2.1 가상머신의 사용

시작하기 전에..

> **가상 머신 시작**- `vagrant up`
>
> **리눅스 or 맥 경우**- `$ vagrant ssh -- -l spark`
>
> **윈도우의 경우** -- Putty, Kitty, MobaXterm등 사용

2.1.1 깃허브 저장소 복제

(git은 가상머신에 미리 설치되어 있음)

```
$ git clone https://github.com/spark-in-action/first-edition
```

/home 아래에 first-edition 폴더가 생기면 완성

2.1.2 자바 찾기

```
$ which java
```

> /user/bin/java

위의 주소는 **심볼릭 링크(Symbloic Link)** 이며 실제로 자바가 설치된 위치는 다음과 같이 확인할 수 있다.

```
spark@spark-in-action:~$ ls -la /usr/bin/java
lrwxrwxrwx 1 root root 22 Apr 19  2016 /usr/bin/java -> /etc/alternatives/java
spark@spark-in-action:~$ ls -la /etc/alternatives/java
lrwxrwxrwx 1 root root 46 Apr 19  2016 /etc/alternatives/java ->
/usr/lib/jvm/java-8-openjdk-amd64/jre/bin/java
```

하둡과 스파크를 실행하기 위해 필요한 JAVA_HOME 환경변수는 다음과 같이 확인 가능

```
spark@spark-in-action:~$ echo $JAVA_HOME
/usr/lib/jvm/java-8-openjdk-amd64/jre
```

2.1.3 가상머신에 설치된 하둡 사용

HDFS의 파일을 읽고 쓰거나 YARN을 실행하기 위해서는 하둡이 필요하다.

- 하둡의 심볼릭 링크 - `/usr/local/hadoop`
- 실제 하둡 바이너리 - `/opt/hadoop-2.7.2`

> *HDFS 셸 명령은 `hadoop fs` 로 시작하며 하둡의 공식 문서 (http://hadoop.apache.org/docs/current/hadoop-project-dist/hadoop-common/FileSystemShell.html) 에서 HDFS 명령의 전체 목록 참조 가능*

- HDFS 데몬 프로세스 시작 - `$ /usr/local/hadoop/sbin/start-dfs.sh`

  (가상머신이 부팅하며 자동 시작)

- HDFS 데몬 프로세스 종료 - `$ /usr/local/hadoop/sbin/stop-dfs.sh`

2.1.4 가상머신에 설치된 스파크 살펴보기

- 스파크 심볼릭 링크 - `/opt/local/spark`
- 실제 스파크 바이너리 - `/opt/spark-2.0.0-bin-hadoop2.7`

> *스파크 공식 내려받기 페이지 - https://spark.apache.org/downloads.html*

2.2 스파크 셸로 첫 스파크 프로그램 작성

스파크를 사용하는 방법에는 두가지가 있다.

1. **스파크 라이브러리, 즉, 스파크 API를 사용해 Scala, Java, Python 독립형 프로그램 작성**
2. **스파크의 스칼라 셸 혹은 파이선 셸 사용하기**

스파크 셸을 스파크 REPL(Read-Eval-Print Loop)이라고도 한다.

> ***read*** *읽고*
>
> ***Evaluate*** *계산한 결과를*
>
> ***print*** *출력하는 과정을*
>
> ***loop*** *반복한다.*

2.2.1 스파크 셸 시작

- 스파크 셸 시작 - `$ spark-shell`

2.2.2 첫 스파크 코드 예제

스파크 루트 디렉토리의 LICENSE 파일중, BSD(Berkely Software Distribution)의 라이센스로 등록된 라이브러리가 몇개나 존재하는지 찾아보는 예제

BSD라이센스로 공개된 라이브러리 정보를 기록한 줄에는 BSD 문자열이 항상 존재한다.

- **스파크 API를 활용해 LICENSE 파일을 읽어들인 후 파일에 포함된 줄 갯수 세기**

```
scala> val licLines = sc.textFile("/usr/local/spark/LICENSE")
// sc - Spark Context
scala> val lineCnt =  licLines.count
```

- **BSD 문자열이 등장한 줄 개수 계산하기**
  - **익명함수 활용**

```
  scala> val bsdLines = licLines.filter(line => line.contains("BSD"))
  // filter를 통해 익명함수에서 true로 판별된 요소만으로 구성된 새로운 컬렉
션 반환
  scala> val bsdCnt = bsdLines.count
```

- **기명함수 활용**

```
  scala> def isBSD(line : String) = {line.contains("BSD")}
  scala> val bsdLines = licLines.filter(line => isBSD(line))
```

- **표현식 활용**

```
  scala> val isBSD = (line : String) => line.contains("BSD")
  scala> val bsdLines1 = licLines.filter(isBSD)
```

- **각 줄별로 출력하기**

```
  bsdLines.foreach(bLine => println(bLine))

  bsdLines.foreach(println)
```

2.2.3 RDD의 개념

> *앞으로 다룰 RDD의 내용을 보강해주는 링크를 첨부합니다.*
>
> *https://12bme.tistory.com/306*

RDD는 스파크의 기본 추상화 객체로 다음과 같은 성질을 지닌다.

