ビッグデータ
Redis によるデータ管理
演習資料

演習 1

高速 NoSQL データベース Redis を利用して、ビッグデータ技術に触れましょう。

Redis とは

Redis は 2009 年にイタリアの個人プログラマーSalvatore Sanfilippo 氏によって開発され始めたインメモリによるキーバリュー型 NoSQL データベースです。現在は Redis Lab が開発主体となっています。

特徴

- ・マスター型のキーバリューストア。単一障害点があるものの、マスターがクラッシュした際にスレーブが、マスターに昇格します。（クラッシュしていないスレーブが残っている場合）
- ・マスターは複数のスレーブを持つことができ、非同期でスレーブに複製を作成します。
- ・コンシステント・ハッシングを用いたシャーディングをクライアント側で行います。
- ・オンメモリでデータが処理されるため、書き込み、読み込みが高速。書き込んだデータは一定時間が過ぎると非同期でディスクに保存されます。ディスクに永続化する前に電源が落ちるとデータは消失します。永続化は RDB に書き込む機能と AOF（Append Only File）によるコマンドの記録ファイルに書き込む機能のいずれかが選択できます。
- ・データストラクチャ・ストアとも呼ばれ、画像などのバイナリ、文字列、リスト、セット、ソート済みのセット、ハッシュなどといった多彩なデータを格納することができます。
- ・アトミックな処理（排他制御を行う事）によって更新をかけることができます。
- ・結果整合性　マスターからスレーブに複製するまでの間、システム内には複数のバージョンが存在します。つまり、最終的に結果が合えばよいという結果整合性の考え方を取ります。
- ・ハイパフォーマンス、レプリケーションなどの機能を要求される場合など有利。
- ・スケーラビリティが高いため、プロトタイピングに便利

デメリット

- ・Replace などの部分置換や条件抽出、集計などの複雑な操作は用意されていません。アプリ側か Lua スクリプトで記述する必要があります。
- ・ロールバック機能が存在しません。
- ・厳密な一貫性は担保されません。
- ・セキュリティ機能に乏しい。信頼されたアクセスを前提としているため、必要最低限です。
  アクセス制限、リモート攻撃は別途対策が必要。（コマンドのリネームは可能です。）

Redis の基本的なデータ型は 5 種類です。

型	格納されている情報	例
STRING 型	512 メガバイトまでのデータ(バイナリセーフ) 文字列、整数、浮動小数点数(数値は加減算可能)	”Redis” 10 1.2 など
LIST 型	文字列の連結リスト（42 億個の要素）先頭、末尾の追加は高速	1,2,3,1
SET 型	一意な文字列の順序のないコレクション 重複を許さないため、多彩な集合演算機能が搭載されている	1,5,2
HASH 型	順序のない、キーと値のハッシュテーブル	field1=1,field2=199 など項目名と値のペア
ZSET 型 （ソート済み集合）	浮動小数点数のスコア順に並べられた文字列 メンバーからスコアへのマッピング	東京：スコア 1, 大阪：スコア 2, 名古屋：スコア 3 などスコアの順序と共に格納

Redis で利用できるクエリ
先頭に L がついている場合は Lists、S の場合は Sets、Z の場合は Sorted Sets、H の場合は Hashes をそれぞれ操作するコマンドになっています。

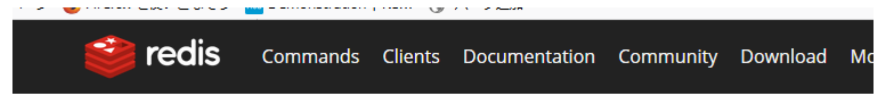
コマンド	説明
SET	キーとそれに対するバリューを指定して、新規に格納するか、既に格納済みの場合は更新する
GET	キーを指定して、バリューを取得する
DEL	キーを指定して、バリューを削除する
INCR	キーを指定して、数値バリューに 1 を足す
MSET	複数キーとそれぞれに対するバリューを指定して、新規に格納するか、既に格納済みの場合は更新する
MGET	複数キーを指定して、それぞれのキーに対するバリューを一括で取得する
LPUSH・RPUSH	キーで指定した List に対して、左端、右端に要素を追加する
LLEN	キーで指定した List の長さを取得する
LRANGE	キーとインデックスを二つ（開始・終了点）指定する事で、キーに対する List の 2 つのインデックス間の要素を取得する
LPOP・RPOP	キーで指定した List の左端、右端からそれぞれ要素を抜き出す。

コマンド	説明
SADD	キーと要素を指定して、指定した Set に対して要素を追加する
SREM	キーと要素を指定して、キーに対する Set から指定した要素を削除する
SISMEMBER	キーと要素を指定して、指定した要素が Set 内に存在するか確認する
SMEMBERS	キーと要素を指定して、Set の要素全てを取得する
SUNION	キーと要素を指定した 2 つ以上の Set を統合した結果を返す。重複は省かれる。
ZADD	キーとバリュー、スコアを指定してキーに対応するソート済み Set にバリューとスコアのエントリを追加する
ZRANGE	キーとインデックスを二つ（開始・終了点）指定することでキーに対するソート済み Set のインデックス間の要素を取得する
HSET	キーとフィールド、バリューを指定して、キーに該当した Hash にフィールドとバリューのエントリを追加する
HGETALL	キーで指定した Hash の要素全てを抜き出す。
HMSET	キーと複数のフィールド、バリューを指定して、キーに対する Hash に指定しフィールドとバリューのエントリをそれぞれ追加する
HGET	キーとフィールドを指定して、キーに対する Hash のフィールドのバリューを取得する
HINCRBY	キーとフィールド、数値を指定すると、キーに対応する Hash の中のフィールドに対応するバリューに指定した数値を加算する
HDEL	キーとフィールドを指定して、キーに対応する Hash の指定したフィールドとバリューを削除する

参考：Redis のインストール

１．インストーラーのダウンロード

https://redis.io/download



Download

Redis uses a standard practice for its versioning: **major.minor.patchlevel**. An even **minor**
like 1.2, 2.0, 2.2, 2.4, 2.6, 2.8. Odd minors are used for **unstable** releases, for example 2.9.x
unstable versions of what will be Redis 3.0 once stable.

Stable (4.0)

Redis 4.0 was released as GA in July 2017. Contains several big improvements: a modules system, much better replication (PSYNC2), improvements to eviction policies, threaded DEL/FLUSH, mixed RDB+AOF format, Raspberry Pi support as primary platform, the

