SCALA 与 SPARK 编程基础

北京数人科技　　马越

yma@dataman-inc.com

北京数人科技

- 初创
- 来自 Google
- to Business
- 云计算平台
- 大数据解决方案
- 工程师文化

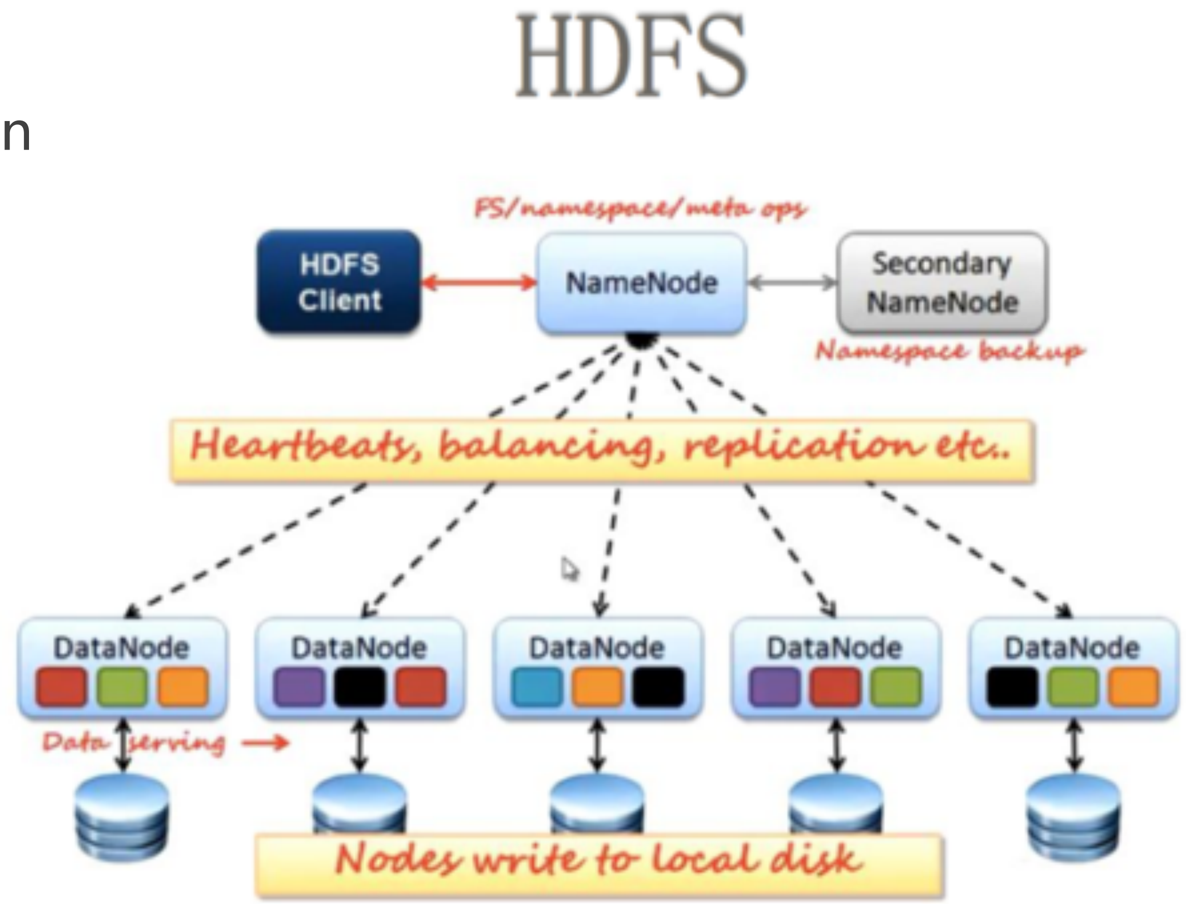
课程内容

- ➢ spark 介绍
- ➢ scala 基础语法
- ➢ spark 编程原理与特性
- ➢ spark 各模块使用
- ➢ 开发实践

Spark 介绍

Talk about Hadoop firstly

- Core Components :
  - Hadoop Common
  - HDFS
  - Hadoop YARN
  - MapReduce



What is Spark

- a fast and general engine for large-scale data processing.
    - fast
    - easy to use
    - generality
    - integrated with Hadoop

Why spark fast

- Memory based computation
- DAG
- Thread model
- Optimization

How easy to use

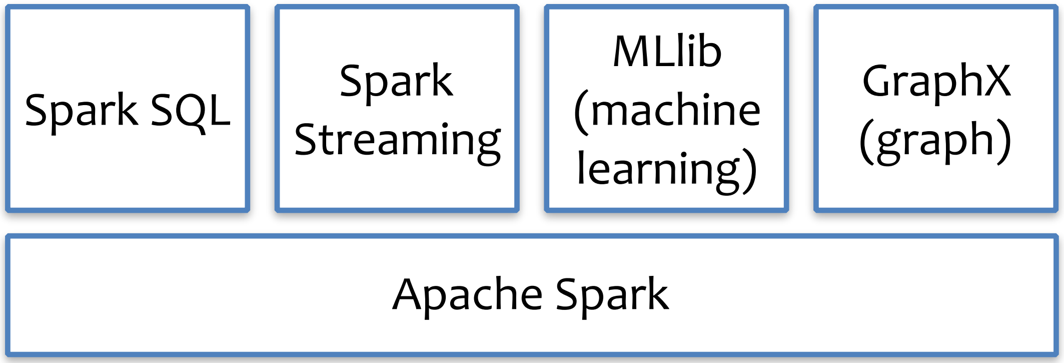
- API Language Support
  - Scala
  - Java
  - Python
  - R

How easy to use

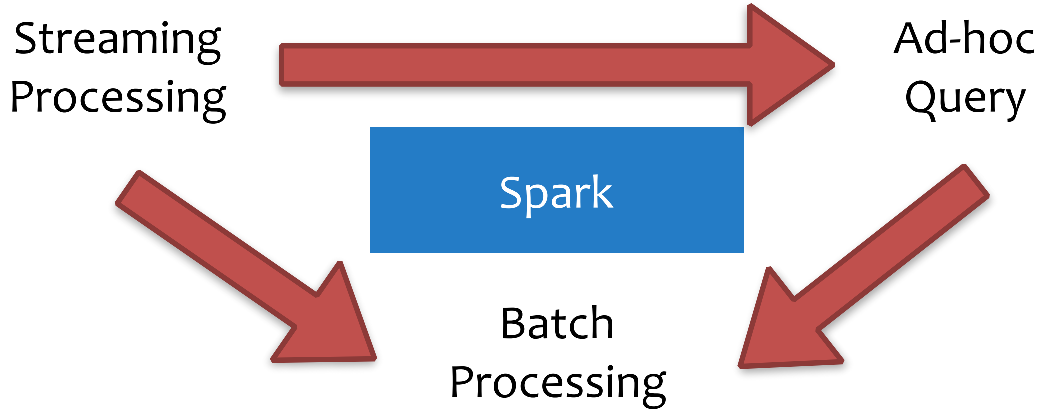
- Multi type Deployment
  - local
  - Standalone
  - Mesos
  - Yarn

All-in-One principle

- Combine SQL, streaming, and complex analytics.

