Spark2 프로그래밍

RDD Action

김철환

Overview

❖ RDD 액션

① 드라이버 프로그램에 값을 리턴하는 메서드
② RDD의 트랜스포메이션 메서드를 호출하여도
액션 메서드를 호출해야 트랜스포메이션 연산이 순차적으로 시작
③ 액션 메서드를 여러 번 호출하면 트랜스포메이션 메서드도 여러 번 실행
④ 반복 수행 성능을 개선하기 위해 캐시를 적절히 사용
⑤ 코드 작성 시 반복 수행 가능성을 염두하고
이에 따른 오류가 발생하지 않도록 코드 작성

first

❖ **RDD 요소 가운데 첫 번째 요소 하나를 돌려줌 (트랜스포메이션 수행 결과 등을 확인)**

```
scala> val rdd = sc.parallelize(List(5, 4, 1))
rdd: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[0] at parallelize at <console>:24

scala> val result = rdd.first
result: Int = 5

scala> println(result)
5
```

- ✓ Parallelize : 데이터셋을 넘겨주고 RDD를 생성
  Spark Context 객체
- ✓ 5, 4, 1 중에 첫 번째 요소인 5을 반환
- ✓ 트랜스포메이션의 수행 결과 등을 빠르게 확인하는 용도로 활용

take

❖ **RDD의 첫 번째 요소로부터 순서대로 n개를 추출해서 되돌려 주는 메서드**

```
scala> val rdd = sc.parallelize(1 to 20, 5)
rdd: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[1] at parallelize at <console>:24

scala> val result = rdd.take(5)
result: Array[Int] = Array(1, 2, 3, 4, 5)

scala> println(result.mkString(", "))
1, 2, 3, 4, 5
```

- ✓ 원하는 데이터를 찾기 위해 RDD의 전체 파티션을 다 뒤지지는 않지만 최종 결과 데이터는 배열 혹은 리스트와 같은 컬렉션 타입으로 반환
- ✓ 이 때 지나치게 큰 n값을 지정하면 메모리 부족으로 오류 발생
- ✓ 즉, 입력 파라미터로 숫자값 n을 받아서 소스 RDD의 처음부터 n번째 요소를 포함한 배열을 리턴
- ✓ ", "를 너무 많이 입력하는 것이 귀찮을 때 mkString을 사용

takeSample

❖ **RDD 요소 가운데 지정된 크기의 샘플을 추출하는 메서드**

```
scala> val rdd = sc.parallelize(1 to 100)
rdd: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[2] at parallelize at <console>:24

scala> val result = rdd.takeSample(false, 20)
result: Array[Int] = Array(4, 88, 65, 59, 53, 70, 13, 34, 71, 54, 64, 3, 40, 74, 83, 9, 26, 56, 28, 41)

scala> println(result.length)
20
```

- ✓ Sample() 메서드와 유사하지만 샘플의 크기를 지정할 수 있다는 것과 결과 타입이 RDD가 아닌 배열이나 리스트 같은 컬렉션 타입이라는 차이점이 있음
- ✓ 샘플의 크기를 크게 지정하면 메모리 오류 발생
- ✓ 첫 번째 인자에서 False는 비복원 추출, True면 복원 추출
- ✓ 복원추출은 한 번 뽑은 것을 다시 뽑을 수 있게 하는 방법
- ✓ 두 번째 인자는 몇 개를 추출할 것인지 정할 수 있음
- ✓ 즉, 크기를 정해놓고 샘플을 추출하고자 할 때 적합

collect

❖ **RDD의 모든 요소를 배열 or 리스트 같은 하나의 컬렉션에 담아서 돌려주는 메서드**

```
scala> val rdd = sc.parallelize(List(1 to 100))
rdd: org.apache.spark.rdd.RDD[scala.collection.immutable.Range.Inclusive] = ParallelCollectionRDD[5] at parallelize at <console>:24

scala> val result = rdd.collect
result: Array[scala.collection.immutable.Range.Inclusive] = Array(Range(1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26,
27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50, 51, 52, 53, 54, 55, 56, 57, 58, 59, 60, 61, 62, 63, 64, 65, 66, 67, 68,
 69, 70, 71, 72, 73, 74, 75, 76, 77, 78, 79, 80, 81, 82, 83, 84, 85, 86, 87, 88, 89, 90, 91, 92, 93, 94, 95, 96, 97, 98, 99, 100))
```