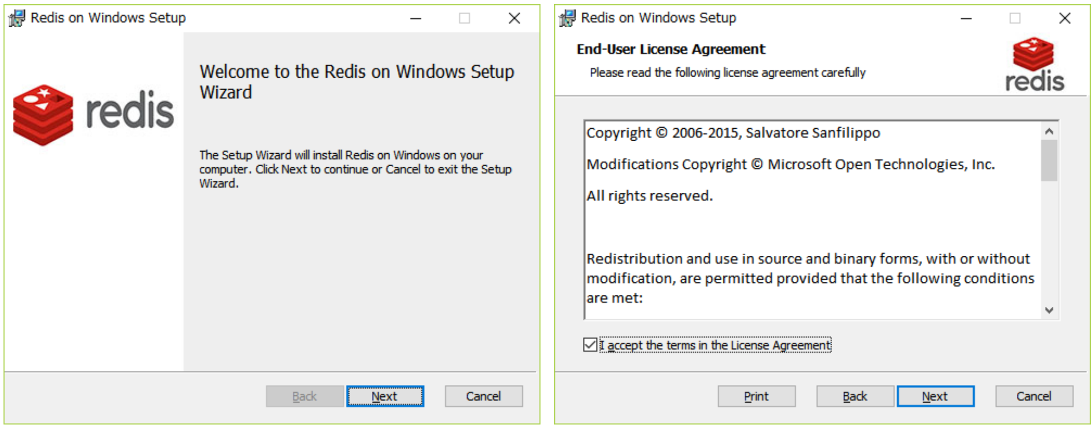
Unstable

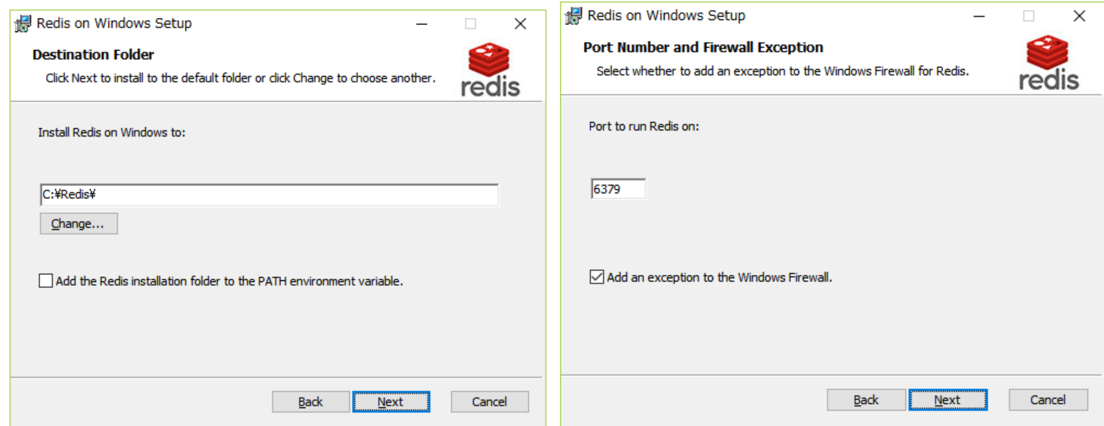
This is where all the development happens. Only for hard-core hackers. Use only if you need to test the latest features or performance improvements. This is going to be the next Redis release in a few months.

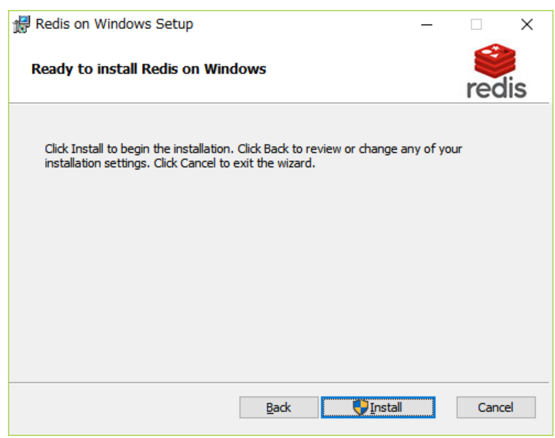
Windows 版は GitHub のリンクから msi ファイルを入手します。

Redis-x64-3.0.500-rc1.msi

２．インストールの実行







３．インストールの確認

- path の確認
    - ＞ path を実行して環境変数を表示します
    - C:¥develop¥Redis が表示されれば大丈夫です
- Redis の確認　（サービスが起動しているか確認しましょう。）

>redis-cli を実行。

```
c:¥>redis-cli
127.0.0.1:6379>
```

以下のように入力してみましょう

> set sample sample

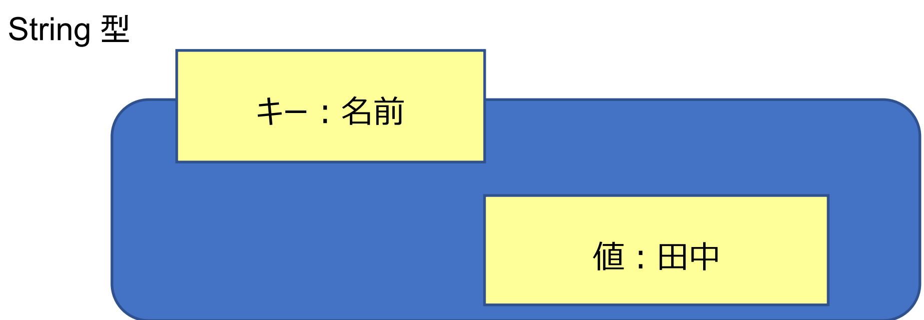
> get sample "sample"

```
127.0.0.1:6379> set sample sample
OK
127.0.0.1:6379> get sample
"sample"
```

演習　Redis の各型について、データの出し入れを行いましょう。

①String 型の保存イメージ

型	格納されている情報	例
STRING 型	512 メガバイトまでのデータ(バイナリセーフ) 文字列、整数、浮動小数点数(数値は加減算可能)	″Redis″　10　1.2 など



Redis の型操作（String）

String 型の操作

以下のように入力してみましょう。

　>set hello world

　>get hello

　>del hello

　>get hello

結果

```
127.0.0.1:6379> set hello world
OK
127.0.0.1:6379> get hello
"world"
```

```
127.0.0.1:6379> del hello
(integer) 1
127.0.0.1:6379> get hello
(nil)
127.0.0.1:6379> _
```

処理成功は 1、処理されない場合は 0 が返ります。

削除されたら値がないため、nil＝空が返る

②List 型の保存イメージ

型	格納されている情報	例
LIST 型	文字列の連結リスト（42 億個の要素）先頭、末尾の追加は高速	1,2,3,1

List 型

重複可

キー：名簿

値：田中

値：山田

値：田中

Redis の型操作（List）

List 型の操作

＞rpush list-key item

＞rpush list-key item2

＞rpush list-key item

＞lrange list-key 0 -1

＞lindex list-key 1

＞lpop list-key

＞lrange list-key 0 -1

結果

```
127.0.0.1:6379> rpush list-key item
(integer) 1
127.0.0.1:6379> rpush list-key item2
(integer) 2
127.0.0.1:6379> rpush list-key item
(integer) 3
```

```
127.0.0.1:6379> lrange list-key 0 -1
1) "item"
2) "item2"
3) "item"
```

先頭を 0　末尾を-1 指定すると

全リストを返します。

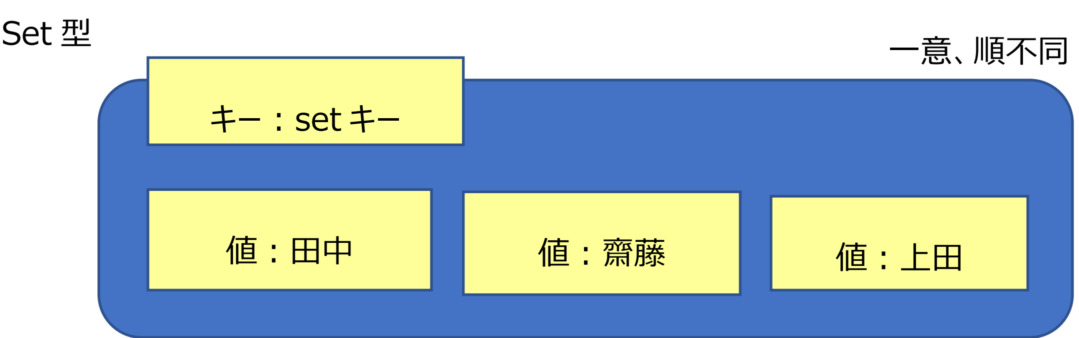
```
127.0.0.1:6379> lindex list-key 1
"item2"
```

```
127.0.0.1:6379> lpop list-key
"item"
127.0.0.1:6379> lrange list-key 0 -1
1) "item2"
2) "item"
```