- **불변성(immutable)** - 읽기 전용
  - 한번 생성된 RDD 객체는 절대 바뀌지 않는다. RDD에서 제공되는 변환 연산자는 항상 새로운 RDD 객체를 생성한다. 컬렉션의 불변성은 분산시스템에서 가장 중요한 장애 내성을 직관적인 방법으로 보장할 수 있다.
- **분산(distributed)** - 노드 한 개 이상에 저장된 데이터 셋
  - 한개의 데이터 컬렉션을 여러 머신에 분산 저장하나, RDD라는 논리적인 개념으로 이러한 분산 데이터를 다룰 수 있는 투명성을 제공한다.
- **복원성(resilient)** - 장애 내성
  - 스파크에 내장된 장애복구 메커니즘은 RDD에 복원성을 부여한다. 즉, 스파크는 데이터를 분산해 저장하고 있는 노드에 장애가 발생하더라도 유실된 RDD를 원래대로 복구할 수 있다.

- RDD는 데이터를 중복저장하지 않는 대신, 데이터셋을 만드는데 사용된 변환 연산자의 로그를 남기는 방식으로 장애 내성을 제공한다.

2.3 RDD의 기본 행동 연산자 및 변환 연산자

RDD연산자는 크게 **변환(Transformation)** 과 **행동(Action)** 이라는 두개의 유형으로 나뉜다.

- **변환 연산자 (TranFormation)**

  RDD의 데이터를 조작해 새로운 RDD를 생성 (ex. filter, map)

- **행동 연산자 (Action)**

  연산자를 호출한 프로그램으로 계산결과를 반환하거나 RDD 요소에 특정 작업을 수행하려고 실제 계산을 시작하는 역할 (ex. count, foreach)

2.3.1 map 변환 연산자

원본 RDD의 각 요소를 변환한 후 변환된 요소로 새로운 RDD를 생성

앞서 살펴본 filter와는 달리, map함수가 호출된 RDD의 타입은 map함수가 반환하는 RDD의 타입과 같을 수도, 다를 수도 있다.

- **map함수를 사용해 RDD에 포함된 각 요소의 제곱값 구하기**

```
scala> val numbers = sc.parallelize(10 to 50 by 10)

// parallelize - Seq 객체를 받아 새로운 RDD 생성

scala> numbers.foreach(x => println(x))

scala> val numbersSquared = numbers.map(num => num*num)

scala> numbersSquared.foreach(println)
```

- **정수형타입의 RDD를 문자열타입의 RDD로 변환 후 문자열 순서 뒤집기**

```
scala> val reversed =  numbersSquared.map(x => x.toString.reverse)

scala> reversed.foreach(println)
```

더 간결하게 작성하면 다음과 같이 쓸 수 있다.

```
scala> val alsoReversed = numbersSquared.map(_.toString.reverse)

//스칼라에서 `_`는 플레이스 홀더라고 하며, 함수에 전달될 파라미터를 대체하는 역할을 한다.
```

> *기타문법 참고*
>
> ```
> // RDD의 첫번째 요소 리턴
> scala> alsoReversed.first
> res0: String = 001
>
> // 값이 가장 큰 요소를 내림차순으로 정렬후 리턴
> // 현 예제에서 배열의 요소 타입은 String이라는 점에 유의
> scala> alsoReversed.top(4)
> res1: Array[String] = Array(009, 0061, 0052, 004)
> ```

2.3.2 distinct와 flatMap 변환 연산자

- **실습 파일 생성**

```
$ echo "15,16,20,20
77,80,94
94,98,16,31
31,15,20" > ~/client-ids.log
```

- **데이터 split하기**

```
scala> val lines = sc.textFile("/home/spark/client-ids.log")

scala> val idsStr = lines.map(line => line.split(","))

scala> idsStr.foreach(println)
// [Ljava.lang.String;@2c52ddd4
// [Ljava.lang.String;@79820d78
// [Ljava.lang.String;@4451028c
// [Ljava.lang.String;@1fb014ad
```

`split()`함수는 `,`통해 분할한 파일을 각각의 줄 단위로 RDD배열 객체를 생성해 RDD 2차원 배열을 리턴한다.

```
scala> idsStr.first
res5: Array[String] = Array(15, 16, 20, 20)
```

- **collect 행동연산자 활용으로 리스트 배열의 리스트 확인하기**

```
scala> idsStr.collect
```

collect는 새로운 배열을 생성한 후, 호출한 RDD의 모든 요소를 이 배열에 모아 리턴한다.