- ✓ Rdd의 모든 원소를 드라이버의 메모리에 불러옴
- ✓ RDD의 요소를 배열로 리턴하는 메소드
- ✓ 모든 워커 노드에서 드라이버 프로그램으로 데이터를 이동시키므로 매우 큰 RDD를 호출하게 되면 오류 발생

count

❖ **RDD에 있는 모든 요소의 개수를 돌려주는 메서드**

```
scala> val rdd = sc.parallelize((1 to 100).toList)
rdd: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[7] at parallelize at <console>:24

scala> val result = rdd.count
result: Long = 100
```

✓ RDD의 요소의 수를 리턴하는 메소드

countByValue

❖ **RDD에 속하는 각 값들이 나타나는 횟수를 구해서 맵 형태로 돌려주는 메서드**

```
scala> val rdd = sc.parallelize(List(1, 1, 2, 3, 3))
rdd: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[9] at parallelize at <console>:24

scala> val result = rdd.countByValue
result: scala.collection.Map[Int,Long] = Map(2 -> 1, 1 -> 2, 3 -> 2)

scala> println(result)
Map(2 -> 1, 1 -> 2, 3 -> 2)
```

- ✓ RDD의 유일한 요소의 수를 리턴하는 메소드
- ✓ 키-값 쌍으로 유일한 요소와 그 수를 포함하는 맵 클래스의 인스턴스를 리턴
- ✓ 예를 들어 1, 1, 2, 2, 3으로 이뤄진 RDD가 있으면
  1과 2가 각각 2개 그리고 3이 한 개 포함된다고 할 수 있음
- ✓ reduce()나 fold()를 떠올리기 전에 적용할 수 있는지 검토
- ✓ map() 메서드를 이용하고 reduceByKey() 메서드를 적용하여
  wordcount를 계산하지만 countByValue 한 번만으로 동일한 결과를 얻을 수 있음

reduce

❖ **RDD에 포함된 임의의 값 두개를 하나로 합치는 함수를 이용하여 모든 요소를 하나의 값으로 병합하고 결과값을 반환하는 메서드**

```
scala> val rdd = sc.parallelize(1 to 10, 3)
rdd: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[13] at parallelize at <console>:24

scala> val result = rdd.reduce(_+_)
result: Int = 55

scala> println(result)
55
```

- ✓ Fold() 메서드와 비슷하지만 파라미터 값이 필요 없음
- ✓ 메소드에 입력되는 함수 두 개의 입력값을 받아서 하나의 값을 리턴
- ✓ 입력되는 함수는 서로 연관성이 있어 이항 연산이 가능해야 함
- ✓ _는 전체라는 뜻

fold

❖ **RDD 내의 모든 요소를 대상으로 교환법칙과 결합법칙이 성립되는 바이너리 함수를 순차 적용해 최종 결과를 구하는 메서드**

```
scala> val rdd = sc.parallelize(List(2, 5, 3, 1))
rdd: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[14] at parallelize at <console>:24

scala> val sum = rdd.fold(0) (_+_)
sum: Int = 11

scala> val pro = rdd.fold(1) (_*_)
pro: Int = 30

scala> println(sum)
11

scala> println(pro)
30
```

- ✓ 입력받은 파라미터와 연관된 이항 연산자를 이용하여 소스 RDD의 요소를 집계하는 메소드
- ✓ 각 RDD 파티션에 있는 요소들을 집계하고 각 파티션으로부터 리턴되는 결과를 다시 집계
- ✓ ~~RDD의 연산자들을 모두 합산하려면 파라미터 값은 0, 곱하기를 원한다면 파라미터 값은 1이 되어야 함~~ fold인자로 덧셈과 곱셈의 항등원 값을 넣어줌(0, 1)

aggregate

❖ **reduce()나 fold()는 모두 입력과 출력 타입이 동일해야 하지만 aggregate는 제약 사항이 없기 때문에 입력과 출력의 타입이 다른 경우에도 사용**

```
scala> val result = rdd.aggregate(Record(0, 0))(_ add _, _ add _)
<console>:26: error: not found: value Record
       val result = rdd.aggregate(Record(0, 0))(_ add _, _ add _)
                                  ^
```

ERROR…?