リストの左端から要素を取りだしたため、リストから削除されます

③Set 型の保存イメージ

型	格納されている情報	例
SET 型	一意な文字列の順序のないコレクション 重複を許さないため、多彩な集合演算機能が搭載されている	1,5,2



Redis の型操作（Set）

Set 型の操作

>sadd set-key item
>sadd set-key item2
>sadd set-key item3

>sadd set-key item
>smembers set-key
>sismember set-key item4
>sismember set-key item
>srem set-key item2

>srem set-key item2
>smembers set-key

```
127.0.0.1:6379> sadd set-key item
(integer) 1
127.0.0.1:6379> sadd set-key item2
(integer) 1
127.0.0.1:6379> sadd set-key item3
(integer) 1
127.0.0.1:6379> sadd set-key item
(integer) 0
127.0.0.1:6379> smembers set-key
1) "item"
2) "item3"
3) "item2"
```

リストに追加された場合は 1 が返る

重複のためリストに追加されなかった場合は 0 が返る

```
127.0.0.1:6379> sismember set-key item4
(integer) 0
127.0.0.1:6379> sismember set-key item
(integer) 1
127.0.0.1:6379> srem set-key item2
(integer) 1
127.0.0.1:6379> srem set-key item2
(integer) 0
127.0.0.1:6379> smembers set-key
1) "item3"
2) "item"
```

item4 は存在しないため 0 が返る

item2 は存在しないため 0 が返る
Item2 が消えていることを確認

④Hash 型の保存イメージ

型	格納されている情報	例
HASH 型	順序のない、キーと値のハッシュテーブル	field1=1,field2=199 など項目名と値のペア

Hash 型

キー：ハッシュ名簿

項目：苗字
値：山田
値：田中

項目：名前
値：幸子
値：健司

Redis の型操作（Hash）

Hash 型の操作

＞hset hash-key sub-key1 value1

＞hset hash-key sub-key2 value2

＞hset hash-key sub-key1 value1

＞hgetall hash-key

＞hdel hash-key sub-key2

＞hdel hash-key sub-key2

＞hget hash-key sub-key1

＞hget hash-key sub-key2

＞hgetall hash-key

複数の項目にアクセスできることから、RDB の行に近い構造です

```
127.0.0.1:6379> hset hash-key sub-key1 value1
(integer) 1
127.0.0.1:6379> hset hash-key sub-key2 value2
(integer) 1
127.0.0.1:6379> hset hash-key sub-key1 value1
(integer) 0
127.0.0.1:6379> hgetall hash-key
1) "sub-key1"
2) "value1"
3) "sub-key2"
4) "value2"
```

Hash が追加できたら 1、できない場合は 0 が返る

Hash の全ての要素をフェッチ

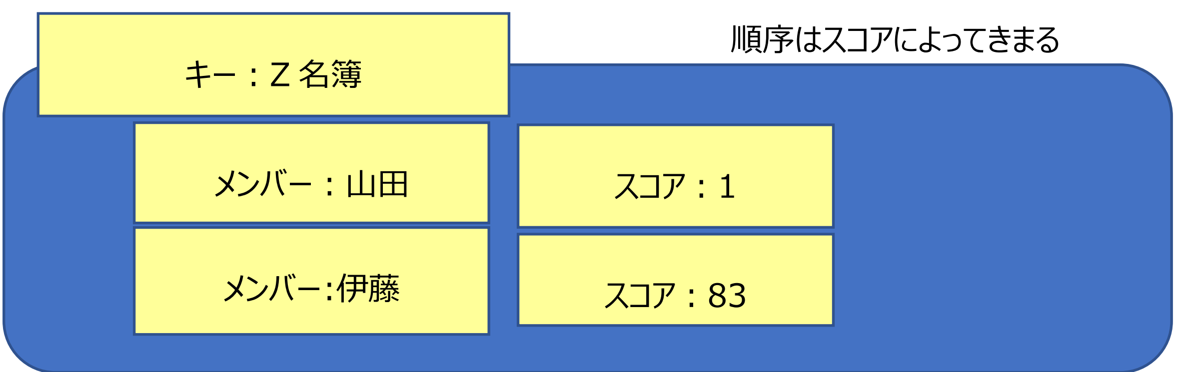
```
127.0.0.1:6379> hdel hash-key sub-key2
(integer) 1
127.0.0.1:6379> hdel hash-key sub-key2
(integer) 0
127.0.0.1:6379> hget hash-key sub-key1
"value1"
127.0.0.1:6379> hget hash-key sub-key2
(nil)
127.0.0.1:6379> hgetall hash-key
1) "sub-key1"
2) "value1"
```

Hash が削除できたら 1、できなかった場合は 0 が返る

削除済みの Hash を取得すると nil が返る

⑤Zset 型の保存イメージ

型	格納されている情報	例
ZSET 型 (ソート済み集合)	浮動小数点数のスコア順に並べられた文字列 メンバーからスコアへのマッピング	東京：スコア 1, 大阪：スコア 2, 名古屋：スコア 3 などスコアの順序と共に格納



Redis の型操作（Zset）

Zset 型の操作

```
>zadd zset-key 887 member1
>zadd zset-key 912 member0
>zadd zset-key 912 member0
>zrange zset-key 0 -1 withscores
 >zrangebyscore zset-key 0 900 withscores
 >zrange zset-key 0 900 withscores
 >zrem zset-key member1
 >zrem zset-key member1
 >zrange zset-key 0 -1 withscores
```

結果

```
127.0.0.1:6379> zadd zset-key 887 member1
(integer) 1
127.0.0.1:6379> zadd zset-key 912 member0
(integer) 1
127.0.0.1:6379> zadd zset-key 912 member0
(integer) 0
```

```
127.0.0.1:6379> zrange zset-key 0 -1 withscores
1) "member1"
2) "887"
3) "member0"
4) "912"
```

```
127.0.0.1:6379> zrangebyscore zset-key 0 900 withscores
1) "member1"
2) "887"
```

スコアの範囲で要素を取得できる

```
127.0.0.1:6379> zrem zset-key member1
(integer) 1
127.0.0.1:6379> zrem zset-key member1
(integer) 0
127.0.0.1:6379> zrange zset-key 0 -1 withscores
1) "member0"
2) "912"
```

参考：Python プログラムから Redis にアクセスする方法

Python から Redis を呼ぶためには Redis クライアントライブラリをインストールする必要があります。
インストールを簡単にするために、easy_install を使用します。

パイソンを起動
c:¥python27¥python

パッケージ ez_setup.py をネットから探し、ダウンロードする
from urllib import urlopen
data = urlopen('http://peak.telecommunity.com/dist/ez_setup.py')
open('ez_setup.py','wb').write(data.read())
パイソンを終了
exit()