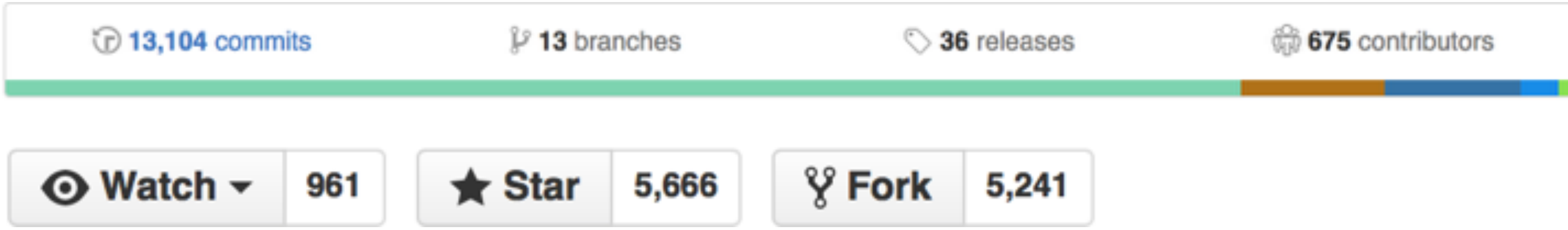


- One stack to rule them all.



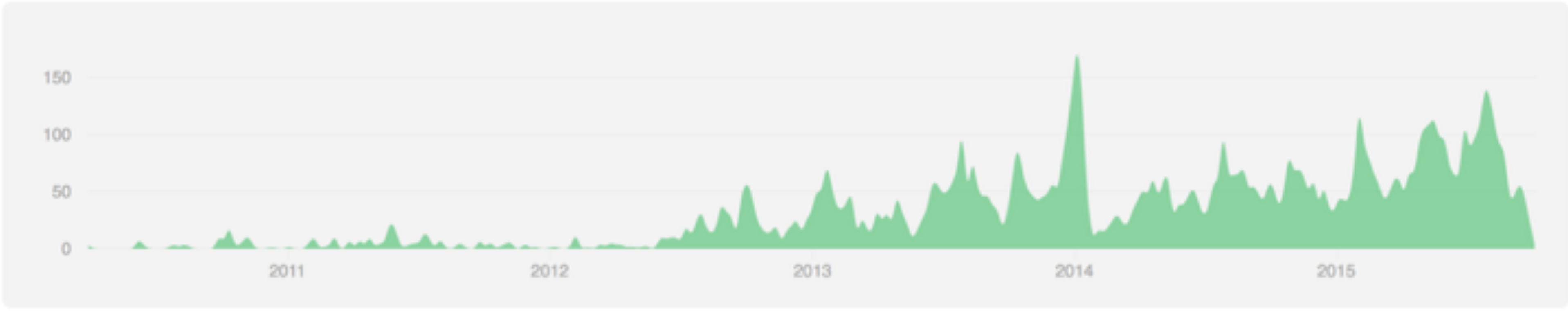
Today of Spark

- Spark 1.5.1 released in Oct 02, 2015



13,104 commits 13 branches 36 releases 675 contributors

Watch 961 Star 5,666 Fork 5,241



Scala 基础

Why to learn scala

- high expandable
  - OOP
  - FP
- compatible with Java
  - sharing class libraries
  - cooperation
- concise phrase
- concurrency support

How to install

- Download: http://www.scala-lang.org/download/
- Linux/Unix/Mac

```
$SCALA_HOME     /usr/local/share/scala
$PATH            $PATH:$SCALA_HOME/bin
```

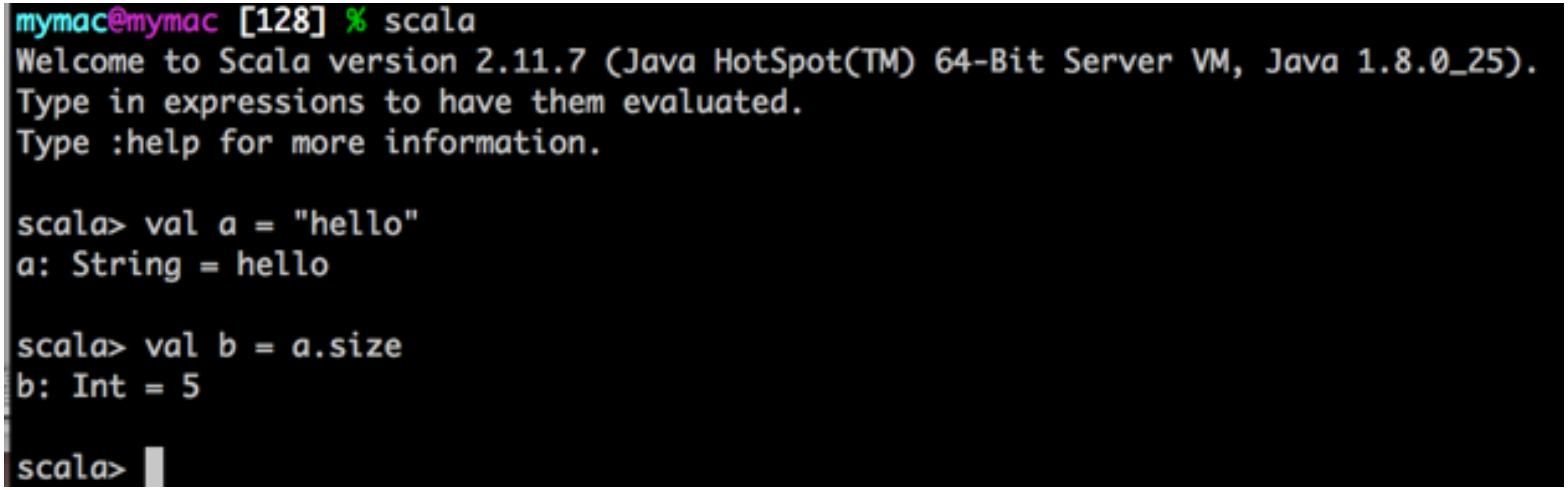
- Windows

```
%SCALA_HOME%     c:\Progra~1\Scala
%PATH%            %PATH%;%SCALA_HOME%\bin
```

- IDE
  - IntelliJ IDEA
  - Eclipse
  - NetBeans

Scala Interpreter

➢ Actually, It is a REPL, not an interpreter



```
mymac@mymac [128] % scala
Welcome to Scala version 2.11.7 (Java HotSpot(TM) 64-Bit Server VM, Java 1.8.0_25).
Type in expressions to have them evaluated.
Type :help for more information.

scala> val a = "hello"
a: String = hello

scala> val b = a.size
b: Int = 5

scala>
```

Scala基础

- ➢ 声明与定义：
  - ➢ val, 常量声明
  - ➢ var, 变量声明
    - 类型省略（默认类型）
    - 声明省略（连续声明）
  - ➢ def, 函数声明
  - ➢ type, 类型声明
  - ➢ class, 类声明
  - ➢ object, 对象声明