: record가 정의되지 않음

- ✓ 두 단계에 걸쳐 병합을 처리
- ✓ 첫 번째는 파티션 단위로 병합을 수행
- ✓ 두 번째는 파티션 단위 병합 결과끼리 다시 병합을 수행
- ✓ seqOP 함수와 combOp 함수 모두 첫 번째 인자로 이전 단계의 병합 결과로 생성된 객체를 재사용하기 때문에 매번 새로운 객체가 생성되는 부담을 덜 수 있음

sum

❖ **RDD를 구성하는 요소의 타입에 따라 좀 더 특화된 편리한 연산을 제공하기 위해 특정 타입의 요소로 구성된 RDD에서만 사용 가능한 메서드**

```
scala> val rdd = sc.parallelize(1 to 10)
rdd: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[16] at parallelize at <console>:24

scala> val result = rdd.sum
result: Double = 55.0

scala> println(result)
55.0
```

✓ RDD를 구성하는 모든 요소가 double, long등 숫자 타입일 경우에만 사용 가능
✓ 전체 요소의 합을 구해줌

foreach

❖ **RDD의 모든 요소에 특정 함수를 적용하는 메서드**

```
scala> val words = "Scala is fun".split(" ")
words: Array[String] = Array(Scala, is, fun)

scala> words.foreach(println)
Scala
is
fun
```

✓ map()과 유사하지만 아무것도 리턴하지 않음

foreachPartitions

❖ **foreach()와 같이 실행할 함수를 인자로 전달받아 사용하지만 개별 요소가 아닌 파티션 단위로 적용**

```
scala> val b = sc.parallelize(List(1, 2, 3, 4, 5, 6, 7, 8, 9, 10), 2)
b: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[19] at parallelize at <console>:24

scala> b.foreachPartition(x => println(x.reduce(_ + _)))
```

- ✓ RDD를 파티션 처리 한다
- ✓ 파티션 개수는 foreachPartitions전에 미리 지정 되어야 함
- ✓ 리턴 값이 필요 없을 때 사용

cache

❖ **RDD 정보를 메모리 또는 디스크 등에 저장해서 다음 액션을 수행할 때 불필요한 재생성 단계를 거치지 않고 원하는 작업을 즉시 실행할 수 있게 해주는 메서드**

```
scala> val rdd = sc.parallelize(1 to 100, 10)
rdd: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[22] at parallelize at <console>:24

scala> rdd.cache
res18: rdd.type = ParallelCollectionRDD[22] at parallelize at <console>:24
```

- ✓ cache()는 RDD의 데이터를 메모리에 저장하라는 의미
- ✓ 클러스터의 익스큐터 메모리에 RDD를 저장
- ✓ RDD의 생성시기
  - RDD 는 스토리지에 데이터를 읽어 올 때 생성
  - RDD를 트랜스포메이션 할 때 새롭게 생성
  - 기본적으로 RDD액션이 호출될 때 생성
  - 스파크는 호출된 RDD의 부모로부터 RDD를 생성하는 작업을 진행
- ✓ 애플리케이션이 RDD를 캐시할 때 스파크는 RDD를 즉각 만들어서 메모리에 저장하는 것은 아니라 캐시된 RDD에서 첫 번째 액션이 일어날 때 메모리에 올라가게 된다. 때문에 RDD가 캐시된 후 첫 번째 호출된 액션은 개시되어 있지 않아 크게 이점은 없다.

persist, unpersist

❖ **RDD 정보를 메모리 또는 디스크 등에 저장해서 다음 액션을 수행할 때 불필요한 재생성 단계를 거치지 않고 원하는 작업을 즉시 실행할 수 있게 해주는 메서드**

```
scala> val lines = sc.textFile("/user/root/README.md")
lines: org.apache.spark.rdd.RDD[String] = /user/root/README.md MapPartitionsRDD[28] at textFile at <console>:24

scala> lines.persist()
res29: lines.type = /user/root/README.md MapPartitionsRDD[28] at textFile at <console>:24
```

- ✓ persist()는 StorageLevel 옵션을 이용해 저장 위치와 저장 방식 등을 상세히 지정할 수 있는 기능
- ✓ 선택적으로 저장 레벨을 입력 파라미터로 받아 메모리나 디스크, 혹은 둘 다에 저장
  - MEMORY_ONLY
    - val lines = sc.textFile("...")
    - lines.persist(MEMORY_ONLY)
  - DISK_ONLY
  - MEMORY_AND_DISK
  - MEMORY_ONLY_SER : 직렬화된 자바 객체로 메모리에 RDD 파티션을 저장
    - 직렬화된 자바 객체는 보다 적은 메모리를 사용하지만 읽기 작업 시 보다 많은 CPU를 사용한다.
  - MEMORY_AND_DISK_SER : 메모리에 RDD 파티션을 저장할 수 있는 만큼 저장하고 나머지는 디스크에 저장
- ✓ unpersist()는 이미 저장 중인 데이터가 더 이상 필요없을 때 캐시 설정을 취소, 제거하는 데 사용

partitions

❖ **RDD의 파티션 정보가 담긴 배열을 돌려줌**

```
scala> val rdd = sc.parallelize(1 to 100, 10)
rdd: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[24] at parallelize at <console>:24

scala> println(rdd.partitions.size)
10

scala> println(rdd.getNumPartitions)
10
```