ez_setup.py を起動

c:¥python27¥python ez_setup.py

Redis クライアントをインストール

c:¥python27¥python -m easy_install redis

実行画面

```
Microsoft Windows [Version 6.1.7601]
Copyright (c) 2009 Microsoft Corporation.  All rights reserved.

C:¥Users¥uni03>c:¥python27¥python
Python 2.7.13 (v2.7.13:a06454b1afa1, Dec 17 2016, 20:53:40) [MSC v.1500 64 bit (
AMD64)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> from urllib import urlopen
>>> data = urlopen('http://peak.telecommunity.com/dist/ez_setup.py')
>>> open('ez_setup.py','wb').write(data.read())
>>> exit()

C:¥Users¥uni03>c:¥python27¥python ez_setup.py
Setuptools version 0.6c11 or greater has been installed.
(Run "ez_setup.py -U setuptools" to reinstall or upgrade.)

C:¥Users¥uni03>c:¥python27¥python -m easy_install redis
Searching for redis
Reading https://pypi.python.org/simple/redis/
Downloading https://pypi.python.org/packages/09/8d/6d34b75326bf96d4139a2ddd8e74b
80840f800a0a79f9294399e212cb9a7/redis-2.10.6.tar.gz#md5=048348d8cfe0b5d0bba2f4d8
35005c3b
Best match: redis 2.10.6
Processing redis-2.10.6.tar.gz
Writing c:¥users¥uni03¥appdata¥local¥temp¥easy_install-gn6xmy¥redis-2.10.6¥setup
.cfg
Running redis-2.10.6¥setup.py -q bdist_egg --dist-dir c:¥users¥uni03¥appdata¥loc
al¥temp¥easy_install-gn6xmy¥redis-2.10.6¥egg-dist-tmp-ne7ilm
warning: no previously-included files found matching '__pycache__'
warning: no previously-included files matching '*.pyc' found under directory 'te
sts'
zip_safe flag not set; analyzing archive contents...
Moving redis-2.10.6-py2.7.egg to c:¥python27¥lib¥site-packages
Adding redis 2.10.6 to easy-install.pth file

Installed c:¥python27¥lib¥site-packages¥redis-2.10.6-py2.7.egg
Processing dependencies for redis
Finished processing dependencies for redis

C:¥Users¥uni03>_
```

プログラムを実行してみましょう。

C:¥Users¥uni03>c:¥python27¥python

Python 2.7.13 (v2.7.13:a06454b1afa1, Dec 17 2016, 20:53:40) [MSC v.1500 64 bit (AMD64)] on win32

Type "help", "copyright", "credits" or "license" for more information.

☆Redis をインポート

>>> import redis

☆Redis へのコネクション生成

>>> conn = redis.Redis()

☆コマンドをセットして送信

>>> conn.set('hello','world')

True

☆コマンドをセットして送信

>>> conn.get('hello')

'world'

Python プログラムの中でも同様に記述できますが、操作を簡単にするためにコマンドラインで実行しています。

```
al¥temp¥easy_install-gn6xmy¥redis-2.10.6¥egg-dist-tmp-ne7ilm
warning: no previously-included files found matching '__pycache__'
warning: no previously-included files matching '*.pyc' found under directory 'te
sts'
zip_safe flag not set; analyzing archive contents...
Moving redis-2.10.6-py2.7.egg to c:¥python27¥lib¥site-packages
Adding redis 2.10.6 to easy-install.pth file

Installed c:¥python27¥lib¥site-packages¥redis-2.10.6-py2.7.egg
Processing dependencies for redis
Finished processing dependencies for redis

C:¥Users¥uni03>:¥python27¥python
C:¥Users¥uni03>c:¥python27¥python
Python 2.7.13 (v2.7.13:a06454b1afa1, Dec 17 2016, 20:53:40) [MSC v.1500 64 bit (
AMD64)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> import redis
>>> conn = redis.Redis()
>>> conn.set('hello','world')
True
>>> conn.get('hello')
'world'
>>>
```

ビッグデータ
Spark 演習資料

Sparkとは

Spark は 2009 年にカリフォルニア大学バークレー校の AMPLab により、大規模データ処理のための分散処理理プラットフォームとして開発が開始され、2014 年に Apache Software Foundation に寄贈されました。

Spark は Hadoop にとって代わるものではなく、MapReduce にとって代わる存在です。
Hadoop が Java 言語で作られているのに対して Spark は Java の派生言語である Scala で作られています。

Spark の特徴

インメモリ処理による高速化

「データの格納場所」に関する選択肢の広さ

Hadoop Distributed File System (HDFS)

Cassandra

OpenStack Swift

Amazon S3

プログラム手法に関する選択肢の広さ

・Java

・Python

・R 言語

また、Spark は「Spark SQL」や DataFlow で SQL 言語を扱えます。

Spark と Hadoop の関係は共存関係

・「Hadoop+Spark」の構成も現実的である

(Hadoop 内の Yarn 制御下で Spark を利用する)

・Spark はデータの入出力場所として HDFS にも対応している

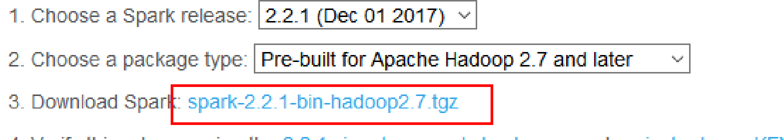
Spark のインストール

1．Java をインストールし、実行できるようにします。

2．Apache の Spark サイトにアクセスします

https://spark.apache.org/downloads.html

Download Apache Spark™



4. Verify this release using the 2.2.1 signatures and checksums and project release KEYS.

Note: Starting version 2.0, Spark is built with Scala 2.11 by default. Scala 2.10 users should download the Spark source package and build with Scala 2.10 support.

3．最新のバージョンの tgz をダウンロードし、
配置したい場所に解凍します。

HTTP

http://ftp.jaist.ac.jp/pub/apache/spark/spark-2.2.1/spark-2.2.1-bin-hadoop2.7.tgz

http://ftp.kddilabs.jp/infosystems/apache/spark/spark-2.2.1/spark-2.2.1-bin-hadoop2.7.tgz

http://ftp.meisei-u.ac.jp/mirror/apache/dist/spark/spark-2.2.1/spark-2.2.1-bin-hadoop2.7.tgz

http://ftp.riken.jp/net/apache/spark/spark-2.2.1/spark-2.2.1-bin-hadoop2.7.tgz

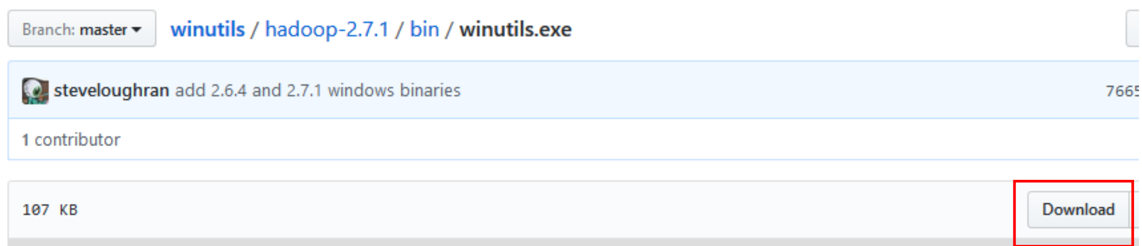
http://ftp.tsukuba.wide.ad.jp/software/apache/spark/spark-2.2.1/spark-2.2.1-bin-hadoop2.7.tgz

http://ftp.yz.yamagata-u.ac.jp/pub/network/apache/spark/spark-2.2.1/spark-2.2.1-bin-hadoop2.7.tgz

4．Windows で Spark を実行するために、Hadoop の依存関係を解決するための exe を
取得します。

winutils.exe

https://github.com/steveloughran/winutils/blob/master/hadoop-2.7.1/bin/winutils.exe