Scala基础

- 数值类型：
  - 整数值
    - Int，Long，Short，Byte
  - 浮点值
    - Double，Float
  - 布尔值
    - Boolean
  - 字符值
    - Char，String

Scala基础

- 操作符：
  - 数学操作符：+、-、*、/、%
  - 比较操作符：<、>、>=、<=
  - 逻辑操作符：&&、||、&、|
  - 对等操作符：==、!=
  - 位操作符：&、|、^、~
  - 没有++、- -
  - 支持操作符重载

Scala基础

- ➢ 语句终止：
  - ➢ 分号和换行均可作为语句的结束
  - ➢ 判定换行是否结束
    - 换行前的符号是一个语句的结束:常量、标识符、保留字及其他分隔符
    - 换行后的符号是一个语句的开始:除部分分隔符及保留字外的所有符号
    - 符号处在一个允许多语句的区域:Scala源文件中,匹配的{与}间
  - ➢ 换行不作为语句结束
    - 匹配的(与)之间,[与]之间
    - XML模式下的区域
    - case符号及匹配的=>符号之间
    - if、while、for、type是允许跨两行的合法语句

Scala基础

- 结构控制语句：
  - 判断：if
  - 循环：while/do
  - 枚举：for
  - 匹配：match
  - 异常处理：throw/try
  - 输出：print/println
  - 输入：readline/Source

注意：scala 没有受验异常

Scala基础

- 包引入：
  - 直接在使用时引入

    ```
    val a = scala.collection.immutable.Map((a1,b1),(a2,b2))
    ```

  - 利用 import 语句引入

    ```
    import scala.collection.mutable.ArrayBuffer
    val b = new ArrayBuffer[T]()
    ```

  - 默认引入

    ```
    import java.lang._
    import scala._
    import Predef._
    ```

Scala基础

- 类/对象/样例类/特质
  - class
  - object
  - case class
  - trait

Scala基础

➢ class

```
class User {
   var name = "anonymous"
   var age:Int = _
   val country = "china"
   def email = name + "@mail"
}

val u = new User
u.name = “qh”; u.age = 30
println(u.name + “, “ + u.age)
u.country = “usa"
u.email
```

- object
  - 定义静态成员
  - 伴生对象

```
object User {
  var name = "anonymous"
  var age:Int = _
  val country = "china"
  def email = name + "@mail"
}
```

注：伴生对象和类必须定义在同一个文件中

- case class
  - 实例创建不用 new
  - 默认提供 getXX 方法
  - 默认提供 toString 方法
  - 常结合类继承使用

```
abstract class Person
case class Man(power:Int) extends Person
case class Woman(beauty:Int, from:String) extends Person

val w1 = Woman(100, “china”)
w1.toString
w1.from
def func(t: Person) = t.match {
  case Man(x) => “man’s power:” + x
  case Woman(x, y) => y + “ beauty:” + x
  case _ => “alien?”
}
func(Woman(20, “china”))
```

- ➢ trait
  - ➢ 同时允许定义抽象方法和具体方法

```
trait Runnable {
  def run(): Unit = println("Dad run")
}

trait t1 extends Runnable

trait t2 extends t1 with Runnable {
  override def run(): Unit = println("Son run")
}

class c3 extends t2 with t1 with Runnable
```

Scala基础

- ➢ 模式匹配
  - ➢ 替代 switch
  - ➢ 类型模式匹配
  - ➢ 集合对象匹配
  - ➢ 样例类匹配

- ➢ 替代 switch

```
var sign = 0
def get(obj:Char) = {
  obj match {
    case '+' => sign = 1
    case '-' | 'v' => sign = -1
    case _ => sign = 0
  }
  println(sign)
}
```

➢ 类型的模式匹配

```
def getType(obj:Any) = {
  obj match{
    case _ :Array[Char] => println("Array[Char]")
    case _ :Int => println("Int")
    case _ :Char => println("Char")
    case _ => println("Error")
  }
}
```

➢ 集合的模式匹配

```
def get(pair:Any) = {
  pair match {
    case (0, _) => "0 ..."
    case (y, 0) => y + " 0"
    case _ => "neither is 0"
  }
}
```

➢ 样例类匹配

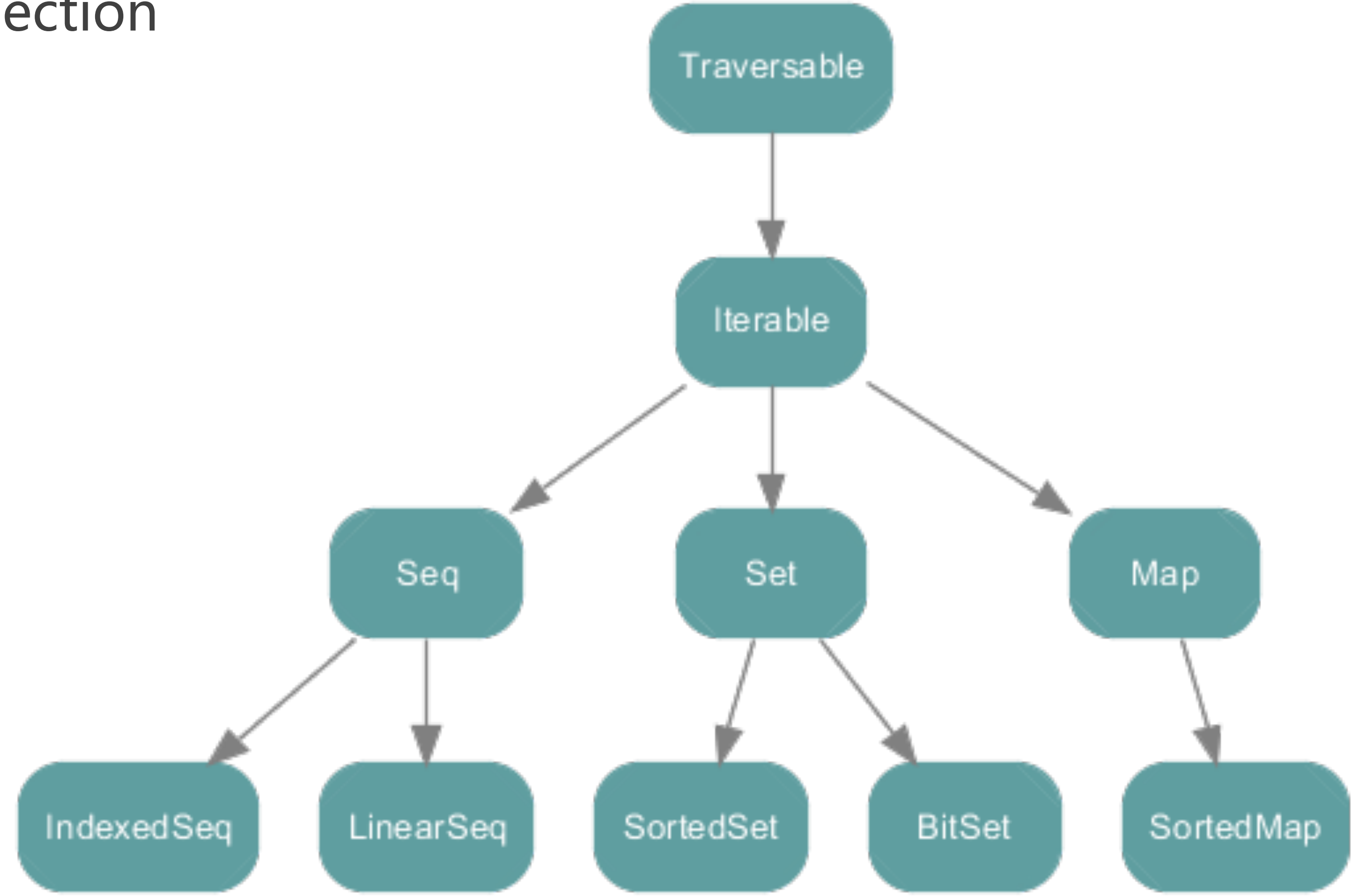
```
abstract class Amount
case class Dollar(value: Double) extends Amount
case class Currency(value: Double, unit: String) extends Amount
case object Nothing extends Amount

def get(amt:Any): String ={
  amt match {
    case Dollar(v) => "$" + v
    case Currency(_, u) => "oh no, I got " + u
    case Nothing => ""
  }
}
```

