1")
( "#hashTag1", "ツイートB #hashTag1 #hashTag2")
( "#hashTag2", "ツイートB #hashTag1 #hashTag2")
```

▼ 同じハッシュタグでグループ分け

```
( "#hashTag1", [ "ツイートA #hashTag1", "ツイートB #hashTag1 #hashTag2"])
( "#hashTag2", [ "ツイートB #hashTag1 #hashTag2"])
```

- `RDD#groupByKey()`
 - タプルの1番目をキーとしてグループ分けされる

[Scala API - RDD](#)

ハッシュタグでグループ分け - 実装

```
val hashTagGroupsRDD: RDD[(String, Iterable[String])] =  
  hashTagTweetPairRDD.groupByKey()
```

```
// Ranking に変換する RDD  
val rankingsRDD: RDD[Ranking] =  
  hashTagGroupsRDD map { case (hashTag, tweets) =>  
    Ranking(hashTag, rank = 1, tweets.toArray, sampleCount = tweets.size)  
  }
```

- sampleCountにはtweets.sizeを設定しておく

ツイートの多い順にソート

- RDD#sortBy() を使う
- tweets.size の降順にしておく

```
// ツイートの数でソートする RDD
val sortedHashTagGroupsRDD: RDD[(String, Iterable[String])] =
  hashTagGroupsRDD.sortBy({ case (_, tweets) =>
    tweets.size
  }, ascending = false)
```

```
// Ranking に変換する RDD
val rankingsRDD: RDD[Ranking] =
  sortedHashTagGroupsRDD map { case (hashTag, tweets) =>
    Ranking(hashTag, rank = 1, tweets.toArray, sampleCount = tweets.size)
  }
```

[Scala API - RDD](#)

ランクを設定

- `RDD#zipWithIndex()`
- `index` は 0 始まりなので +1 しておく

```
// ソートされた各要素にインデックスを付ける RDD
val rankedHashTagGroupsRDD: RDD[((String, Iterable[String]), Long)] =
  sortedHashTagGroupsRDD.zipWithIndex()
```

```
// Ranking に変換する RDD
val rankingsRDD: RDD[Ranking] =
  rankedHashTagGroupsRDD map { case ((hashTag, tweets), index) =>
    Ranking(hashTag, rank = index + 1, tweets.toArray, sampleCount = tweets.size)
  }
```

[Scala API - RDD](#)

解答

実装が間に合わなかった場合は下記コマンドで
解答をチェックアウトしてください

```
# spark-hands-on-development ディレクトリの直下で実行してください
git commit -am "「Sparkでプログラミング ①」の途中まで実装"
git checkout step2
```

※ 途中の作業は `step1` ブランチに保存されます

Sparkでプログラミング ②

ストリームデータを処理する

現在の実装

```
/**  
 * ② Spark Streams を使ってリアルタイムにツイートを解析  
 */  
def analyzeRankingWithStream(sc: SparkContext, ssc: StreamingContext, receiver: ActorSelection):  
  
    // Twitter の DStream  
    val twitterStream: ReceiverInputDStream[Status] =  
        TwitterUtils.createStream(ssc, None)  
  
    // ツイート の DStream に変換  
    val tweetStream: DStream[String] =  
        twitterStream map { status =>  
            status.getText  
        }  
  
    // ストリームの塊を処理する  
    tweetStream.foreachRDD { rdd: RDD[String] =>  
        /* ...省略... */  
    }
```

現在の実装

- DStream の API を通してストリームを操作する
- foreachRDD でストリームから RDD を取り出せる
 - * ② Spark Streams を使ってリアルタイムにツイートを解析

```
def analyzeRankingWithStream(sc: SparkContext, ssc: StreamingContext, receiver: ActorSelection):  
  
    // Twitter の DStream  
    val twitterStream: ReceiverInputDStream[Status] =  
        TwitterUtils.createStream(ssc, None)  
  
    // ツイート の DStream に変換  
    val tweetStream: DStream[String] =  
        twitterStream map { status =>  
            status.getText  
        }  
  
