分布式计算框架Spark

Grissom | 2025年11月

TRANSWARP
星 环 科 技

目录》
CONTENTS

Spark 简介

- MapReduce有较大的局限性
  - 仅支持Map、Reduce两种语义操作，划分为两个阶段（模型较为粗糙）
  - 执行效率低，时间开销大（很难避免Shuffle）
  - 主要用于大规模离线批处理
  - 不适合迭代计算、在线分析、实时流处理等场景
- 计算框架种类多，选型难，学习成本高
  - 批处理：MapReduce
  - 流处理：Storm、Flink
  - 在线分析：Impala、Presto
  - 机器学习：Mahout
- 统一计算框架，简化技术选型，降低学习成本
  - 在统一框架下，实现离线批处理、流处理、在线分析和机器学习

- 由加州大学伯克利分校AMP实验室（Algorithms、Machines and People Lab）开发
- 专为大规模数据处理而设计的快速通用的内存计算引擎
- 基于Scala语言开发
- 官网地址：http://spark.apache.org/

2016年7月 Spark2.0.0正式版发布

2016年5月 Spark2.0.0Preview发布

2016年1月 Spark1.6.0

2015年9月 Spark1.5.0

2015年3月 Spark1.3.0

2014年5月 Spark1.0.0发布

8个月

2014年2月 成为Apache顶级项目

2013年6月 进入Apache孵化器项目