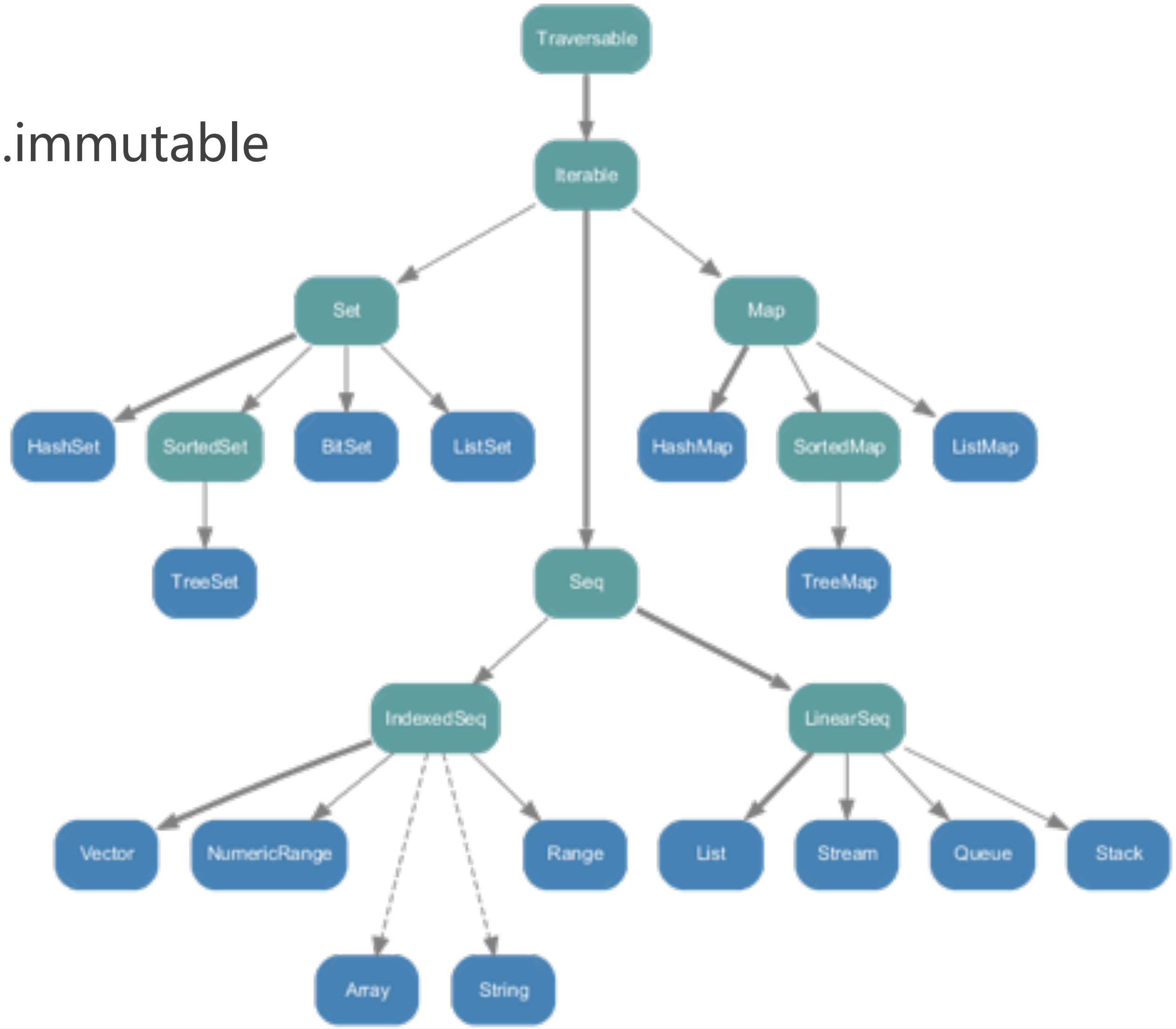
Scala基础

- 集合
  - 集合类库：
    - scala.collection
    - scala.collection.immutable
    - scala.collection.mutable