    // ストリームの塊を処理する  
    tweetStream.foreachRDD { rdd: RDD[String] =>  
        /* …省略… */  
    }
```

DStream ? - Discretized Stream

- 離散化ストリーム
- 連續するRDDの集合
 - RDDはバッチインターバルごとの入力データを持つ
 - バッチインターバルはStreamingContextに設定
- RDDと似たAPIを持つ

[Apache Spark - Discretized Streams \(DStreams\) \(Scala API\)](#)

現在の実装

```
/**  
 * ② Spark Streams を使ってリアルタイムにツイートを解析  
 */  
def analyzeRankingWithStream(sc: SparkContext, ssc: StreamingContext, receiver: ActorSelection):  
/* …省略… */  
  
// ストリームの塊を処理する  
tweetStream.foreachRDD { rdd: RDD[String] =>  
    // Ranking に変換する RDD  
    val rankingsRDD = rdd map { tweet: String =>  
        // String を Ranking ケースクラスに変換  
        Ranking("#hashTag", rank = 1, Array(tweet), sampleCount = 1)  
    }  
    // collect() を呼び出すことによって実際の RDD の処理が始まる  
    val rankings = rankingsRDD.collect()  
  
    receiver ! rankings  
}  
  
// start() を呼び出すことによって上記で定義した Stream の処理が始まる  
ssc.start()  
}
```

現在の実装

- foreachRDDで取り出したRDDには

```
/** バッチインターバルのデータが含まれている
 * ② Spark Streams を使ってリアルタイムにツイートを解析
 */
```

- バッチインターバル以上の時間幅の集計も可能

```
/* …省略… */

// ストリームの塊を処理する
tweetStream.foreachRDD { rdd: RDD[String] =>
    // Ranking に変換する RDD
    val rankingsRDD = rdd map { tweet: String =>
        // String を Ranking ケースクラスに変換
        Ranking("#hashTag", rank = 1, Array(tweet), sampleCount = 1)
    }
    // collect() を呼び出すことによって実際の RDD の処理が始まる
    val rankings = rankingsRDD.collect()

    receiver ! rankings
}

// start() を呼び出すことによって上記で定義した Stream の処理が始まる
ssc.start()
}
```

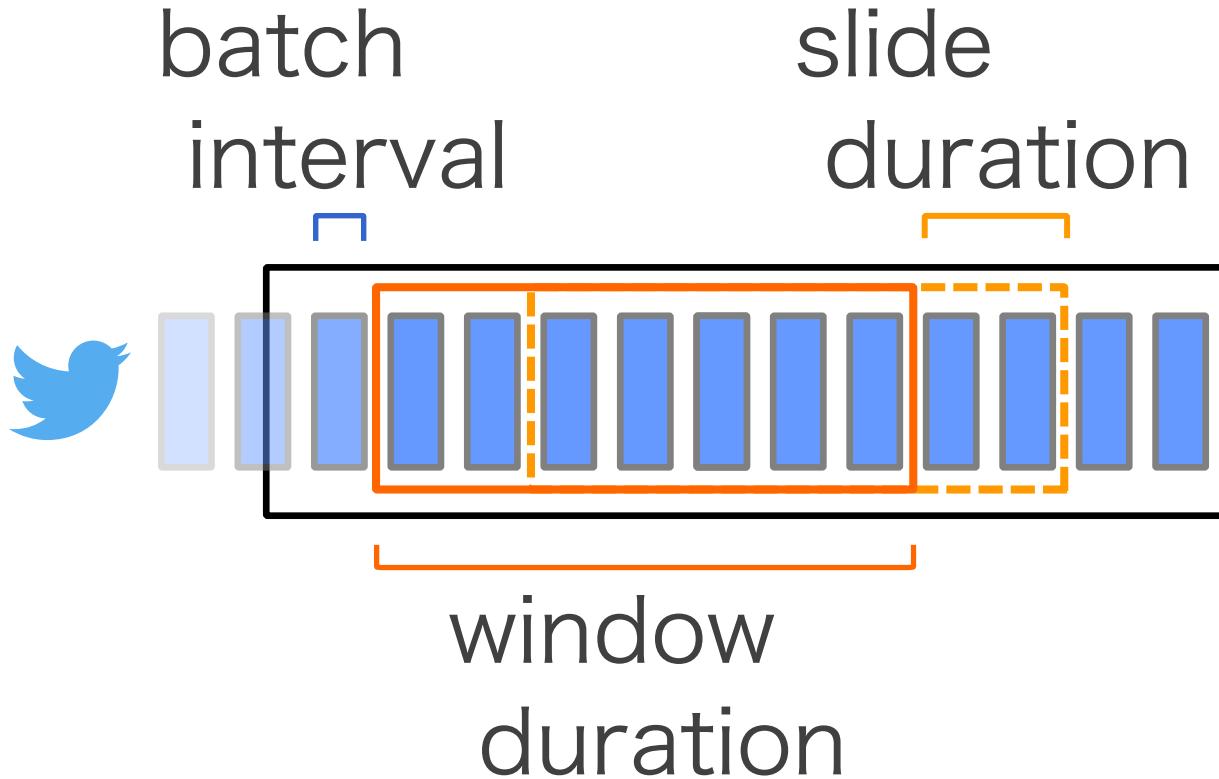
バッヂインターバル以上の集計