5．ダウンロードした exe ファイルを、展開後の Spark パッケージの bin の下に配置します。

6．環境変数の設定を行います。

- PATH: Spark インストール先¥bin を追加
- HADOOP_HOME: Spark インストール先

Spark コマンドラインの起動

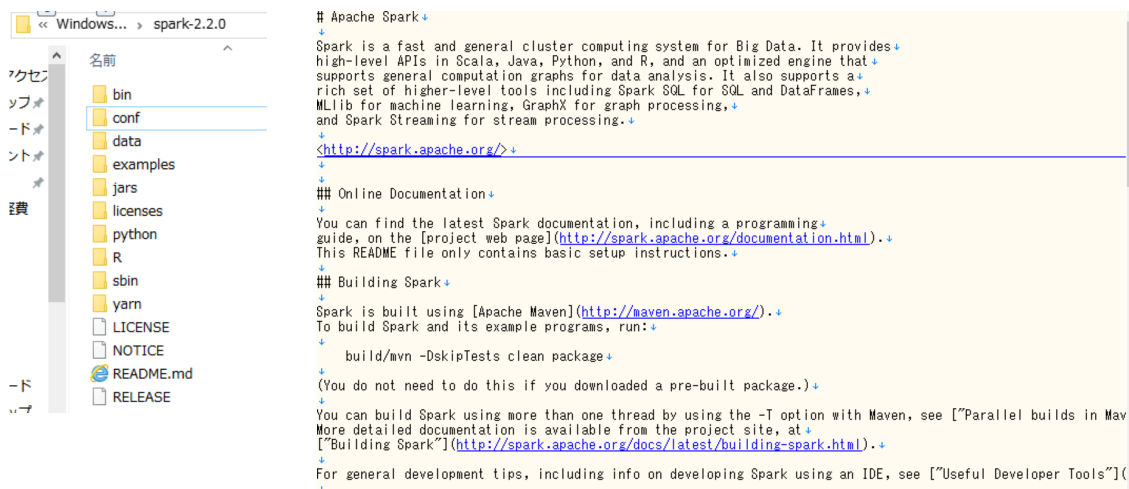
コマンド・プロンプトを開いて、spark-shell を実行します。

C:¥Users¥XXXX>spark-shell

①簡単なサンプルコードの起動

ReadMe に含まれる英単語をカウントしてみましょう。README.md ファイルは Spark インストールディレクトリの直下にあります。

テキストエディタで開いてみます



コマンドラインから以下を入力してみましょう。

val lines = sc.textFile("C:/spark-2.2.0/README.md")

```
scala> val lines = sc.textFile("C:/spark-2.2.0/README.md")
lines: org.apache.spark.rdd.RDD[String] = C:/spark-2.2.0/README.md MapPartitionsRDD[3] at textFile at <console>:24
```

1 コマンド目の val　は RDD 宣言を表します。textFile メソッドで　README.md ファイルを読み込み、lines という名前の RDD を宣言します。

lines.count()

```
scala> lines.count()
res0: Long = 103
```

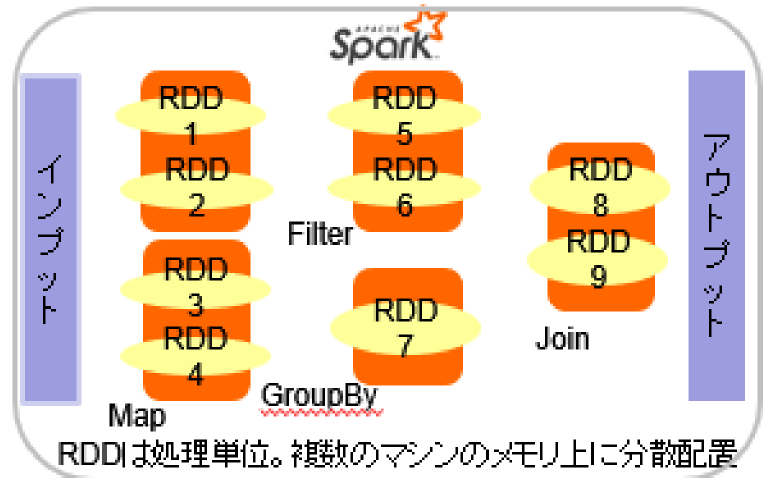
lines.first()

```
scala> lines.first()
res1: String = # Apache Spark
```

Count 関数で行数を、first 関数で最初の行を取得しています。

RDDとは

RDD(Resilient Distributed Dataset) はSparkの主要なデータの処理単位です。これは、クラスタ全体で並列化された要素の分散集合です。RDDは、YARNクラスタの複数の物理ノードのメモリ上に分割、分散された並列操作可能なオブジェクトの集まりです。



RDDを作成する方法は3つあります。

- 既存のデータ集合を並列化します。データがすでにSpark内に存在し、並列で操作できる状態であれば可能です。
- データセットを参照してRDDを作成します。 このデータセットは、HDFS、Cassandra、HBaseなどのHadoopでサポートされているすべてのストレージソースから取得できます。
- 既存のRDDを変換して新しいRDDを作成してRDDを作成します。

②簡単なサンプルコードの起動

RDDを加工して、スペース区切りにし、それをwordという(キー,1回)というMapに格納し、reduceByKeyメソッドで同じキーの出現回数を集計してみましょう

```
val wordCounts = lines.flatMap(line => line.split(" ")).map(word =>
(word,1)).reduceByKey((a,b) => a + b )
```

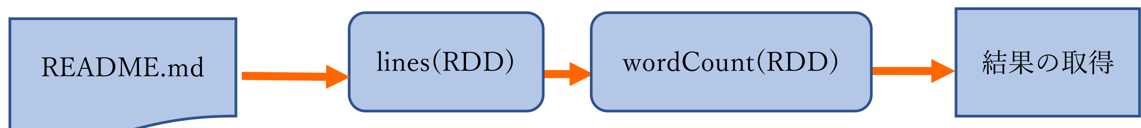
```
scala> val wordCounts = lines.flatMap(line => line.split(" ")).map(word => (word,1)).reduceByKey((a,b) => a + b )
wordCounts: org.apache.spark.rdd.RDD[(String, Int)] = ShuffledRDD[9] at reduceByKey at <console>:26
```

結果をcollectメソッドで取得してみます。

wordCounts.collect()

```
scala> wordCounts.collect()
res5: Array[(String, Int)] = Array((package,1), (this,1), (Version"](http://spark.apache.org/docs/latest/building-spark.html#sp
ecifying-the-hadoop-version),1), (Because,1), (Python,2), (page](http://spark.apache.org/documentation.html).,1), (cluster.,1),
 (its,1), ([run,1), (general,3), (have,1), (pre-built,1), (YARN,,1), (locally,2), (changed,1), (locally.,1), (sc.parallelize(1,
1), (only,1), (several,1), (This,2), (basic,1), (Configuration,1), (learning,,1), (documentation,3), (first,1), (graph,1), (Hiv
e,2), (info,1), (["Specifying,1), ("yarn",1), ([params]`.,1), ([project,1), (prefer,1), (SparkPi,2), (<http://spark.apache.org/
>,1), (engine,1), (version,1), (file,1), (documentation,,1), (MASTER,1), (example,3), (["Parallel,1), (are,1), (params,1), (sca
la>,1), (DataFrames,,1), (provides,...
```

Lines の RDD から、別の RDD を生成して利用しています。



③簡単なサンプルコードの起動

Scala という単語を含む行を探索し、結果を表示してみましょう。

val filterdLines = lines.filter(line => line.contains("Scala"))

```
scala> val filterdLines = lines.filter(line => line.contains("Scala"))
filterdLines: org.apache.spark.rdd.RDD[String] = MapPartitionsRDD[10] at filter at <console>:26
```

件数を表示します

filterdLines.count()

```
scala> filterdLines.count()
res6: Long = 3
```

filterdLines.first()