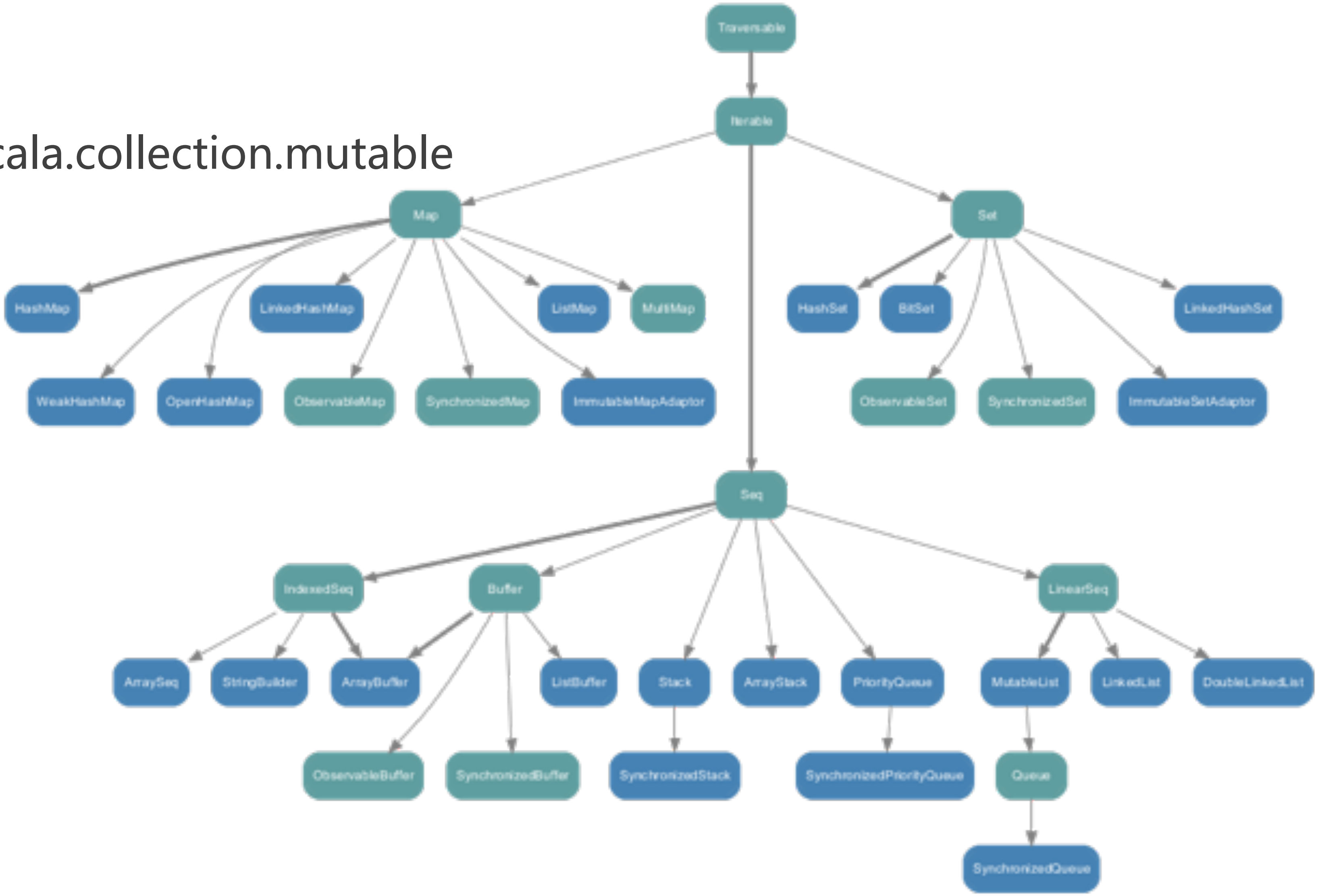
- scala.collection



- scala.collection.immutable



- scala.collection.mutable



- mutable/immutable

```
def basicSetSize[A](as: scala.collection.Set[A]): Int = as.size
basicSetSize(scala.collection.immutable.Set(1, 2, 3))
res0: Int = 3
basicSetSize(scala.collection.mutable.Set(1, 2, 3))
res1: Int = 3
```

- Tuple

```
val t1 = ("a","b","c")
var t2 = ("a", 123, 3.14, new java.util.Date())
val (a,b,c) = (2,4,6)
1->"hello world"
(1, "hello world")

t1._1
t2._2
```

- 集合操作：
  - map
  - filter/filterNot
  - partition/splitAt/groupBy
  - foreach
  - exists
  - find
  - sorted/sortWith/sortBy
  - distinct
  - flatMap
  - indices/zipWithIndex/slice
  - take/drop
  - count
  - updated/patch
  - contains/startsWith/endWith
  - intersect/diff/union

Scala 基础

- ➢ 函数式编程
  - ➢ In computer science, functional programming is a programming paradigm that treats computation as the evaluation of mathematical functions and avoids state and mutable data.
  - ➢ 一种“编程范式”，将计算机运算看做是数学函数的计算，避免状态以及变量的概念。
  - ➢ 特征：

    1. 函数是“第一等公民”
    2. 无“副作用”
    3. 引用透明
    4. 尾递归优化

- 函数是“第一等公民”
    - 可以传递、赋值
    - 嵌套函数和匿名函数
    - 高阶函数、闭包
    - 局部函数
    - 柯里化

➢ 无 “副作用”

```
def expr(x:Int, n:Int):Int = {
  var result = 1
  for(i <- 0 until n){
    result = result*x
  }
  result
}
```

$\Rightarrow$

```
def expr(x:Int,n:Int):Int={
  if(n==0)
    1
  else x*expr(x,n-1)
}
```

➢ 引用透明

　如果程序中两个相同值得表达式能在该程序的任何地方互相替换，而不影响程序的动作，那么该程序就具有引用透明性。纯函数式语言没有变量，所以它们都具有引用透明性。

➢ 尾递归调用

尾递归不保持当前递归函数的状态，而把需要保持的东西全部用参数给传到下一个函数里，这样就可以自动清空本次调用的栈空间。

```
//递归
def fib(n:Int):Int = n match{
  case 0 => 1
  case 1 => 1
  case _ =>fib(n-1) + fib(n-2)
}
//尾递归
def fib2(a:Int,b:Int,n:Int):Int = n match{
  case 0 => b
  case _ =>fib2(b,a+b,n-1)
```

Reference

- 《Programming in scala》
  - Author: Martin Odersky, Lex Spoon, Bill Venners
  - Publisher: Artima Inc.

- 《快学 scala》(Scala for the impatient)
  - Author: Cay S. Horstmann
  - Publisher: 电子工业出版社