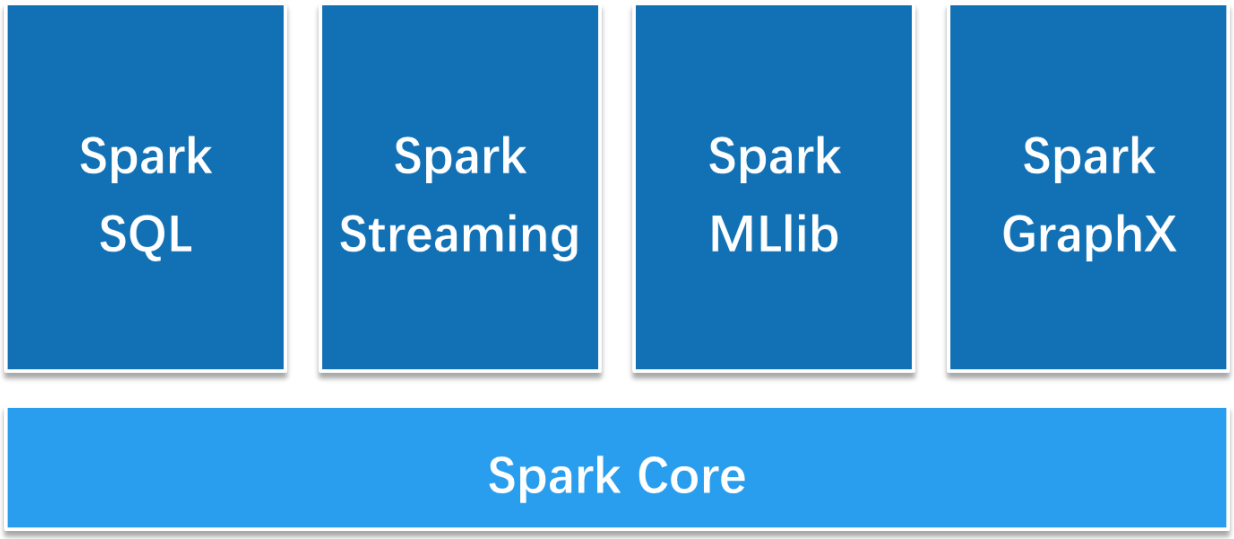
2012年2月 发布0.6.0版本

2010年 开放源代码

2009年 Berkeley's AMPLab开始编写最初源代码

➢ Spark

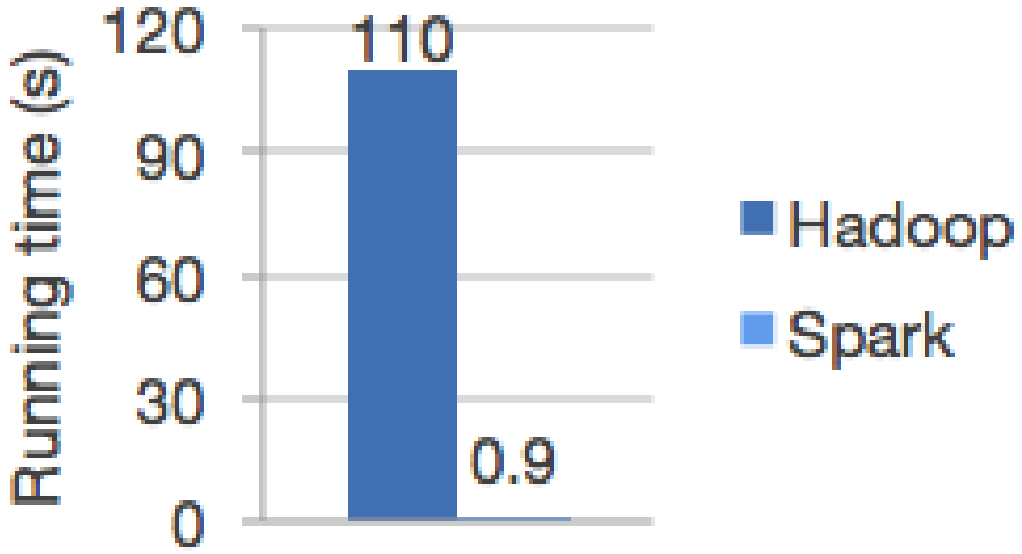
- Spark Core
  - Spark的基础核心，实现了Spark的基本功能，如：内存计算、任务调度、部署模式、故障恢复、存储管理等。
  - 将分布式数据抽象为弹性分布式数据集（RDD），Spark建立在统一的RDD基础之上
- Spark SQL
  - 提供类SQL方式处理、操作RRD，进行复杂数据分析
- Spark Streaming
  - 基于微批模式的实时大数据分析引擎
- Spark MLlib
  - 机器学习，提供了包括聚类、分类、回归、协同过滤等的分布式实现，大幅降低了机器学习的门槛
- Spark GraphX
  - 分布式图计算



➢ Spark优势

Speed
快

- 内存计算
- DAG引擎



➢ Spark优势

Ease of Use
易用

- 丰富的编程接口支持
  - Scala、Java、Python、R、SQL
- 丰富的算子支持 （>80）

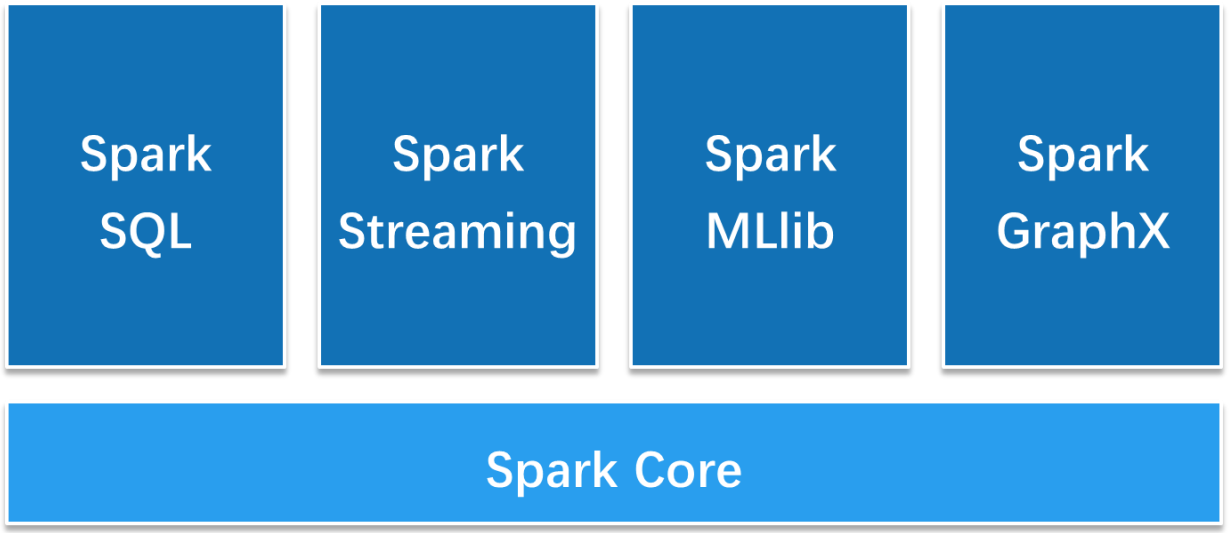
```
df = spark.read.json("logs.json")
df.where("age > 21")
  .select("name.first").show()
```