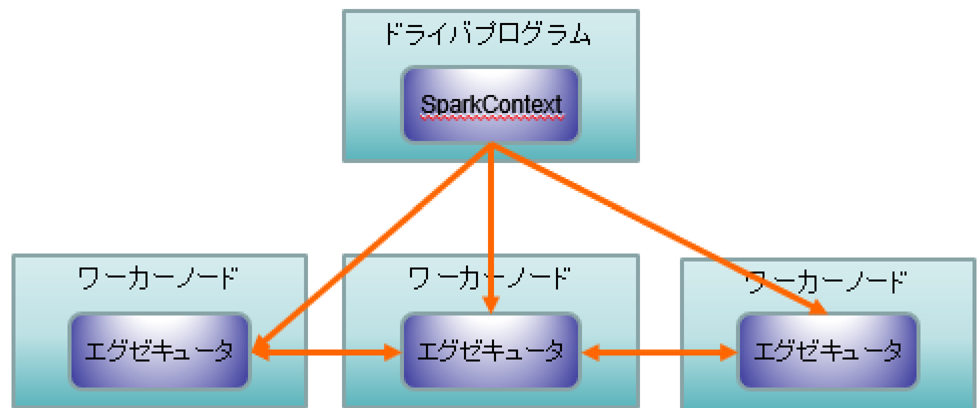
```
scala> filterdLines.first()
res7: String = high-level APIs in Scala, Java, Python, and R, and an optimized engine that
```

filterdLines.collect()

```
scala> filterdLines.collect()
res8: Array[String] = Array(high-level APIs in Scala, Java, Python, and R, and an optimized engine that, ## Interactive
hell, The easiest way to start using Spark is through the Scala shell:)
```

Spark の内部動作（ワーカーノードとエグゼキューター）

Spark アプリケーションは「Driver Program」によって全体の処理を制御します。これまでの RDD 生成や変換の処理を記述したものがドライバプログラムです。ドライバプログラムは実際に分散処理を実行する「Worker Node（ワーカーノード）」上の「Executer（エグゼキューター）」を管理しており、エグゼキューターがプログラム内容を実行します。



Spark の内部動作（SparkContext）

今回の演習では、コマンドラインからローカルマシン 1 台で処理を行っていますが、クラスタ環境で実行した場合は、同じドライバプログラムで複数のワーカーノードに分散されて実行されます。
分散実行を実現するために、SparkContext が用意されています。 SparkContext オブジェクトはそれぞれのマシンが管理する Spark へ接続し、実行します。
今回利用しているインタラクティブシェルでは、起動時に SparkContext が起動されています。

sc と入力して実行してみましょう。

```
scala> sc
res9: org.apache.spark.SparkContext = org.apache.spark.SparkContext@802d25
```

SparkContext が存在することが確認できます。

Sc.master でマスタ URL を調べると local[*]が指定されており、ローカルモードであることが分かります。

```
scala> sc.master
res10: String = local[*]
```

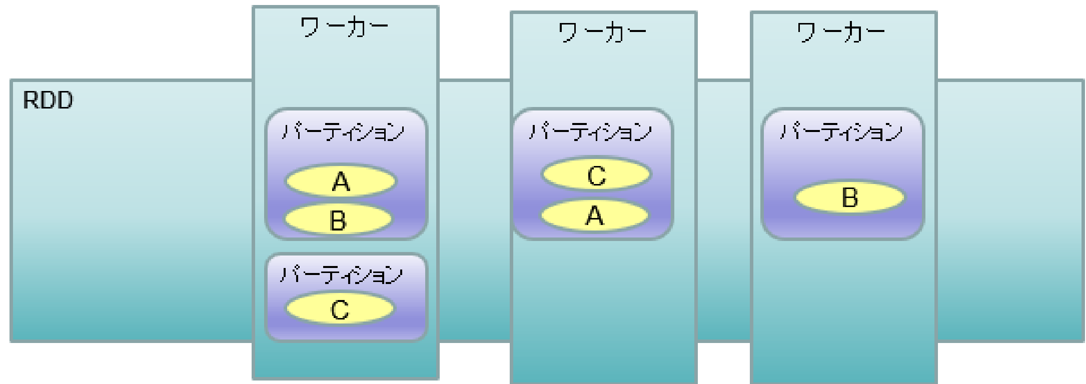
RDD の基本と構造

RDD（Resilient Distributed Dataset）

RDD はイミュータブルなデータ要素の分散コレクションです。

イミュータブルとは、一度作成したオブジェクトはその後状態を変えられない事を意味します。分散処理のため、どこかのワーカーで値を変えた場合、後続のワーカーに影響が出ることを避けるための制約です。

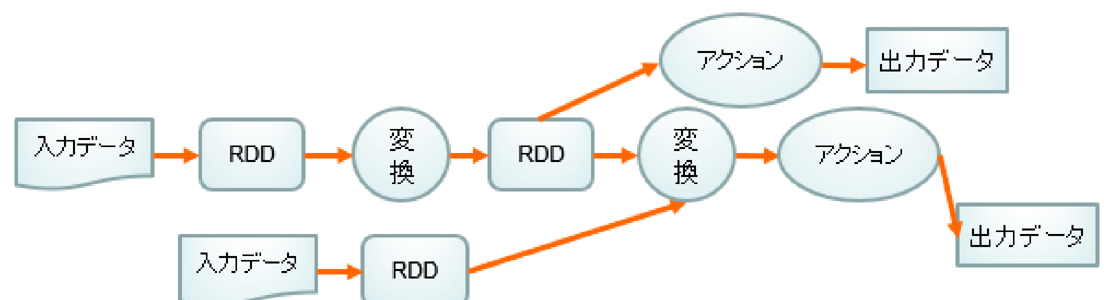
RDD はパーティションと呼ばれる単位に分割され、Spark クラスタの複数のノードに分散されて配置されます。パーティション単位で処理が行われるため、大量のデータでも複数のノードで分散して処理できます。



RDD の基本と構造

RDD（Resilient Distributed Dataset）

RDD はイミュータブルなデータ要素です。データの変更は RDD から新たな RDD の作成を繰り返す事で処理を実行します。



RDD の操作

RDD を「新規に」作成する方法は大きく分けて 2 種類あります。

① テキストファイルやデータベースなどの外部データソースからデータをロードする方法
ファイルの例

```
val lines = sc.textFile("C:/spark-2.2.0/README.md")
```

【参考】Hive の例：コンテキストを生成し、sql メソッドで SQL クエリを発行します。

```
val hiveContext = new org.apache.spark.sql.hive.HiveContext(sc)
hiveContext.sql("show tables").collect.foreach(println)
```

以下は Zeppelin Notebook 上での実行結果です

```
%spark
hiveContext.sql("show tables").collect.foreach(println)
```

FINISHED

```
[avg_mileage_user001,false]
[drivermileage_user001,false]
[geolocation,false]
[geolocation_user001,false]
[health_table,false]
[truck_mileage_user001,false]
[trucks_user001,false]
```

Took 1 sec. Last updated by anonymous at February 15 2017, 7:57:19 PM.