- Scaladoc
  - http://www.scala-lang.org/api

Spark基础

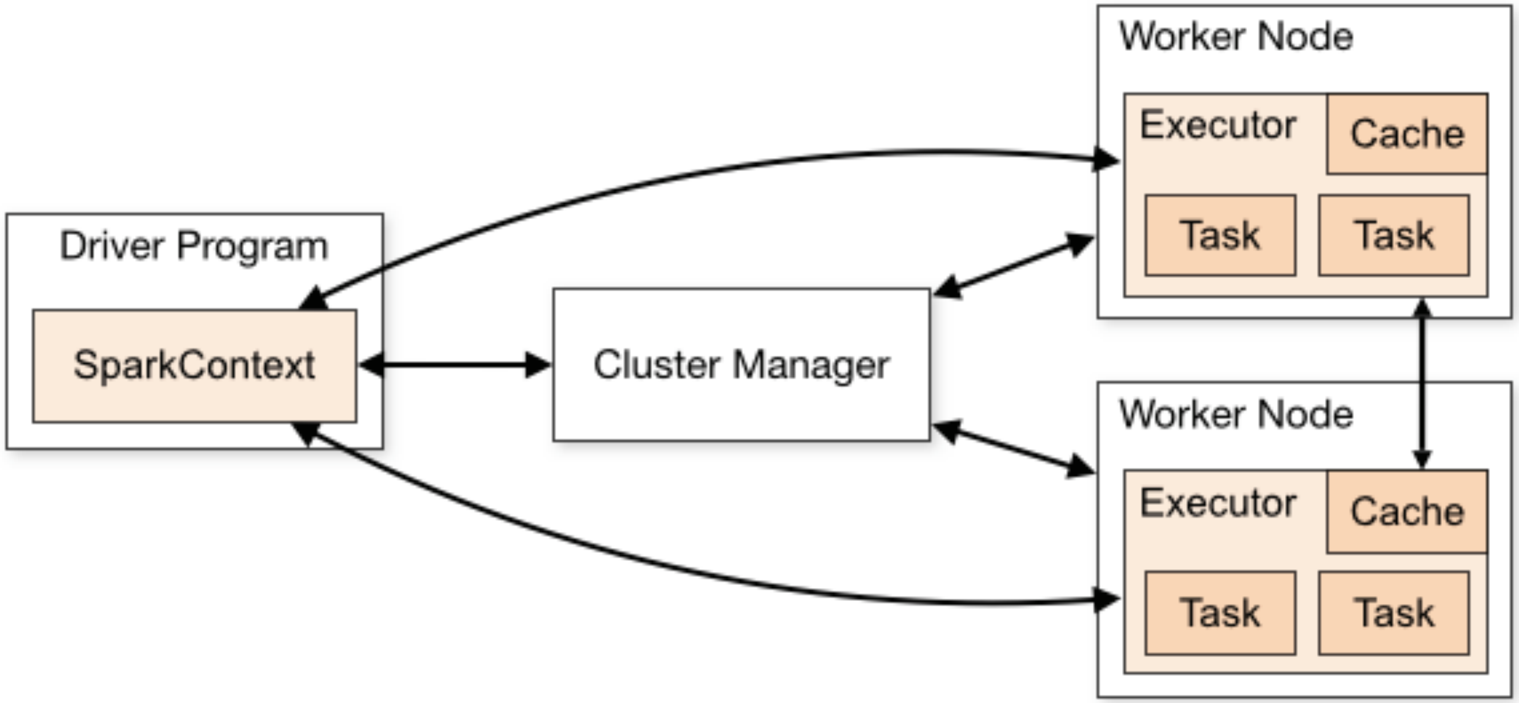
Spark基础

- Install
  - manual build
  - pre-built distribution

- Run
  - Requirements : jdk , scala
  - spark-shell
  - spark-submit

Spark基础

- cluster deploying



Spark基础

- RDD
  - A list of partitions
  - A function for computing each split
  - A list of dependencies on other RDDs
  - Optionally, a Partitioner for key-value RDDs
  - Optionally, a list of preferred locations to compute each split on

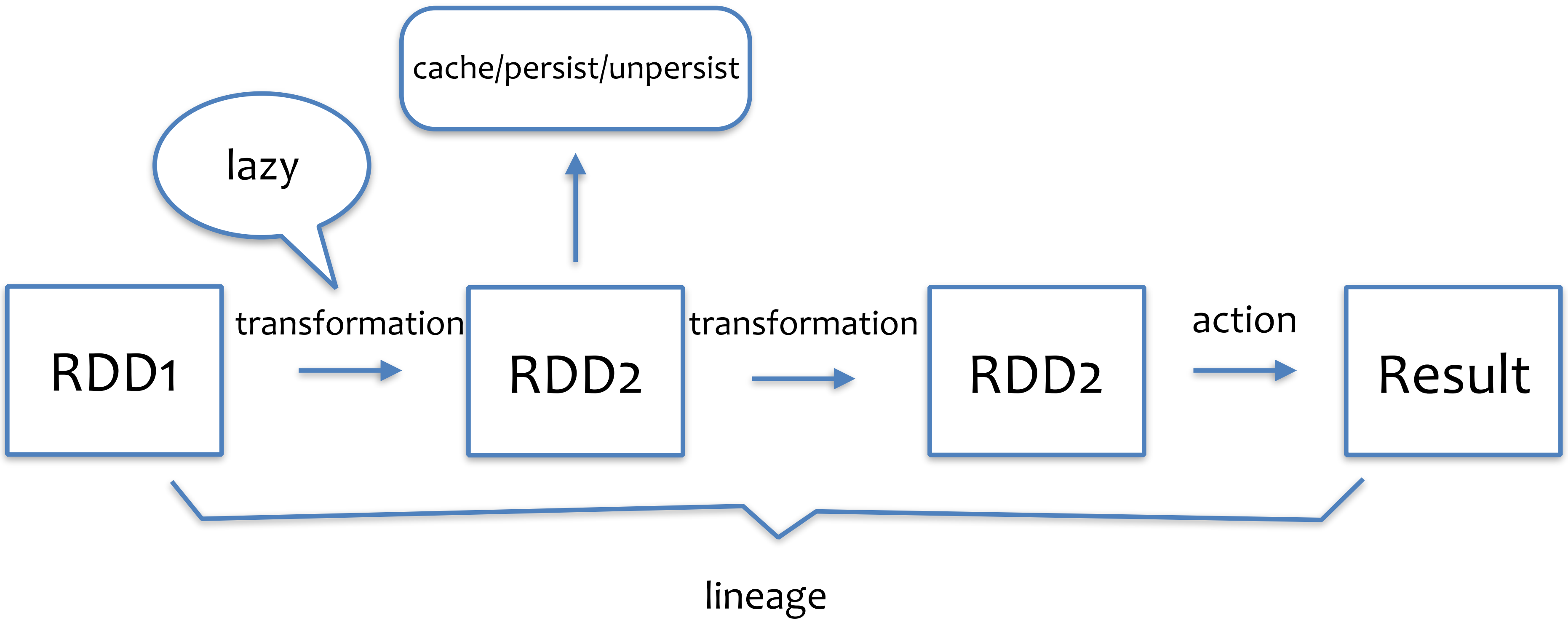
《Resilient Distributed Datasets:
A Fault-Tolerant Abstraction for In-Memory Cluster Computing》

Spark 基础

- ➢ RDD API
  - ➢ Transformations
  - ➢ Actions

Spark 基础

➢ programming model



Spark 基础

- ➢ How to create RDD
  - ➢ from Collection

```
val a = sc.parallelize(1 to 9, 3)
```

  - ➢ from File

```
val b = sc.textFile(“/path/to/file”)
```

  - ➢ from other RDDs

```
val c = b.map(line => line.split(“,”))
```

Spark 基础

➢ transformations

map(f: T => U)	sortBy[K](f: (T) => K)
mapPartitions(f: Iterator[T] => Iterator[U])/ mapPartitionsWithIndex(f: (Int, Iterator[T]) => Iterator[U])	repartition(numPartitions)/ coalesce(numPartitions)
keyBy(f: T => K)	glom():RDD[Array[T]]
flatMap(f: T => Iterable[U])	cartesian(RDD[U]): RDD[(T, U)]
filter(f: T => Boolean)	groupBy[K](f: T => K): RDD[(K, Iterable[T])]
distinct(numPartitions)	pipe(command: String)
intersection(RDD[T])	persist/cache
sample()/randomSplit()/takeSample()	zip(RDD[U]): RDD[(T, U)]
union(RDD[T])	join(RDD[(K,V)]): RDD([K, V)]

Spark 基础

➢ Actions

reduce(f: (T, T) => T)	top(n)(ordering)
collect(): Array[T]	max()/min()
count()	saveAsTextFile(path)
first()	foreach(f: T => Unit)
take(num): Array[T]	subtract(RDD[T])
zipWithIndex(): RDD[(T, Long)]/ zipWithUniqueId(): RDD[(T, Long)]	aggregate(zeroValue: U)(seqOp: (U, T) => U, combOp: (U, U) => U)