- **flatmap을 활용해 단일배열로 분해하기**

주어진 함수를 모든 RDD의 요소에 적용한다는 점에서 map과 동일하나, 익명함수가 반환한 배열의 중첩구조를 한 단계 제거하고, 모든 배열의 요소를 단일 컬렉션으로 병합한다.

```
scala> val ids = lines.flatmap(_.split(","))
scala> ids.collect
```

- **mkString을 활용해 배열의 요소 하나로 연결하기**

`mkString`은 스칼라의 표준 라이브러리로 제공되는 Array 클래스의 메서드이다. 메서드에 전달한 인수는 연결한 문자열의 구분자로 사용된다.

```
scala> ids.collect.mkString(";")
res2: String = 15;16;20;20;77;80;94;94;98;16;31;31;15;20
// 하나의 문자열로 병합
```

- **toInt로 Int타입으로 변환하기**

```
    scala> val intIds = ids.map(_.toInt)
```

- **distinct로 중복값 제거하기**

```
    scala> val uniqueIds = intIds.distinct
    scala> uniqueIds.collect
    scala> val finalCnt = uniqueIds.count
```

2.3.3 sample, take, takeSample 연산으로 RDD의 일부 요소 가져오기

- **sample 메서드를 활용해 무작위 샘플링하기**

> ***sample(withReplacement, fraction, seed)*** **1) withReplacement(boolean)** - 같은 요소를 여러 번 샘플링 할 수 있는지 유무 지정되며 false일 경우 비복원 샘플링이 된다. **2) fraction(Double)** - 복원샘플링의 경우 각 요소가 샘플링될 횟수의 기대값을 의미(0 이상)하며, 비복원 샘플링의 경우 각 요소가 샘플링될 기대확률(0~1 사이의 부동소수점 숫자)을 의미한다. 주의할 점은 fraction이 반드시 해당 결과를 보장하는 것은 아니라는 점이다. **3) seed(Long)** - 난수생성에 사용

앞선 예제의 uniqueIds에 요소중 30%(fraction: 0.3)을 비복원 샘플링 해보자

```
    scala> val s = uniqueIds.sample(false, 0.3)
    scala> val s.collect
```

```
// res6: Array[Int] = Array(80, 20, 94, 15)
```

다음은 복원샘플링을 통해 위의 결과와 비교해보자.

```
scala> val swr = uniqueIds.sample(true,0.5)
scala> swr.collect
//res7: Array[Int] = Array(80, 80, 20, 31, 31)
```

- **takesample을 활용해 확률값 대신 정확한 값으로 샘플링하기**

사용법은 sample과 거의 비슷하나 두번째 인자인 `fraction`이 정수값이라는 점에서 차이가 존재한다. 즉, 요소가 샘플링될 기대확률이 아닌 샘플링하고자 하는 요소의 개수를 직접 명시한다는 점에서 차이가 존재한다.

또한, sample은 변환 연산자인 반면, takesample은 collect와 같이 배열을 반환하는 행동연산자이다.

```
scala> val take = uniqueIds.takeSample(false, 5)
take: Array[Int] = Array(15, 16, 31, 94, 77)
```

- **take를 활용해 데이터의 하위 집합 가져오기**

take역시 행동 연산자로서, 지정된 개수의 요소를 모들 때 까지 RDD의 파티션(클러스터의 여러 노드에 저장된 데이터의 일부분)을 하나씩 처리해 결과를 반환한다. RDD의 데이터를 살짝 엿보는 데 자주 사용되므로 잘 기억해두자.

2.4 Double RDD 전용 함수

Double 객체만으로 RDD를 구성하면 암시적변환을 사용할 수 있다.

2.4.1 Double RDD 함수로 기초 통계량 계산

- **mean 평균구하기**

```
scala> intIds.mean
```

- **sum 더하기**

```
scala> intIds.sum
```

- **stats 기초통계량 구하기**

```
scala> intIds.stats
```

- **variance 분산구하기**

```
scala> intIds.variance
```

- **stdev 표준편차 구하기**

```
scala> intIds.stdev
```

2.4.2 히스토그램으로 데이터 분포 시각화

- **설정구간에 속한 요소의 개수 반환받기**

```
scala> intIds.histogram([1.0 , 50.0, 100.0])
```

배열은 오름차순 정렬되어 있어야 하며, 반드시 두개 이상의 요소로 구성되어 있어야 한다.

- **데이터의 전체 범위를 균등하게 나누기**

```
scala> intIds.histogram(3)
res17: (Array[Double], Array[Long]) = (Array(15.0, 42.66666666666667,
70.33333333333334, 98.0),Array(9, 0, 5))
```

반환된 첫번째 배열은 계산된 구간경계의 위치를 알려주는 배열이며, 두번째 배열은 각 구간에 속한 요소 개수가 저장된 배열이다.

근사합계 및 평균계산

- **sumApprox**
- **meanApprox**

위의 두 메서드는 지정된 제한시간동안 근사합계 또는 근사 평균을 계산한다. 두 메서드는 동일한 파라미터를 인자로 받는데 이는 다음과 같다.

1. **timeout**: Long - 작업이 수행되는 최대 시간을 지정한다. 만약 입력한 시간이 초과할 경우, 해당 시간까지 계산된 중간 결과를 반환한다.
2. **confidence**: Double - 반환될 결과값에 영향을 준다.

두 메소드의 리턴값은 PartialResult이며 다음 두개의 필드로 구성되어 있다.

1. **finalValue** - 단일결과값이 아닌, 값의 확률범위(하한 및 상한), 평균값, 신뢰 수준을 제공한다.
2. **failure** - 예외가 발생했을 때만 Exception 객체를 반환한다.