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 셸에서는 자동 생성되어 바로 사용 가능
  - sc 변수 또는 spark.sparkContext 변수로 참조

RDD 생성

- 드라이버 프로그램의 컬렉션 객체 사용

```
val rdd1 = sc.parallelize(List("a", "b", "c", "d", "e"), 2)
```

- 파일 또는 데이터베이스 등 외부 데이터 사용

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

RDD 기본 액션

- collect
  - RDD의 모든 원소를 모아서 배열로 반환

```
val rdd = sc.parallelize(1 to 10)
val result = rdd.collect
println( result.mkString(", ") )
```

- count
  - 전체 요소의 개수 반환

```
val rdd = sc.parallelize(1 to 10)
val result = rdd.count
println( result )
```

- take
  - 지정된 개수의 요소를 모아서 배열로 반환

```
val rdd = sc.parallelize(1 to 10)
val result = rdd.take(5)
println( result.mkString(", ") )
```

RDD 기본 액션

- first
  - RDD의 첫 번째 요소를 반환

```
val rdd = sc.parallelize(List(5, 4, 1))
val result = rdd.first
println(result)
```

- countByValue
  - 각 값들이 나타나는 횟수를 구해서 맵 형태로 반환

```
val rdd = sc.parallelize(List(1, 1, 2, 3, 3))
val result = rdd.countByValue
println(result)
```

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

```
val rdd = sc.parallelize(1 to 10, 3)
val result = rdd.reduce(_ + _)
println(result)
```

RDD 기본 액션

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

```
val rdd = sc.parallelize(1 to 10)
val result = rdd.sum
println(result)
```

- foreach
  - 모든 요소에 특정 함수 적용

```
val rdd = sc.parallelize(1 to 10, 3)
rdd.foreach { v =>
    println(s"Value Side Effect: ${v}")
}
```

RDD의 기본 트랜스포메이션

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

```
val rdd = sc.parallelize(1 to 5)
val result = rdd.map(_ + 1)
println( result.collect.mkString(", ") )
```

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

```
val rdd = sc.parallelize(1 to 5)
val result = rdd.map(_ + 1)
printlnval fruits = List("apple,orange", "grape,apple,mango",
"blueberry,tomato,orange")
val rdd = sc.parallelize(fruits)
val rdd2 = rdd.flatMap(_.split(","))
print(rdd2.collect.mkString(", ")) ( result.collect.mkString(", ") )
```

RDD의 기본 트랜스포메이션

- mapValues
  - 키와 값으로 이루어진 pariRDD에 적용
  - 인자로 전달 받은 요소의 키는 무시하고 값만 처리해서 반환

```
val rdd = sc.parallelize(List("a", "b", "c")).map((_, 1))
val result = rdd.mapValues( i => i + 1 )
println( result.collect.mkString("\t") )
```

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

```
val rdd = sc.parallelize(List("a", "b", "c")).map((_, 1))
val result = rdd.mapValues( i => i + 1 )
println( result.collect.mkString("\t"))
```

RDD의 기본 트랜스포메이션

- mapValues
  - 키와 값으로 이루어진 pariRDD에 적용
  - 인자로 전달 받은 요소의 키는 무시하고 값만 처리해서 반환

```
val rdd = sc.parallelize(List("a", "b", "c")).map((_, 1))
val result = rdd.mapValues( i => i + 1 )
println( result.collect.mkString("\t") )
```

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

```
val rdd = sc.parallelize(List("a", "b", "c")).map((_, 1))
val result = rdd.mapValues( i => i + 1 )
println( result.collect.mkString("\t"))
```

RDD 기본 트랜스포메이션

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

```
val rdd1 = sc.parallelize(List("a", "b", "c"))
val rdd2 = sc.parallelize(List(1, 2, 3))
val result = rdd1.zip(rdd2)
println( result.collect().mkString(", ") )
```