② Driver Program で Array などのプログラミング言語が用意する通常のコレクションから作成する。コレクションから RDD を作成するメソッドとして SparkContext の parallelize メソッドを使用します。

```
val nums = sc.parallelize(Array(3,5,3,2,1))
```

```
scala> val nums = sc.parallelize(Array(3,5,3,2,1))
nums: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[11] at parallelize at <console>:24
```

nums に配列を生成して値を代入した状態です。

parallelize の第 2 引数では、どのパーティションで作成するかを明示的に指定することもできます。ローカルの場合は、CPU のコア数になります。

```
val numsPar2 = sc.parallelize(Array(3,5,3,2,1),2)
```

RDD の操作には、変換（Transformation）とアクション（Action）の 2 種類があります。

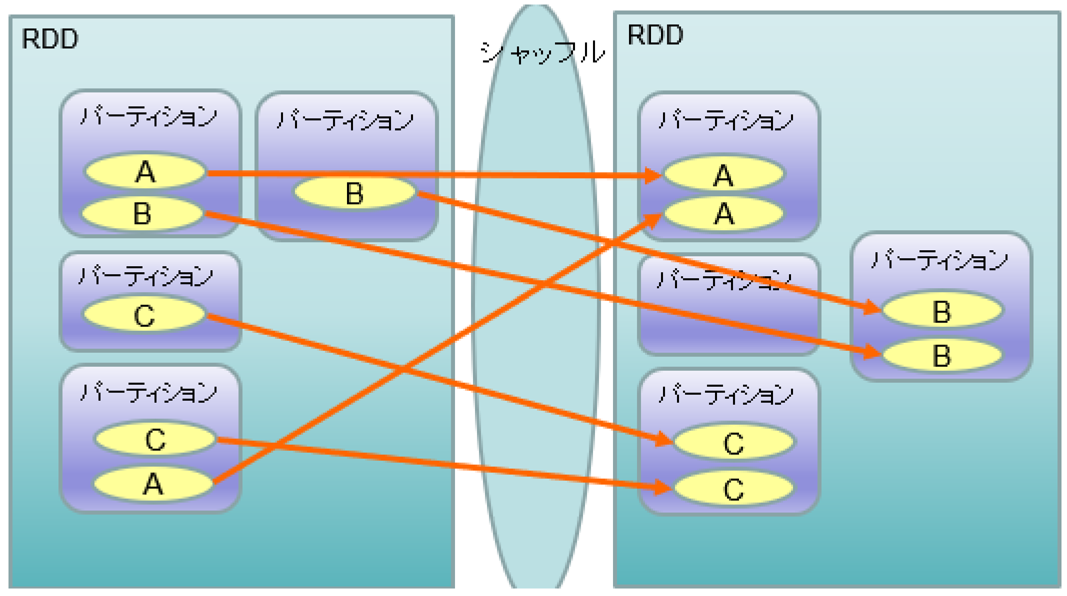
変換：RDD に処理を加えて新しい RDD を作成していく操作。map や filter など、一般的なプログラミング言語におけるコレクションへの操作のようなものです。

アクション：ワーカーでの処理結果をまとめ上げて、Scala での Array など通常のコレクションに変換したり、HDFS や AmazonS3 などの外部ストレージに値を保存したりする操作。

シャッフル

フィルターのようにワーカー内で完結する操作と、集計操作のようにデータを集めなければならない操作があります。その場合は Spark 内でシャッフル（Shuffle）という機能が働き、異なるパーティションに含まれる同一キーの要素を同一のパーティションにまとめる処理を行います

シャッフルのイメージ　Spark 裏側で動きますが、気が付かないところで大量のシャッフルやネットワーク間通信が発生する場合もあります。フィルターをかけてから集計をするなどの配慮が必要です。



RDD の操作（遅延評価）

遅延評価・・・実際に計算する必要が生じるまで値を評価しない処理方式。

具体的にはアクションが実行されるまでは、変換操作は記録だけ行われるものの、実行自体は何も行われません。
アクションが実行された時点で RDD の変換を逆にたどって、まとめて処理されます。

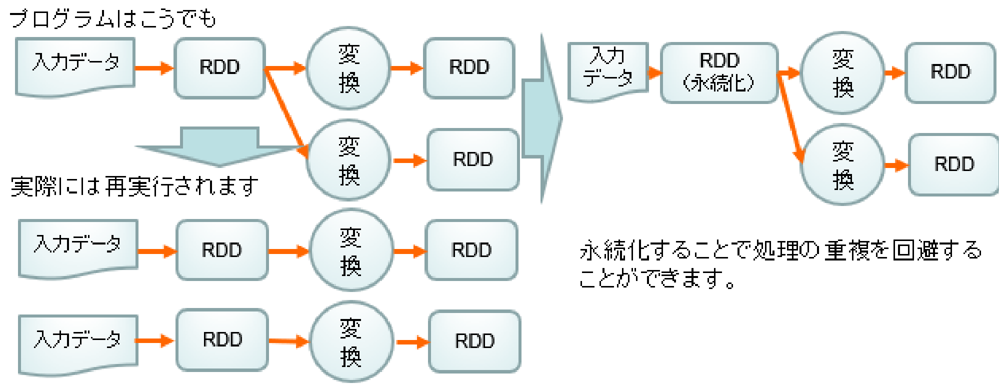
Spark がメモリ上に大量のデータを乗せる仕組みであるため、不要なデータをメモリに保持したり、不要な処理を行ったりすることを回避することは重要な要素です。本当に必要な処理にリソースを集中させるための仕組みとなります。

RDD の処理には処理 DAG（ Directed acyclic graph：有向非循環グラフ）を構築します。
DAG は循環しない向きのある関係性で、終点から遡ると必ず一本道で始点へと到達できます。

DAG のイメージ　向きはあるが起点に戻らない

RDD の操作（永続化）

永続化　一度作成された RDD を再利用する場合に再度データの読み直しや再処理を
避けるため、RDD を明示的に永続化する事が可能です。
永続化する先はメモリの他、ディスクも選択可能、レプリケーション数や
シリアライズ方法の指定もできます。
シャッフル前の RDD は暗黙的にローカルファイルに永続化されます。



RDD の操作（RDD と DAG の関係）

Spark では RDD へのアクションが実行されると、論理実行プランとして RDD の DAG（有向非循環グラフ）を作成します。次に Spark は DAG を元に物理的な実行プランを構築します。
ノード間通信が発生するシャッフルを境界にして複数のステージに分割します。シャッフルが発生しない処理はなるべく一つのステージにグルーピングします。
グルーピングされたステージから、RDD のデータ配置とパーティションに基づいて構築される処理単位がタスクになります。
実行プランを toDebugString メソッドで確認します。

wordCounts.toDebugString
上記を実行すると以下のように表示されます。

```
scala> wordCounts.toDebugString
res11: String =
(2) ShuffledRDD[6] at reduceByKey at <console>:26 []
 +-(2) MapPartitionsRDD[5] at map at <console>:26 []
    |  MapPartitionsRDD[4] at flatMap at <console>:26 []
    |  C:/spark-2.2.0/README.md MapPartitionsRDD[3] at textFile at <console>:24 []
    |  C:/spark-2.2.0/README.md HadoopRDD[2] at textFile at <console>:24 []
```

同じインデントが同一のステージです。インデントが変わるとステージが変わる事を
意味します。

基本の変換

map

map メソッドは、RDD の各要素に対して引数で与えられた関数を適用し、新しい RDD を返却します。新しい RDD の要素型は元の RDD のものと同一である必要はありません。結果値は配列やリストになります。単語の RDD から、文字長の RDD を生成してみましょう。

```
val address = sc.parallelize(Array("Tokyo","saitama","kanagawa","chiba"))
address.map(address => address.length).collect()
```

```
scala> val address = sc.parallelize(Array("Tokyo","saitama","kanagawa","chiba"))
address: org.apache.spark.rdd.RDD[String] = ParallelCollectionRDD[16] at parallelize at <console>:24
```

```
scala> address.map(address => address.length).collect()
res21: Array[Int] = Array(5, 7, 8, 5)
```

flatMap

結果値をリストではなく、フラットな状態で返します。

```
val lines = sc.parallelize(Array("Apple is red", "PineApple is yellow"))
lines.flatMap(line => line.split(" ")).collect()
```

```
scala> val lines = sc.parallelize(Array("Apple is red", "PineApple is yellow"))
lines: org.apache.spark.rdd.RDD[String] = ParallelCollectionRDD[21] at parallelize at <console>:24
```

```
scala> lines.flatMap(line => line.split(" ")).collect()
res22: Array[String] = Array(Apple, is, red, PineApple, is, yellow)
```

