

Resilient Distributed Datasets

스파크 컨텍스트

- 스파크 애플리케이션과 클러스터 연결을 관리하는 객체
- 모든 스파크 애플리케이션은 반드시 스파크 컨텍스트를 생성

```
val conf = new SparkConf().setAppName(appName).setMaster(master)  
new SparkContext(conf)
```

- » 마스터 서버의 정보와 애플리케이션 이름은 필수 항목
- 스파크 셀에서는 자동 생성되어 바로 사용 가능
 - » sc 변수 또는 spark.sparkContext 변수로 참조
- 스파크 컨텍스트를 통해 RDD 생성

RDD

- 스파크 전용 분산 컬렉션으로 스파크의 기본 추상화 객체
- 특성
 - » 불변성(immutable) → 읽기 전용(readonly)
 - › 한 번 생성된 RDD는 절대 변경할 수 없음
 - » 복원성(resilient) → 장애 내성
 - › 노드에 장애가 발생해서 유실된 RDD를 원래대로 복구 가능
 - › 데이터셋을 만드는 데 사용된 작업 로그를 보존해서 장애 내성 제공
 - » 분산(distributed) → 한 개 이상의 노드에 저장된 데이터셋
 - › 사용자에게는 위치 투명성 제공 → 일반적인 데이터셋을 다루는 것과 차이가 없음

RDD 연산

- Transformation과 Action이라는 두 종류의 연산 제공
- Transformation
 - » RDD의 데이터를 조작해 새로운 RDD 생성 (filter, map 등)
 - » Action 연산이 호출될 때까지 실행되지 않음
- Action
 - » 연산을 호출한 프로그램으로 계산 결과를 변환하거나 RDD 요소에 특정 작업을 수행하기 위해 실제 계산을 시작하는 역할 (count, foreach 등)
- Action 연산이 호출되면 스파크는 RDD의 계보(lineage)를 살펴보고 이를 바탕으로 연산 그래프를 작성해서 최적화된 방식으로 연산 수행

RDD 생성

- 드라이버 프로그램의 컬렉션 객체 사용

```
rdd1 = sc.parallelize(["a", "b", "c", "d", "e"], 2)
rdd1.collect()
```

- 파일 또는 데이터베이스 등 외부 데이터 사용

```
rdd1 = sc.textFile("file:///home/hadoop/apps/spark/README.md")
rdd1.collect()
```

RDD 기본 액션

▪ collect

» RDD의 모든 원소를 모아서 배열로 반환

```
rdd = sc.parallelize(range(1, 11))
result = rdd.collect()
print(result)
```

▪ count

» 전체 요소의 개수 반환

```
rdd = sc.parallelize(range(1, 11))
result = rdd.count()
print(result)
```

▪ take

» 지정된 개수의 요소를 모아서 배열로 반환

```
rdd = sc.parallelize(range(1, 100))
result = rdd.take(5)
print(result)
```

RDD 기본 액션

▪ first

- » RDD의 첫 번째 요소를 반환

```
rdd = sc.parallelize([5, 4, 1])
result = rdd.first()
print(result)
```

▪ countByValue

- » 각 값들이 나타나는 횟수를 구해서 맵 형태로 반환

```
rdd = sc.parallelize((1, 1, 2, 3, 3))
result = rdd.countByValue()
print(result)
```

▪ reduce

- » RDD에 포함된 임의의 두 값을 하나로 합치는 함수를 통해 모든 요소를 하나의 값으로 병합

```
rdd = sc.parallelize(list(range(1, 10, 3)))
result = rdd.reduce(lambda x, y: x + y)
print(result)
```

RDD 기본 액션

■ sum

- » 요소의 자료형이 숫자형인 경우에 사용
- » 전체 요소 값의 합 반환

```
rdd = sc.parallelize(list(range(1, 11)))
result = rdd.sum()
print(result)
```

RDD의 기본 트랜스포메이션

- map

- » 모든 요소에 지정된 함수를 적용하고 그 결과로 새로운 RDD 생성 및 반환

```
rdd = sc.parallelize(list(range(1, 6)))
result = rdd.map(lambda x: x + 1)
print(result)
```

- flatMap

- » 전달인자 함수는 컬렉션 반환 → 최종 결과 함수는 1차원 배열 반환

```
fruits = ("apple,orange", "grape,apple,mango", "blueberry,tomato,orange")
rdd1 = sc.parallelize(fruits)
rdd2 = rdd1.flatMap(lambda x: x.split(","))
# rdd2 = rdd.map(lambda x: x.split(","))
print(rdd2.collect())
```

RDD의 기본 트랜스포메이션

■ mapValues

- » 키와 값으로 이루어진 pairRDD에 적용
- » 인자로 전달 받은 요소의 키는 무시하고 값만 처리해서 반환

```
rdd1 = sc.parallelize(["a", "b", "c"])
rdd2 = rdd1.map(lambda v: (v, 1))
rdd3 = rdd2.mapValues(lambda i: i + 1)
print(rdd3.collect())
```