Spark's Python DataFrame API
Read JSON files with automatic schema inference

- Spark优势

Generality

通用



➢ Spark优势

Runs Everywhere

兼容

多种运行方式，不同的资源调度管理系统

- Local、Standalone
- Yarn
- Mesos、EC2、Kubernets

支持多种数据源

- HDFS、HBase、Hive、Cassandra

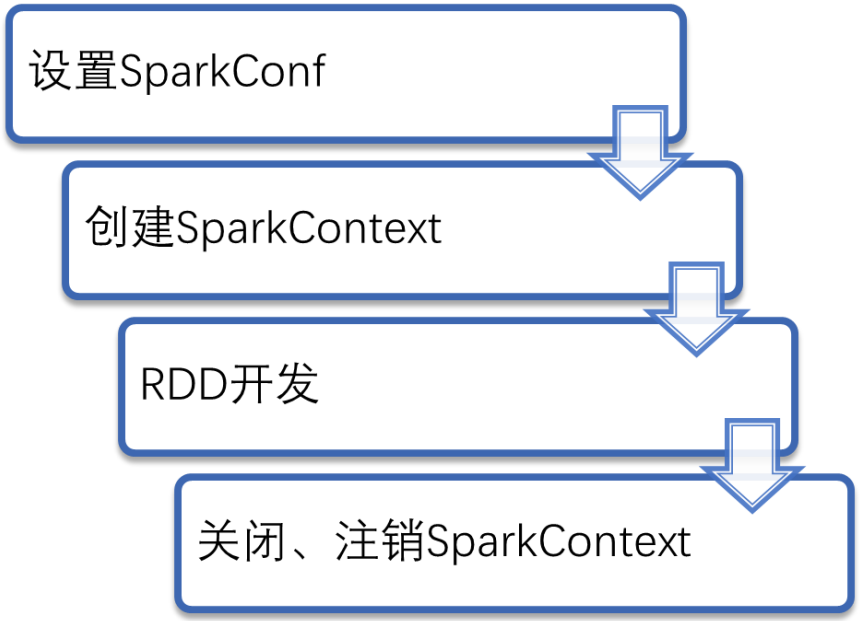
2
chapter

Spark Core

- ✓ Spark 编程模型
- ✓ Spark 运行机制
- ✓ Spark 集群部署
- ✓ 作业监控

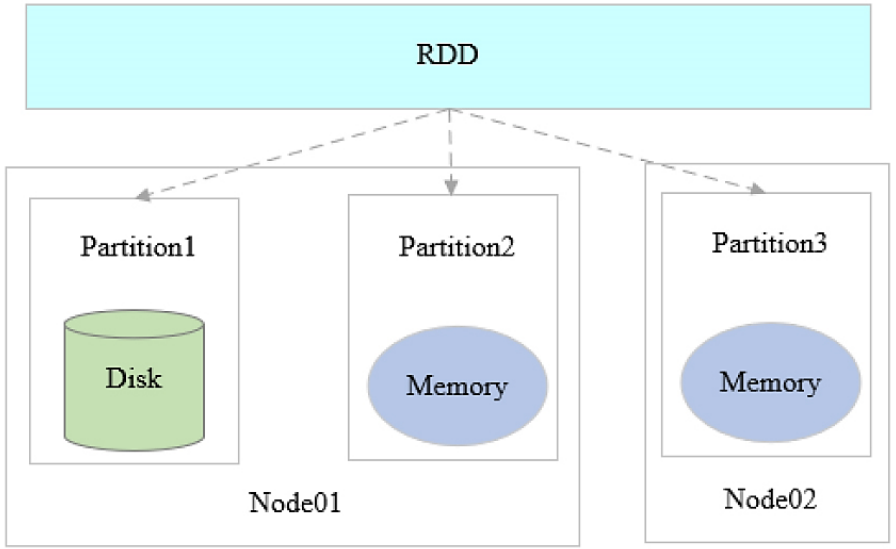
➢ 案例：WordCount

```
import org.apache.spark.{SparkConf, SparkContext}
object WordCount{
  def main(args: Array[String]): Unit = {

    //1.创建SparkConf并设置App名称
    val conf = new SparkConf().setAppName("WC")

    //2.创建SparkContext，该对象是提交Spark App的入口
    val sc = new SparkContext(conf)

   //3.使用sc创建RDD并执行相应的transformation和action
    val rdd1 = sc.textFile(args(0))

    val rdd2 = rdd1.flatMap(_.split(" ")).map((_, 1)).reduceByKey(_+_)

    rdd2.saveAsTextFile(args(1))

    //4.关闭连接
    sc.stop()
  }
}
```