Spark 基础

- map

map是对RDD中的每个元素都执行一个指定的函数来产生一个新的RDD。任何原RDD中的元素在新RDD中都有且只有一个元素与之对应。

```
scala> val a = sc.parallelize(1 to 9, 3)
scala> val b = a.map(x => x*2)
scala> a.collect
res10: Array[Int] = Array(1, 2, 3, 4, 5, 6, 7, 8, 9)
scala> b.collect
res11: Array[Int] = Array(2, 4, 6, 8, 10, 12, 14, 16, 18)
```

Spark 基础

➢ flatMap

与map类似，区别是原RDD中的元素经map处理后只能生成一个元素，而原RDD中的元素经flatmap处理后可生成多个元素来构建新RDD。

```
scala> val a = sc.parallelize(1 to 4, 2)
scala> val b = a.flatMap(x => 1 to x)
scala> b.collect
res0: Array[Int] = Array(1, 1, 2, 1, 2, 3, 1, 2, 3, 4)
```

Spark 基础

➢ reduce

reduce将RDD中元素两两传递给输入函数，同时产生一个新的值，新产生的值与RDD中下一个元素再被传递给输入函数直到最后只有一个值为止。

```
scala> val c = sc.parallelize(1 to 10)
scala> c.reduce((x, y) => x + y)
res4: Int = 55
```

Spark 基础

➢ reduceByKey

reduceByKey对元素为KV对的RDD中Key相同的元素的Value进行reduce，因此，Key相同的多个元素的值被reduce为一个值，然后与原RDD中的Key组成一个新的KV对。

```
scala> val a = sc.parallelize(List((1,2),(3,4),(3,6)))
scala> a.reduceByKey((x,y) => x + y).collect
res7: Array[(Int, Int)] = Array((1,2), (3,10))
```

Spark 基础

➢ saveAsTextFile/saveAsSequenceFile/saveAsObjectFile

将 RDD 输出到存储介质。

```
scala> val a = sc.parallelize(List((1,2),(3,4),(3,6)))
scala> a.reduceByKey((x,y) => x + y).saveAsTextFile(“hdfs://0.0.0.0:9000/test”)
```

Spark 基础

➢ other APIs

join: 对类型为(K,V)的 RDD 进行关联，返回 RDD 类型为(K, (V,V))

```
scala> val kv1=sc.parallelize(List(("A",1),("B",2),("C",3),
("A",4),("B",5)))
scala> val kv3=sc.parallelize(List(("A",10),("B",20),("D",
30)))
scala> kv1.join(kv3).collect
```

cogroup: 对两个类型为(K,V)的RDD进行联合 group，返回 RDD 类型为(K, (Iterable, Iterable))

```
scala> kv1.cogroup(kv3).collect
```

Spark 基础

➢ other APIs

cartesian: 求两个 RDD 的笛卡尔积

```
scala> val rdd1=sc.parallelize(Array(1,2,3,4))
scala> val rdd2=sc.parallelize(Array(5,6))
scala> rdd1.cartesian(rdd2).collect
```

coalesce: 将 RDD 的分区数减至指定数量

```
scala> val rdd3 = rdd1.coalesce(2, false)
```

Spark 基础

➢ other APIs

union: 求两个 RDD 的笛卡尔积

```
scala> val a = sc.parallelize(1 to 3, 1)
scala> val b = sc.parallelize(5 to 7, 1)
scala> val c = a.union(b).collect
```

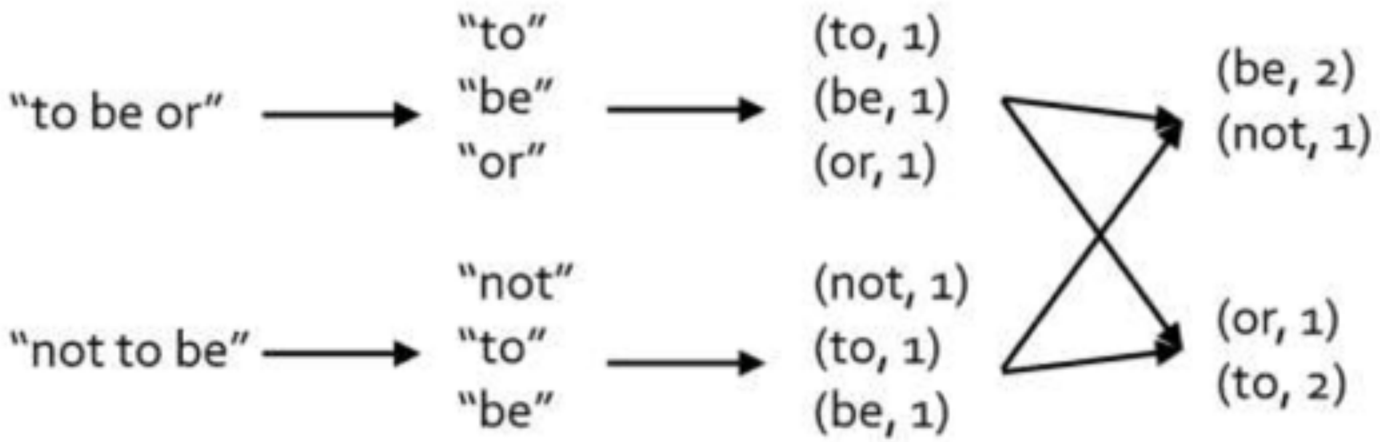
subtract: 将 RDD 的分区书减至指定数量

```
scala> val a = sc.parallelize(1 to 9, 3)
scala> val b = sc.parallelize(1 to 3, 3)
scala> val c = a.subtract(b).collect
```

Spark 基础

➢ CASE1: word count

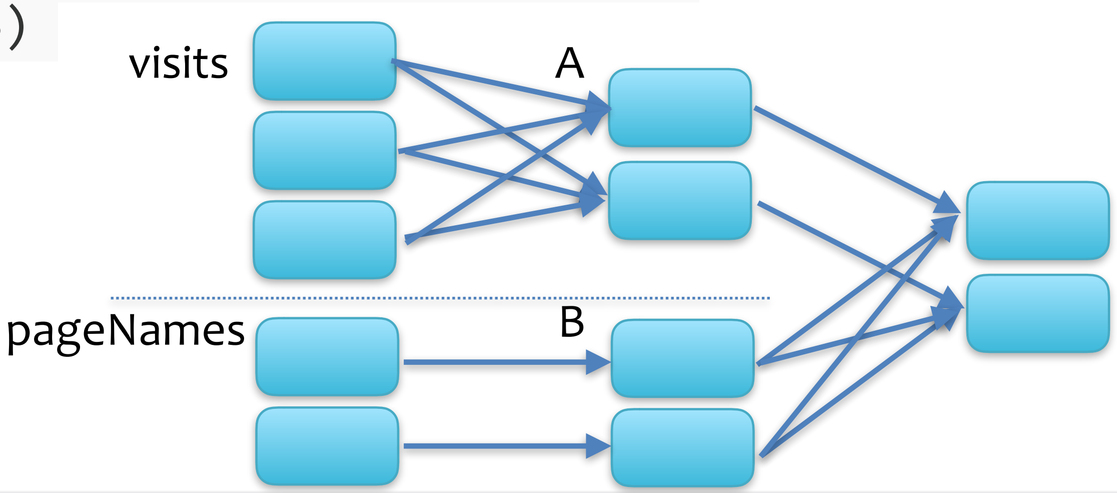
```
lines = sc.textFile(“wordcount.txt”)
counts = lines.flatMap(line => line.split(“ “))
              .filter(_.length > 1)
             .map(word => (word, 1))
             .reduceByKey((x, y) => x + y)
```