■ flatMapValues

- » 키와 값으로 이루어진 pairRDD에 적용
- » 인자로 전달 받은 요소의 키는 무시하고 값만 처리해서 반환
- » flatMap처럼 결과를 1차원 배열 형식으로 반환

```
rdd1 = sc.parallelize([(1, "a,b"), (2, "a,c"), (1, "d,e")])
rdd2 = rdd1.flatMapValues(lambda s: s.split(","))
print(rdd2.collect())
```

RDD 기본 트랜스포메이션

- zip
 - » 두 개의 서로 다른 RDD에서 순차적으로 요소를 뽑아서 (키, 값) 쌍으로 묶어서 반환

```
rdd1 = sc.parallelize(["a", "b", "c"])
rdd2 = sc.parallelize([1, 2, 3])
result = rdd1.zip(rdd2)
print(result.collect())
```

RDD 기본 트랜스포메이션

▪ groupBy

- » RDD 요소를 일정한 기준에 따라 여러 개의 그룹으로 나누고 이 그룹으로 구성된 새로운 RDD 반환

```
rdd1 = sc.parallelize(range(1, 11))
rdd2 = rdd1.groupBy(lambda v: "even" if v % 2 == 0 else "odd")
for x in rdd2.collect():
    print(x[0], list(x[1]))
```

▪ groupByKey

- » key-value로 구성된 데이터에 대해 키에 따라 여러 개의 그룹으로 나누고 이 그룹으로 새로운 RDD 반환

```
rdd1 = sc.parallelize(["a", "b", "c", "b", "c"]).map(lambda v: (v, 1))
rdd2 = rdd1.groupByKey()
for x in rdd2.collect():
    print(x[0], list(x[1]))
```

RDD 기본 트랜스포메이션

▪ distinct

- » 중복을 제외한 요소로만 구성된 새로운 RDD 반환

```
rdd = sc.parallelize([1, 2, 3, 1, 2, 3, 1, 2, 3])
result = rdd.distinct()
print(result.collect())
```

▪ filter

- » 원하는 요소만 걸러내는 동작 수행

```
rdd1 = sc.parallelize(range(1, 6))
rdd2 = rdd1.filter(lambda i: i > 2)
print(rdd2.collect())
```

▪ sortByKey

- » 키 값을 기준으로 요소를 정렬하는 연산

```
rdd = sc.parallelize([('q', 1), ('z', 1), ('a', 1)])
result = rdd.sortByKey()
print(result.collect())
```

RDD 기본 트랜스포메이션

- keys, values

- » 각각 key와 value로 구성된 RDD 반환

```
dd = sc.parallelize([("k1", "v1"), ("k2", "v2"), ("k3", "v3")])
print(rdd.keys().collect())
print(rdd.values().collect())
```

- sample

- » 샘플을 추출해 새로운 RDD 생성

```
rdd = sc.parallelize(range(1, 101))
result1 = rdd.sample(False, 0.5, 100)
result2 = rdd.sample(True, 1.5, 100)
print(result1.take(5))
print(result2.take(5))
```

RDD 기본 트랜스포메이션

▪ cartesian

- » 두 RDD 요소의 Cartesion 곱을 구하고 결과를 요소로 RDD 생성

```
rdd1 = sc.parallelize([1, 2, 3])
rdd2 = sc.parallelize(["a", "b", "c"])
result = rdd1.cartesian(rdd2)
print(result.collect())
```

▪ subtract

- » 두 RDD의 차집합으로 RDD 생성

```
rdd1 = sc.parallelize(["a", "b", "c", "d", "e"])
rdd2 = sc.parallelize(["d", "e"])
result = rdd1.subtract(rdd2)
print(result.collect())
```

RDD 기본 트랜스포메이션

- union

- » 두 RDD 합집합으로 RDD 생성

```
rdd1 = sc.parallelize(["a", "b", "c"])
rdd2 = sc.parallelize(["d", "e", "f"])
result = rdd1.union(rdd2)
print(result.collect())
```

- intersection

- » 두 RDD의 교집합으로 RDD 생성

```
rdd1 = sc.parallelize(["a", "a", "b", "c"])
rdd2 = sc.parallelize(["a", "a", "c", "c"])
result = rdd1.intersection(rdd2)
print(result.collect())
```

RDD 기본 트랜스포메이션

▪ join

- » key - value 형식의 요소에 대해 적용 가능
- » 서로 같은 키를 가지고 있는 요소를 모아서 그룹을 만들고 새로운 RDD 생성

```
rdd1 = sc.parallelize(["a", "b", "c", "d", "e"]).map(lambda v: (v, 1))
rdd2 = sc.parallelize(["b", "c"]).map(lambda v: (v, 2))
result = rdd1.join(rdd2)
print(result.collect())
```

▪ leftOuterJoin, rightOuterJoin

- » 외부 조인 수행 결과로 생성

```
rdd1 = sc.parallelize(["a", "b", "c"]).map(lambda v: (v, 1))
rdd2 = sc.parallelize(["b", "c"]).map(lambda v: (v, 2))
result1 = rdd1.leftOuterJoin(rdd2)
result2 = rdd1.rightOuterJoin(rdd2)
print("Left: %s" % result1.collect())
print("Right: %s" % result2.collect())
```