➢ RDD

- 弹性分布式数据集（Resilient Distributed Datesets）
  - 分布在集群中的只读对象集合
  - 由多个Partition组成
  - 通过转换操作构造
  - 失效后自动重构（弹性）
  - 存储在内存或磁盘中
- Spark基于RDD进行计算



➢ RDD的特性：

- ✓ 一组分区（Partition），即数据集的基本组成单位
- ✓ 一个计算每个分区的函数
- ✓ RDD之间的依赖关系
- ✓ 一个Partitioner，即RDD的分片函数
- ✓ 一个列表，存储存取每个Partition的优先位置（preferred location）

```
* Internally, each RDD is characterized by five main properties:
*
*  - A list of partitions
*  - A function for computing each split
*  - A list of dependencies on other RDDs
*  - Optionally, a Partitioner for key-value RDDs (e.g. to say that the RDD is hash-partitioned)
*  - Optionally, a list of preferred locations to compute each split on (e.g. block locations for
*    an HDFS file)
```

- RDD的创建：

① 从集合中创建

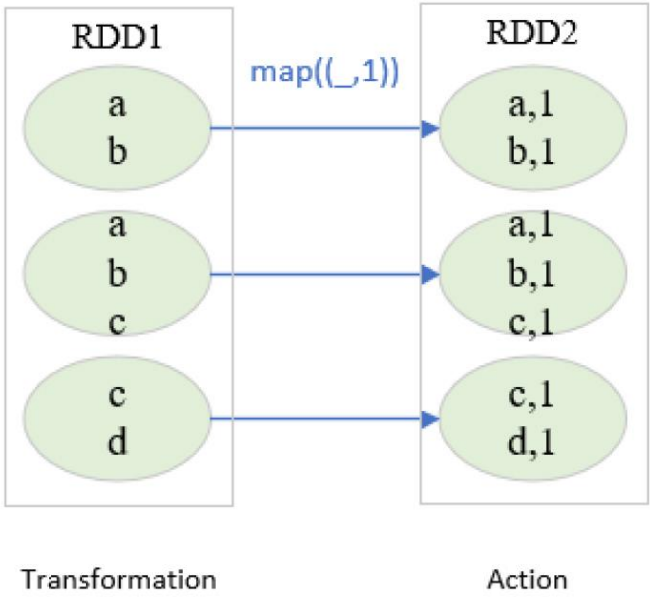
```
val rdd = sc.parallelize(Array(1,2,3,4,5,6,7,8))

val rdd1 = sc.makeRDD(Array(1,2,3,4,5,6,7,8))
```

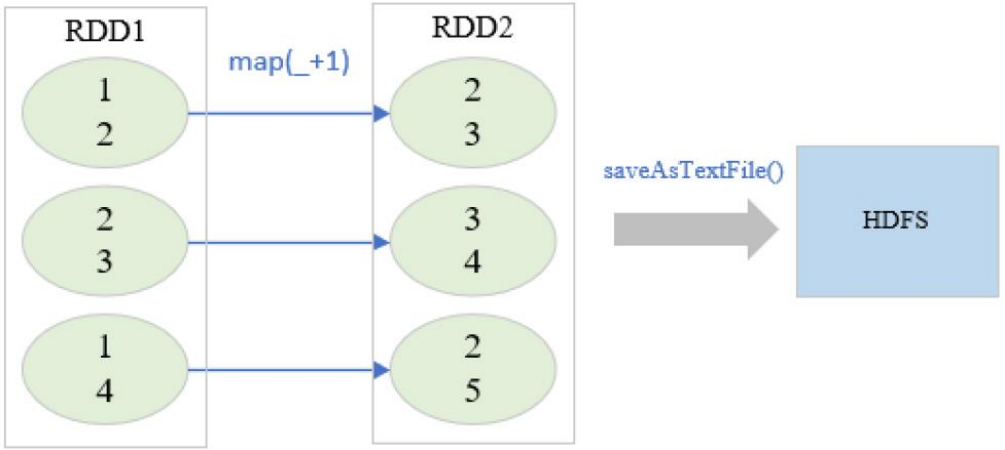
② 由外部存储系统的数据集创建

```
val rdd2= sc.textFile("hdfs://192.168.6.101:9000/xxx")
```