```
// ペアの1番目をキーとしてツイートのグループを作る DStream  
val hashTagGroupsStream: DStream[(String, Iterable[String])] =  
hashTagTweetPairStream.groupByKey()
```

- ⇒ バッヂインターバル(500ミリ秒)
ごとのツイートが集計される
- 過去何分かの集計を表示したい場合は
DStream#groupByKeyAndWindowを使って
ウィンドウ集計する。 ↓

```
// ペアの1番目をキーとしてツイートのグループを作る DStream  
val hashTagGroupsStream: DStream[(String, Iterable[String])] =  
hashTagTweetPairStream  
.groupByKeyAndWindow(windowDuration = Minutes(1), slideDuration = Seconds(1))
```

ウィンドウ集計



実装してみよう

- DStream#filterで日本語のツイートを抽出
- for or DStream#map, DStream#flatMap
でハッシュタグ・ツイートのペアを作成
- DStream#groupByKeyAndWindowで
過去1分間の集計を1秒ごとに行う
- 集計したものをDStream#foreachRDDの中で
 - ソートしインデックスを付与
 - Rankingに変換

実装してみよう

analyzeLogicを書き換えて確認してください

```
val analyzeLogic: analyzeLogicType = analyzeRanking
```



```
val analyzeLogic: analyzeLogicType = analyzeRankingWithStream
```

解答

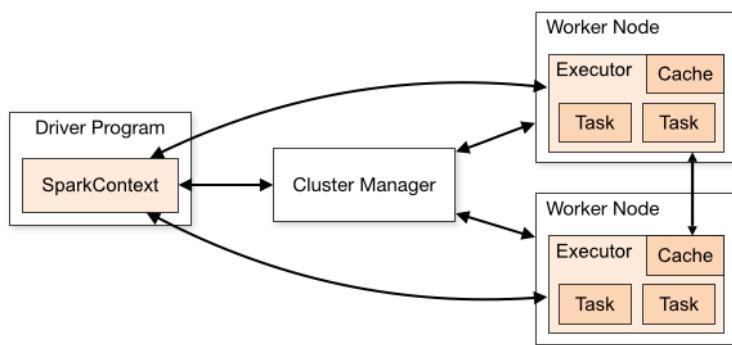
実装が間に合わなかった場合は下記コマンドで
解答をチェックアウトしてください

```
# spark-hands-on-development ディレクトリの直下で実行してください
git commit -am "「Sparkでプログラミング ②」の途中まで実装"
git checkout step3
```

※ 途中の作業は `step2` ブランチに保存されます

Sparkのクラスタを作る

Sparkのクラスターアーキテクチャ



- **Driver Program**
 - アプリケーションのメインプロセス。
SparkContextを起動する。
- **SparkContext**
 - クラスタへアクセスするための入り口となるオブジェクト。
これからRDDを作る。
- **Cluster Manager**
 - クラスタのリソースを管理する。
- **Worker Node**
 - アプリケーションコードが実行されるノード。
- **Executor**
 - ワーカーノード上で起動するプロセス。
Taskを実行したり中間データを保持する。
- **Task**
 - Executor上で実行される処理。

<http://spark.apache.org/docs/latest/cluster-overview.html>

手順

1. バックエンドの JAR(Java Archive) を作成
2. Sparkのクラスタを起動
 - Master
 - Worker
3. JAR を Spark のクラスタへ submit

バックエンドのJARを作成