RDD 기본 트랜스포메이션

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

```
val rdd = sc.parallelize(1 to 10)
val result = rdd.groupBy {
    case i: Int if (i % 2 == 0) => "even"
    case _ => "odd"
}
result.collect.foreach {
    v => println(s"${v._1}, [${v._2.mkString(",")}]")
}
```

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

```
val rdd = sc.parallelize(List("a", "b", "c", "b", "c")).map((_, 1))
val result = rdd.groupByKey
result.collect.foreach {
    v => println(s"${v._1}, [${v._2.mkString(",")}]")
}
```

RDD 기본 트랜스포메이션

- distinct
  - 중복을 제외한 요소로만 구성된 새로운 RDD 반환

```
val rdd = sc.parallelize(List(1, 2, 3, 1, 2, 3, 1, 2, 3))
val result = rdd.distinct
println(result.collect.mkString(", "))
```

- filter
  - 원하는 요소만 걸러내는 동작 수행

```
val rdd = sc.parallelize(1 to 5)
val result = rdd.filter(_ > 2)
println(result.collect.mkString(", "))
```

- sortByKey
  - 키 값을 기준으로 요소를 정렬하는 연산

```
val rdd = sc.parallelize(List("q", "z", "a"))
val result = rdd.map( (_, 1) ).sortByKey()
println( result.collect.mkString(", "))
```

RDD 기본 트랜스포메이션

- keys, values
  - 각각 key와 value로 구성된 RDD 반환

```
val rdd = sc.parallelize(List(("k1", "v1"), ("k2", "v2"), ("k3", "v3")))
println(rdd.keys.collect.mkString(","))
println(rdd.values.collect.mkString(","))
```

- sample
  - 샘플을 추출해 새로운 RDD 생성

```
val rdd = sc.parallelize(1 to 100)
val result1 = rdd.sample(false, 0.5)
val result2 = rdd.sample(true, 1.5)
println(result1.take(5).mkString(", "))
println(result2.take(5).mkString(", "))
```

RDD 기본 트랜스포메이션

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

```
val rdd1 = sc.parallelize(List(1, 2, 3))
val rdd2 = sc.parallelize(List("a", "b", "c"))
val result = rdd1.cartesian(rdd2)
println(result.collect.mkString(", "))
```

- subtract
  - 두 RDD의 차집합으로 RDD 생성

```
val rdd1 = sc.parallelize(List("a", "b", "c", "d", "e"))
val rdd2 = sc.parallelize(List("d", "e"))
val result = rdd1.subtract(rdd2)
println(result.collect.mkString(", "))
```

RDD 기본 트랜스포메이션

- union
  - 두 RDD 합집합으로 RDD 생성

```
val rdd1 = sc.parallelize(List("a", "b", "c"))
val rdd2 = sc.parallelize(List("d", "e", "f"))
val result = rdd1.union(rdd2)
println(result.collect.mkString(", "))
```

- intersection
  - 두 RDD의 교집합으로 RDd 생성

```
val rdd1 = sc.parallelize(List("a", "a", "b", "c"))
val rdd2 = sc.parallelize(List("a", "a", "c", "c"))
val result = rdd1.intersection(rdd2)
println(result.collect.mkString(", "))
```

RDD 기본 트랜스포메이션

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

```
val rdd1 = sc.parallelize(List("a", "b", "c", "d", "e")).map( (_, 1) )
val rdd2 = sc.parallelize(List("b", "c")).map( (_, 2) )
val result = rdd1.join(rdd2)
println(result.collect.mkString("\n"))
```

- leftOuterJoin, rightOuterJoin
  - 외부 조인 수행 결과로 생성

```
val rdd1 = sc.parallelize(List("a", "b", "c")).map( (_, 1) )
val rdd2 = sc.parallelize(List("b", "c")).map( (_, 2) )
val result1 = rdd1.leftOuterJoin(rdd2)
val result2 = rdd1.rightOuterJoin(rdd2)
println("Left Outer Join " + result1.collect.mkString("\n"))
println("Right Outer Join " + result2.collect.mkString("\n"))
```