③ 从其他RDD创建

- RDD操作（算子）
  - Transformation（转换）
    - 将Scala集合或Hadoop输入数据构造成一个新RDD
    - 通过已有的RDD产生新RDD
    - 惰性执行：只记录转换关系，不触发计算
    - 例如：map、filter、flatmap、union、distinct、sortbykey
  - Action（动作）
    - 通过RDD计算得到结果或者落盘
    - 真正触发计算
    - 例如：first、count、collect、foreach、saveAsTextFile
- 示例：RDD的两种操作
  - rdd1.map(_,+1).saveAsTextFile(“hdfs://node01:9000”)





➢ RDD操作 —— 转换Transform

- 示例：

```
var rdd  = sc.parallelize(1 to 10)
val maprdd = rdd.map(_ * 2)
maprdd.collect()
返回输出：Array[Int] = Array(2, 4, 6, 8, 10, 12, 14, 16, 18, 20)
```

Value类型

- map
- mapPatririons
- mapPartitionsWithIndex
- flatMap
- glom
- groupBy
- filter
- sample
- distinct
- coalesce
- repartition
- sortBy

双Value类型

- intersection
- union
- subtract
- zip

Key-Value类型

- partitionBy
- reduceByKey
- groupByKey
- aggregateByKey
- foldByKey
- combineByKey
- sortByKey
- join
- leftOuterJoin & rightOuterJoin
- Cogroup

➢ RDD操作 —— 动作Action

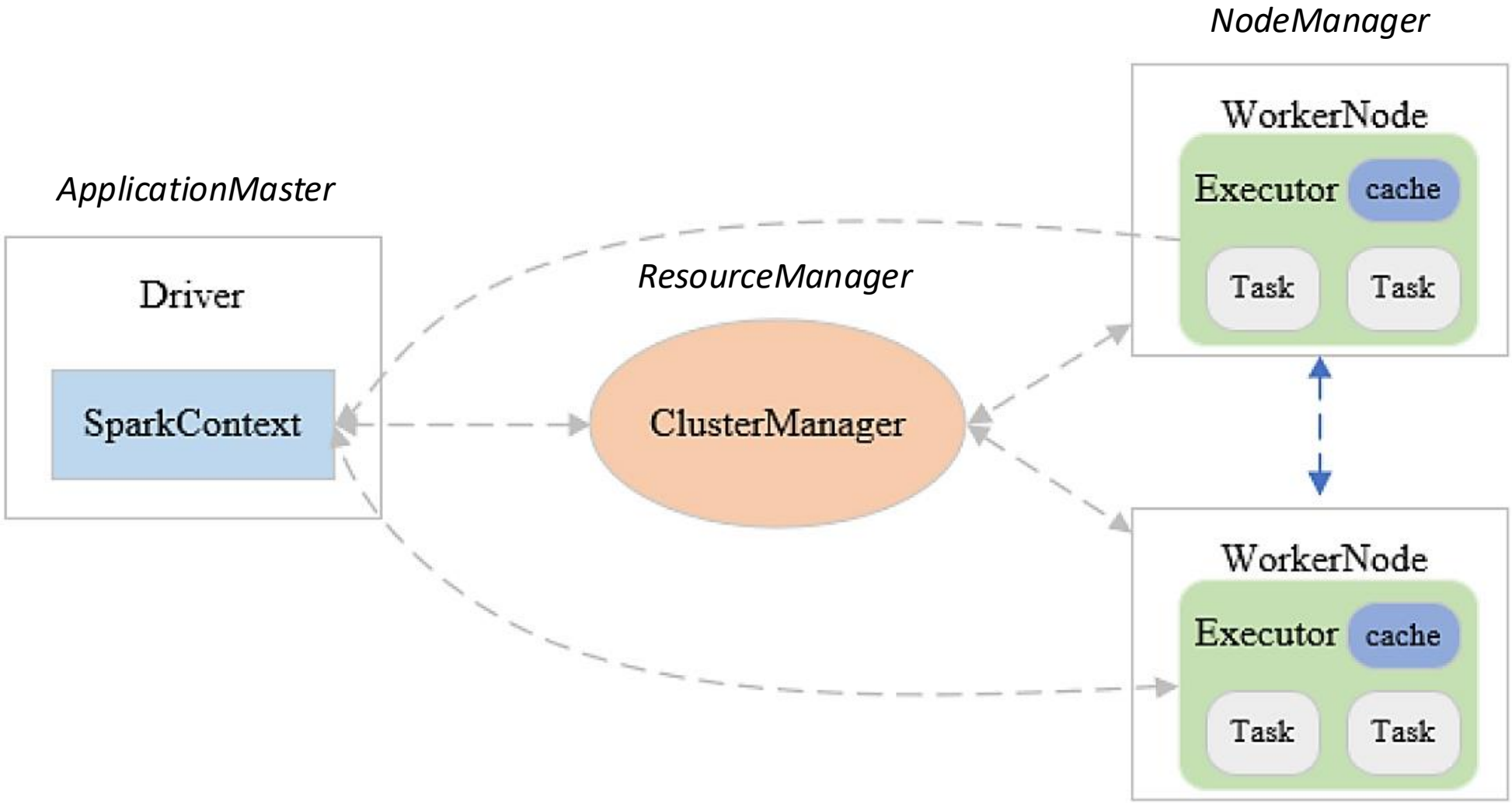
- 示例：

```
var rdd  = sc.parallelize(1 to 10)
rdd.count
返回输出：Long = 10
```