filter

filter メソッドは、RDD の各要素をフィルタリングし、条件に合致した要素を新しい RDD として返却します。条件は Boolean 型を返却する関数として filter メソッドの引数に渡します。
次の例では先頭の文字が T の値だけを抽出します。

```
val address = sc.parallelize(Array("Tokyo", "saitama", "kanagawa", "chiba"))
address.filter(address => address.startsWith("T")).collect()
```

```
scala> val address = sc.parallelize(Array("Tokyo", "saitama", "kanagawa", "chiba"))
address: org.apache.spark.rdd.RDD[String] = ParallelCollectionRDD[23] at parallelize at <console>:24

scala> address.filter(address => address.startsWith("T")).collect()
res24: Array[String] = Array(Tokyo)
```

distinct

distinct メソッドは要素の重複を除外した RDD を返します。データが複数ノードに渡る場合はシャッフルが発生します。

```
val address = sc.parallelize(Array("Tokyo", "saitama", "saitama", "Tokyo"))
address.distinct().collect()
```

```
scala> val address = sc.parallelize(Array("Tokyo", "saitama", "saitama", "Tokyo"))
address: org.apache.spark.rdd.RDD[String] = ParallelCollectionRDD[25] at parallelize at <console>:24

scala> address.distinct().collect()
res25: Array[String] = Array(saitama, Tokyo)
```

zip

zip メソッドは、RDD を引数にとり、それぞれの要素のペア RDD を作成します。

```
val address1 = sc.parallelize(Array("Tokyo","saitama","kanagawa","chiba"))
val address2 = sc.parallelize(Array("東京", "埼玉", "神奈川", "千葉"))
address1.zip(address2).collect()
```

```
scala> val address1 = sc.parallelize(Array("Tokyo","saitama","kanagawa","chiba"))
address1: org.apache.spark.rdd.RDD[String] = ParallelCollectionRDD[29] at parallelize at <console>:24

scala> val address2 = sc.parallelize(Array("東京", "埼玉", "神奈川", "千葉"))
address2: org.apache.spark.rdd.RDD[String] = ParallelCollectionRDD[30] at parallelize at <console>:24

scala> address1.zip(address2).collect()
res26: Array[(String, String)] = Array((Tokyo,東京), (saitama,埼玉), (kanagawa,神奈川), (chiba,千葉))
```

union

union は要素の型が同じ RDD を結合します。重複の削除や、パーティションの統合などは行われません。

```
val address1 = sc.parallelize(Array("Tokyo","saitama","kanagawa","chiba"))
val address2 = sc.parallelize(Array("東京", "埼玉", "神奈川", "千葉"))
address1.union(address2).collect()
```

```
scala> address1.union(address2).collect()
res27: Array[String] = Array(Tokyo, saitama, kanagawa, chiba, 東京, 埼玉, 神奈川, 千葉)
```

collect

collect メソッドは RDD の全ての要素をドライバプログラムに Array 型で返します。全てのパーティションの RDD を返すため、実行には注意が必要です。

```
address.collect()
```

foreach

foreach メソッドは各要素に引数で与えた関数を適用します。戻り値はありません。

```
val address = sc.parallelize(Array("Tokyo","saitama","kanagawa","chiba"))
address.foreach(println)
```

```
scala> address.foreach(println)
saitama
saitama
Tokyo
Tokyo
```

count

count メソッドは RDD の要素数を返します。

```
val address = sc.parallelize(Array("Tokyo","saitama","kanagawa","chiba"))
address.count()
```

```
scala> address.count()
res29: Long = 4
```

top　, takeOrdered

top メソッドは最も値が大きい順に引数分だけ値を返します（つまり降順）。

takeOrdered は昇順で値を返します。

```
val nums = sc.parallelize(Array(3,2,4,3,2,1))
nums.top(3)
nums.takeOrdered(3)
```

```
scala> val nums = sc.parallelize(Array(3,2,4,3,2,1))
nums: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[36] at parallelize at <console>:24
```

```
scala> nums.top(3)
res40: Array[Int] = Array(4, 3, 3)
```

```
scala> nums.takeOrdered(3)
res43: Array[Int] = Array(1, 2, 2)
```

reduce

reduce は RDD の要素を集約します。集約の方法はメソッドの引数に関数を渡します。

数値の合計の例を示します。パーティションは 2 に設定しています。

```
val nums = sc.parallelize(Array(3,2,4,3,2,1),2)
nums.reduce((x,y)=>x+y)
```

```
scala> val nums = sc.parallelize(Array(3,2,4,3,2,1),3)
nums: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[39] at parallelize at <console>:24

scala> nums.reduce((x,y)=>x+y)
res45: Int = 15
```

ビッグデータ
ビッグデータ活用の
プランニング

演習 ビッグデータ活用プランニング

　あなたは以下の企業の情報企画室長です。ビックデータを活用した何らかの改善活動、システム構築などを企画・立案ください。

社　　　名	株式会社　スポーツワン
本　　　社	東京都渋谷区
代　表　者	山田　大輔（52歳）
設　　　立	1973年 6月 8日
資　本　金	100億円
従 業 員 数	正社員：1000名 パートタイマー、アルバイト：2032名
店　舗　数	100店舗　フラグシップ店舗　10店舗
売　上　高	3790億円／年（単体）
主な事業内容	スポーツ用品、ウェア製造・販売

当社を取り巻く状況

創業時期はゴルフ用品、スキー用品など高付加価値の商材を独自開発、北関東を拠点に全国に専門店を出店。現在は総合スポーツ用品店として100店舗を展開。業界5位の規模。楽天、アマゾンに対抗しECサイトを立ち上げたが、リアル店舗を補完するほどの売り上げには達しておらず、リアル店舗との連動も進んでいない。スマホアプリも開発済みでリアル店舗から一定量の顧客の利用を実現したが、売上増進の牽引役にはなっていない。

EC、スマホアプリ共に購買履歴や行動ログを取得する準備はできているものの、具体的な施策がないため、活用されていない。

リアル店舗の売り上げはID-POSによって管理されており、センターにデータが集まってきてはいるものの戦略的活用はされていない。

関東に比べると、関西九州地区への進出は進んでおらず、今後進出する地

域として有力である。創業時に開発した商材のシリーズは今もマニアからは愛好され、静かなブームを呼んでいるとの Twitter 上の噂がある。
スポーツ分野ごとの季節性は、感触として皆持っているが販売店への施策として打ち出されていない。
競合店の位置を加味した出店計画はしておらず、競合の新規キャンペーンによる売上変化も分析されていない。
各支店からのレポートは週次、月次の Excel レポート。

ポイント
- 社内に存在するビッグデータは、現時点で現存するか、新規に取得できるものとして、常識の範囲内で自由に設定してください。
- オープンデータは、現存すると予想されるものを自由に設定してください。
- 社外購入データは、予算の範囲内で調達可能と考えてください。
- 取りえるビッグデータ活用企画を立案し、中期計画の形で立案ください。プロジェクト期間は 2 年以内です。

「企画の背景・目的」、「現状」、「要因分析」、「プラン」、
「スケジュール」の形でまとめてください。

模造紙等にまとめる内容

グループ名	
リーダー	
メンバー	
プロジェクト名	
背景・目的	本企画が立案された課題背景と目指すゴール、目的
現状	背景に記載された課題を記載
要因分析	課題・原因・対策候補を記載
プラン	プロセス・体制・投資対効果・利用データ・利用ツール 開発内容、新規サービス内容など
スケジュール	立案から実施までの大まかなスケジュールを記載