Spark 基础

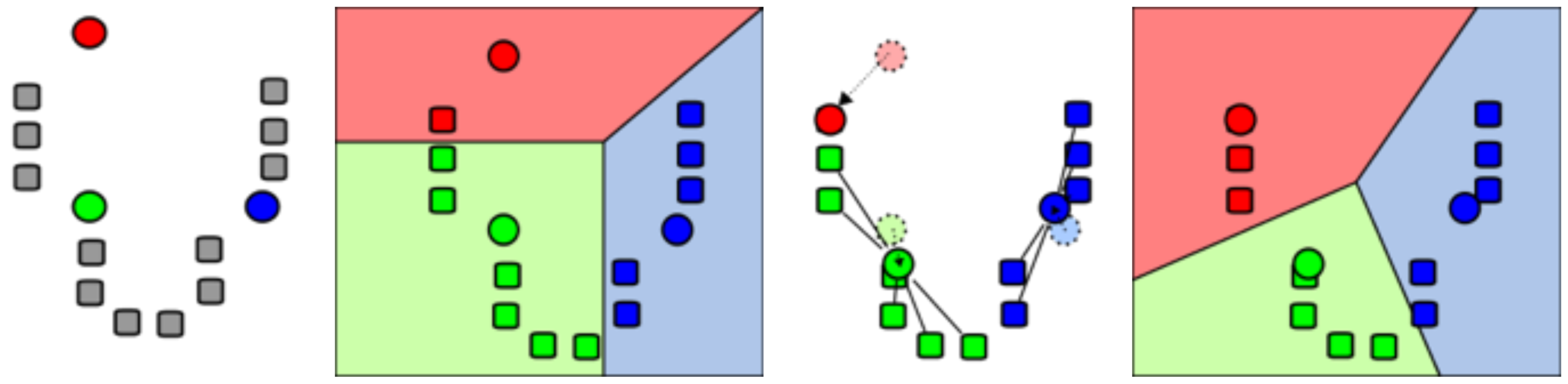
➢ CASE2: join

```
val visits = sc.parallelize(List(“index.html”, “1.2.3.4”),
                          (“about.html”, “3.4.5.6”),
                          (“index.html”, “1.3.3.1”))
val pageNames = sc.parallelize([ (“Home”, “index.html”),
                                 (“About”, “about.html”) ])
val A = visits.groupByKey()
val B = pageNames.map(x => x._2 -> x._1)
val C = A.join(B)
C.collect
```



Spark 基础

- CASE3: K-means



1 2 3 4

Spark 基础

➢ case3: K-means

$$V = \sum_{i=1}^{k} \sum_{x_j \in S_i} (x_j - \mu_i)^2$$

1. 从N个文档随机选取K个文档作为聚类中心；
2. 对剩余的每个文档测量其到每个聚类中心的距离，并把它归到最近的聚类；
3. 重新计算已经得到的各个聚类的中心；
4. 迭代2～3步直至新的聚类中心与原聚类中心相等或小于指定阈值，算法结束。

Spark 作业

- 使用 spark api 实现倒排索引 (Inverted index)

  倒排索引源于实际应用中需要根据属性的值来查找记录。这种索引表中的每一项都包括一个属性值和具有该属性值的各记录的地址。由于不是由记录来确定属性值，而是由属性值来确定记录的位置，因而称为倒排索引(inverted index)。

- 构建方法

  1. 将文档分析成单词term并标记
  2. 按单词进行合并计算
  3. 对单词生成倒排列表

➢ 输入：

近2000篇英文文档，每篇文档分配一个整数ID，每篇英文文档已经去掉了标点符号等非字母和非数字的字符，也去掉了停用词以及由纯数字组成的字符串（诸如年份、价格等字符串），同时每个单词均是小写并做完stemming处理。

➢ 要求：建立倒排表

倒排表的JSON格式定义为：

{w1: [{ w1_d1: [w1_d1_p1, d1_p2] },{ w1_d2: [w1_d2_p1] },{ w1_d3: [w1_d3_p1] }]}

{w2: [{ w2_d1: [w1_d1_p1, d1_p2, d1_p3] },{ w2_d2: [w1_d2_p1] }]}

如上倒排表所示，单词w1出现在三篇文档中，文档的ID分别：w1_d1、w1_d2、w1_d3。该单词w1在文档w1_d1中出现了两次，出现的位置分别为w1_d1_p1和d1_p2，在文档w1_d2和w1_d3中哥出现一次，其出现位置分别为w1_d2_p1和w1_d3_p1。

➢ 输出：

所有DF=10的单词及其对应的倒排表键值，输出格式每一行要求为一个合法的JSON格式字符串。

➢ 工程及数据：

https://github.com/wuti1609/spark_demo.git

NLP

- TF-IDF

  TF-IDF (term frequency-inverse document frequency) is a way to score the importance of terms in a document based on how frequently they appear across a collection of documents (corpus).

  TF-IDF 是一种排序方法，通过一个词项在文档集中出现的频次来计算

- TF-IDF
  - 建立一个搜索引擎大致需要几件事：
    - 自动下载尽可能多的网页
      - 遍历算法、提取 URL、哈希表记录
    - 建立快速有效的索引
      - 布尔代数、运算
      - 关键词用二进制数表示
    - 根据相关性对网页进行排序
      - 网页的质量信息—PageRank
      - 关键词与页面的相关性—TF-IDF

- TF (term frequency)
  - 关键词在该网页中出现的次数/该网页的总字数

$$\mathrm{tf}(t,d) = 0.5 + \frac{0.5 \times \mathrm{f}(t,d)}{\max\{\mathrm{f}(t,d) : t \in d\}}$$

- IDF (inverse document frequency)
  - 如果一个词在很少的网页中出现，通过它就很容易定位搜索目标，它的权重也就应该大；反之，如果一个词在大量网页中出现，它的权重就小。

$$\mathrm{idf}(t, D) = \log \frac{N}{|\{d \in D : t \in d\}|}$$

- TF-IDF
  - 随词项的频率的增大而增大
  - 随词项的普遍度的增大而减小

$$\mathrm{tfidf}(t, d, D) = \mathrm{tf}(t, d) \times \mathrm{idf}(t, D)$$