Action

- reduce
- collect
- count
- first
- take
- takeOrdered
- aggregate
- fold
- countByKey
- save相关算子
- foreach

- Spark运行架构


NodeManager
WorkerNode
Executor
cache
Task
Task
ApplicationMaster
Driver
SparkContext
ResourceManager
ClusterManager
WorkerNode
Executor
cache
Task
Task

➢ Spark运行架构

- Application - 应用程序

  是指用户编写的Spark应用程序

  包含驱动程序（Driver）、分布在集群中多个节点上运行的Executor代码

  在执行过程中，有一个或多个Job作业组成（*通常为一个Job*）

- Driver - 驱动程序

  运行Application程序中的Main函数

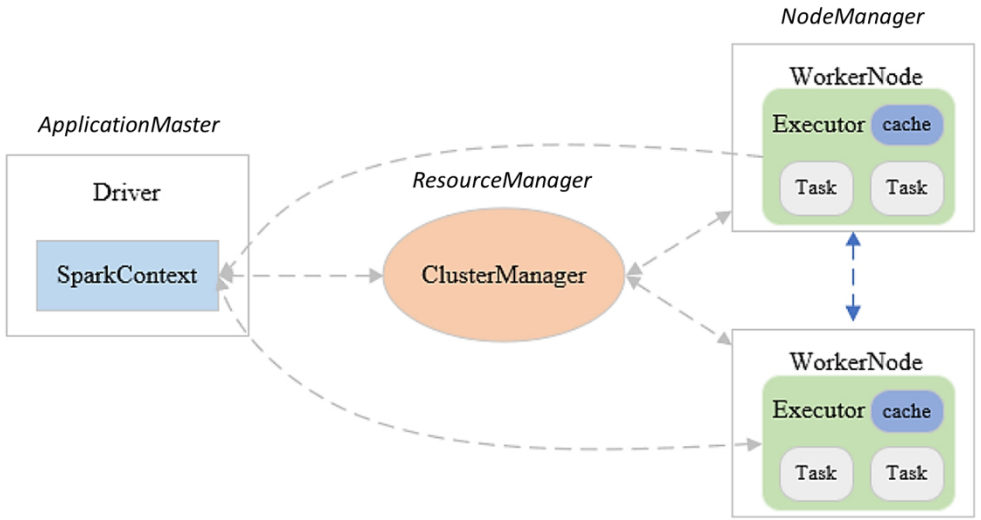
  创建SparkContext，为Application准备运行环境