- ✓ 파티션의 크기를 알아보기 위한 용도로 많이 활용
- ✓ 단순히 크기 정보만 알아볼 목적이라면 getNumPartitions()를 사용
- ✓ 파이썬인 경우 partitions()를 사용할 수 없음

textFile

❖ **범용적인 데이터 포맷**

- ~~val rdd = sc.textFile("hdfs://namenode:9000/path/to/file-or-directory")~~
- val rdd = sc.textFile("hdfs://namenode:9000/path/to/directory/*.gz")

✓ 입력받은 텍스트 파일을 읽고 파일의 각 라인에 해당하는 문자열을 하나의 RDD 요소로 하는 새로운 RDD를 생성하는 메서드

✓ HDFS, 아마존, S3, Hadoop을 지원하는 스토리지 시스템에 저장된 디렉터리에서 하나 혹은 다수의 파일을 읽어 라인들을 표현하는 String RDD를 리턴

✓ 실습 할ㄹ 때 디렉토리를 지정했을때는 읽을 수가 없는 것으로 나타남

objectFile

❖ **오브젝트 직렬화 방법을 이용해 RDD를 구성하는 요소를 파일에 읽고 쓰는 기능을 수행**

- val numbersRdd = sc.parallelize((1 to 10000).toList)
- val filteredRdd = numbersRdd filter { x => x % 1000 == 0}
- filteredRdd.saveAsObjectFile("numbers-as-object")

- ✓ RDD에 포함된 데이터를 오브젝트 파일로 다루기 위하여 각 요소 오브젝트가 자바의 Serializable 인터페이스를 구현하고 있어야 함
- ✓ 파이썬의 경우 objectFile 대신 파이썬의 pickle 라이브러리를 사용하는 saveAsPickleFile()과 pickleFile() 메서드를 사용해야 함
- ✓ 저장된 RDD의 타입이 RDD[INT]일 경우 이 파일을 읽어서 생성한 RDD도 RDD[INT] 타입이여야 함

sequenceFile

❖ **키와 값으로 구성된 데이터를 저장하는 이진(바이너리) 파일 포맷**

➢ val rdd = sc.sequenceFile[String, String](“시퀀스파일 저장한 경로")

✓ 하둡에서도 자주 사용되는 대표적 파일 포맷
✓ 대량의 데이터 처리에 적합한 분할 압축 기능
✓ 효율적인 파일 관리에 적합한 구조를 띠고 있음

공유 변수

❖ **브로드캐스트 변수 및 어큐뮬레이터 들어가기 전에..**

- ✓ 스파크는 기본적으로 비공유 아키텍쳐
- ✓ 태스크 간 혹은 드라이버 프로그램과 공유하는 글로벌메모리 없음
- ✓ 프로그램과 잡 태스크 간의 메시지를 통해 공유
- ✓ 드라이버 프로그램의 변수를 그대로 복사하여 태스크에 사용

broadcast

❖ **드라이버 프로그램에 의해 공유되어 복사되는 변수**

- val broadcastUser = sc.broadcast(Set("u1", "u2"))
- val rdd = sc.parallelize(List("u1", "u3", "u3", "u4", "u5", "u6"), 3)
- val result = rdd.filter(broadcastUser.value.contains(_))

- ✓ 먼저 공유하고자 하는 데이터를 포함한 오브젝트를 생성
- ✓ 이 오브젝트를 스파크컨텍스트의 브로드캐스트 메서드의 인자로 지정하여 메서드 실행
- ✓ 생성된 브로드캐스트 변수를 사용할 때는 위에서 생성한 브로드캐스트 변수의 value() 메서드를 통해 접근

accumulator

❖ **태스크가 오로지 더하기 연산만 가능한 공유변수**

- val acc1 = sc.longAccumulator("invalidFormat")
- val acc2 = sc.collectionAccumulator[String]("invalidFormat2")
- val data = List("U1:Addr1", "U2:Addr2", "U3", "U4;Addr4", "U5::Addr5")
- Sc.parallelize(data, 3).foreach { v =>
- If (v.split(":").length != 2) {
- Acc1.add(1L)
- Acc2.add(v)
- }
- }
- println("잘못된 데이터 수:" + acc1.value)
- println("잘못된 데이터:" + acc2.value)

✓ 클러스터 내의 모든 서버가 공유하는 쓰기 공간을 제공함
✓ 각 서버에서 발생하는 특정 이벤트의 수를 세거나 관찰하고 싶은 정보를 모아두는 용도로 사용