```
### Mac OS X ###
cd spark-hands-on-development
# JAR を作成
./activator backend/assembly
# JAR が作成できていることを確認
ls modules/backend/target/scala-2.10/
```

```
### Windows ###
cd spark-hands-on-development
# JAR を作成
activator backend/assembly
# JAR が作成できていることを確認
dir modules\backend\target\scala-2.10\
```

▶ twitter-hashtag-ranking-backend-assembly-1.0.0.jar

クラスタの起動 - Master

```
spark-class org.apache.spark.deploy.master.Master
```

```
...  
15/10/13 12:44:34 INFO Master: Starting Spark master at spark://192.168.179.3:7077  
...
```

- 起動時のログに Master のアドレスが output される

クラスタの起動 - Workers

```
spark-class org.apache.spark.deploy.worker.Worker spark://192.168.179.3:7077
```

- 引数にMasterのアドレスを指定する
- Workerはいくつでも立ち上げることができる

クラスタへの submit

```
# JAR があるディレクトリに移動  
cd spark-hands-on-development/modules/backend/target/scala-2.10/  
# Sparkのクラスタへ submit  
spark-submit --deploy-mode cluster --master spark://192.168.179.3:7077 twitter-hashtag-ranking-ba
```

画面を確認

フロントエンドの起動

```
### Mac OS X ###
cd spark-hands-on-development
./activator run
```

```
### Windows ###
cd spark-hands-on-development
activator run
```

▶ ブラウザで <http://localhost:9000/> を開く

クラスタマネージャーの種類

- Spark Standalone
 - Sparkのパッケージに含まれるクラスタマネージャ。
簡単にクラスタを構築できる。
- Apache Mesos
 - 汎用クラスタマネージャ。
- Hadoop YARN
 - Hadoopのリソースマネージャ。

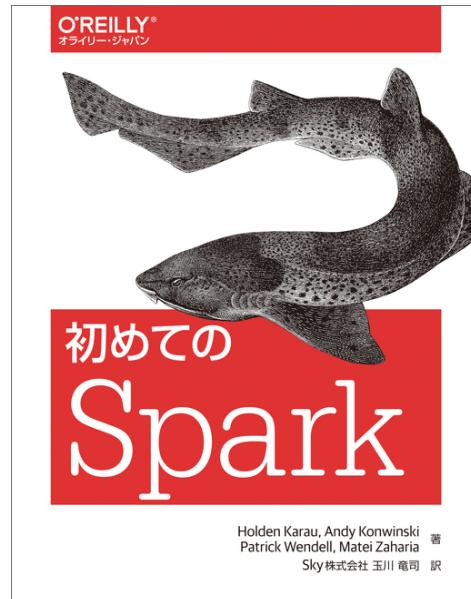
[Cluster Manager Types](#)

まとめ

- 分散処理が簡単にプログラミングできる
- ライブラリを組み合わせることで実現できることの幅が広がる
- プログラムを書き換えることなくスケールアウトできる

参考情報

書籍: 初めてのSpark



<https://www.oreilly.co.jp/books/9784873117348/>

書籍: Apache Spark入門



<https://www.shoeisha.co.jp/book/detail/9784798142661>

TIS(株) リアクティブ・システム コンサルティングサービス

- Typesafe のコンサルティングパートナー
 - Reactive Platform (Play / Akka / Slick / Scala)
 - 技術検証
 - 設計レビュー
 - コードレビュー
 - システム構築
- ※ Spark は近い将来ラインナップします！！
- https://www.tis.jp/service_solution/goreactive/

Thank you!

お願ひ

アンケートにご協力ください



[回答する](#)