- Cluster Manager – 集群资源管理器

  在集群上获取资源的外部服务

  包括：

  ① Standalone
  ② Yarn
  ③ Mesos



➢ Spark运行架构

- Worker Node – 工作节点

  集群中运行Application程序的节点

  例如：

  ① Standalone —— Worker

  ② Yarn —— NodeManager

- Executor – 执行进程

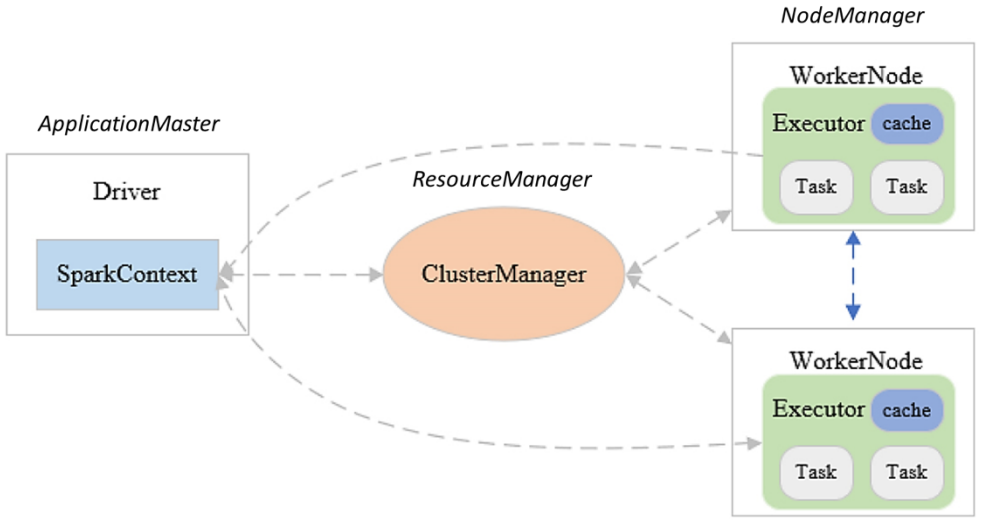
  运行在Worker Node工作节点之上的**进程**

  负责运行Task任务、计算数据的存储

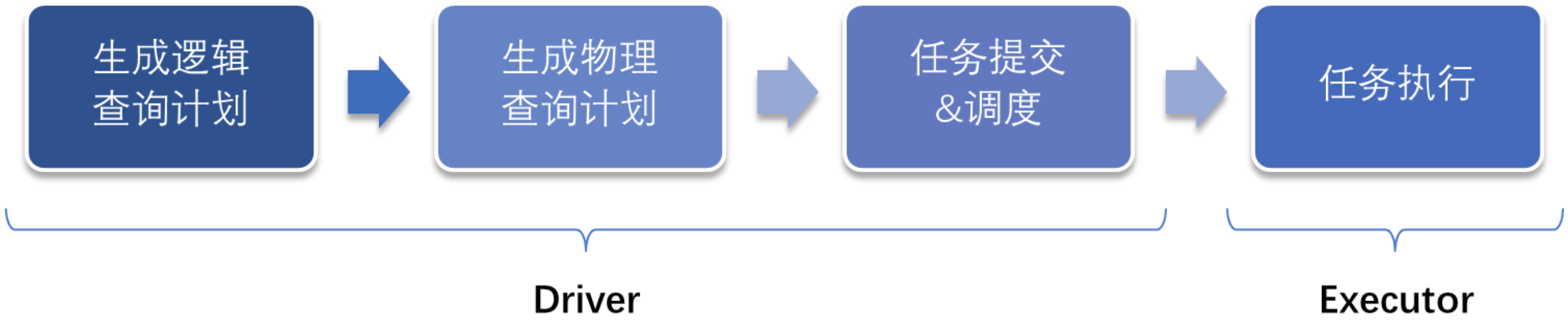
  每个Application都有各自独立的一批Executor

- Task – 任务

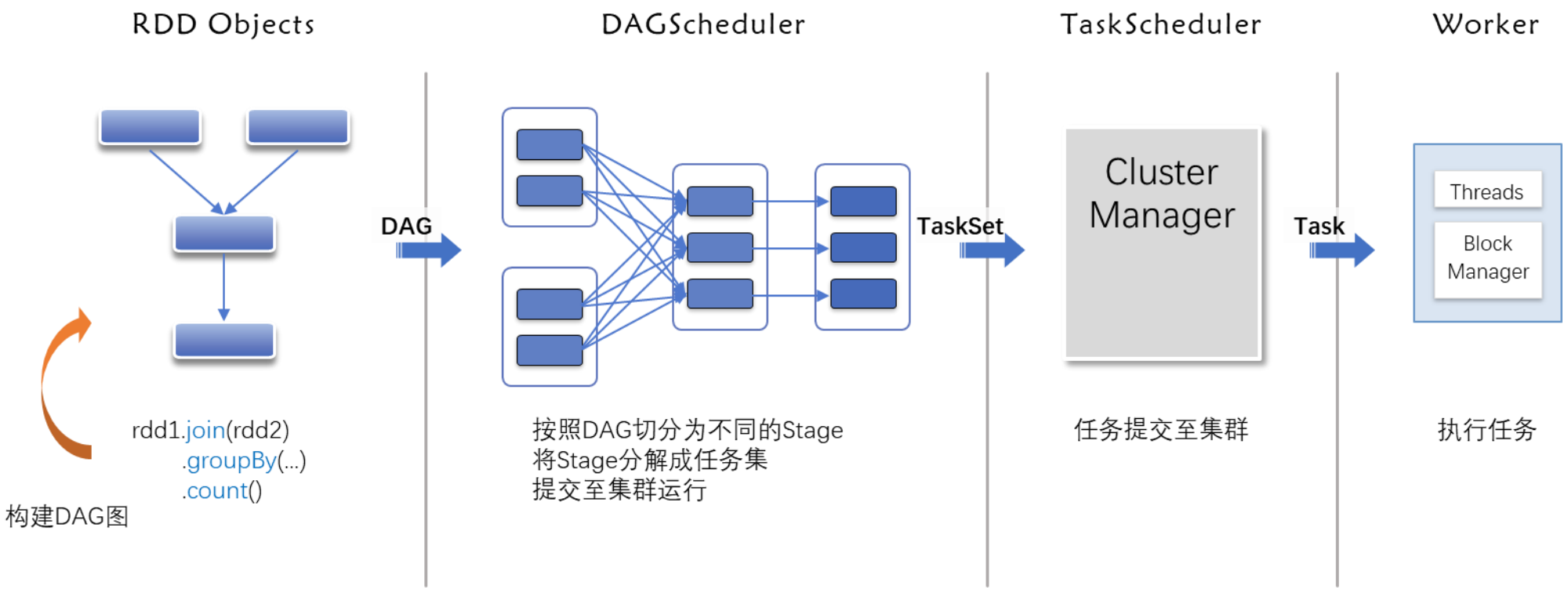
  Spark实际执行应用程序的最小单元



- Spark程序执行流程



➢ Spark程序执行流程

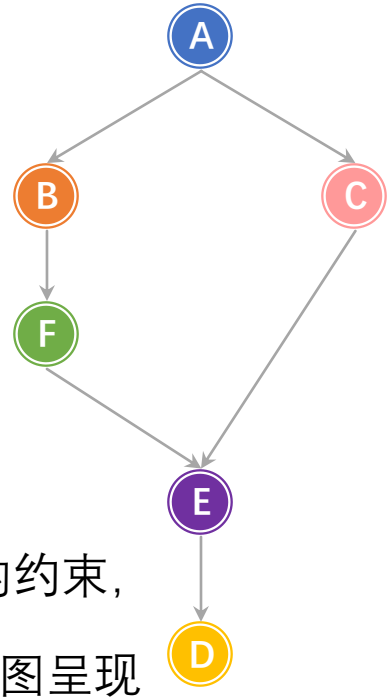


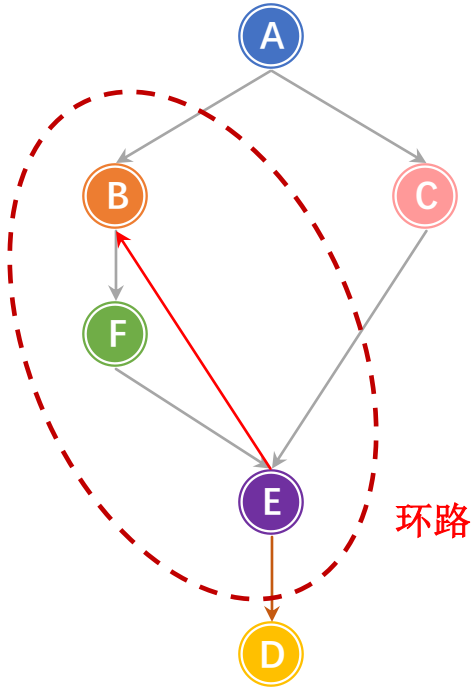
➢ 核心概念：

- **DAG** – 有向无环图，用来描述Spark程序内部的计算逻辑
- **Job** – RRD中有Action操作所生产的一个或多个Stage调度阶段
- **Stage** – 每个Job作业根据RRD之间依赖关系拆分成的任务集TaskSet
- **Task** – 被分配到计算节点上的计算单元，Spark实际执行应用的最小单元
- **DAG Scheduler** – 接收Spark应用程序提交的Job作业，根据RDD依赖关系切分Stage，并提交给TaskScheduler
- **Task Scheduler** – 接收DAGScheduler的Stage（TaskSet），将Task任务分发给Worker Node节点运行

➢ DAG

- 有向无环图，Directed Acyclic Graph
  - ✓ 有向：有方向（同一个方向）
  - ✓ 无环：不产生闭环
- 一个有向图无法从任意顶点出发经过若干条边回到该点



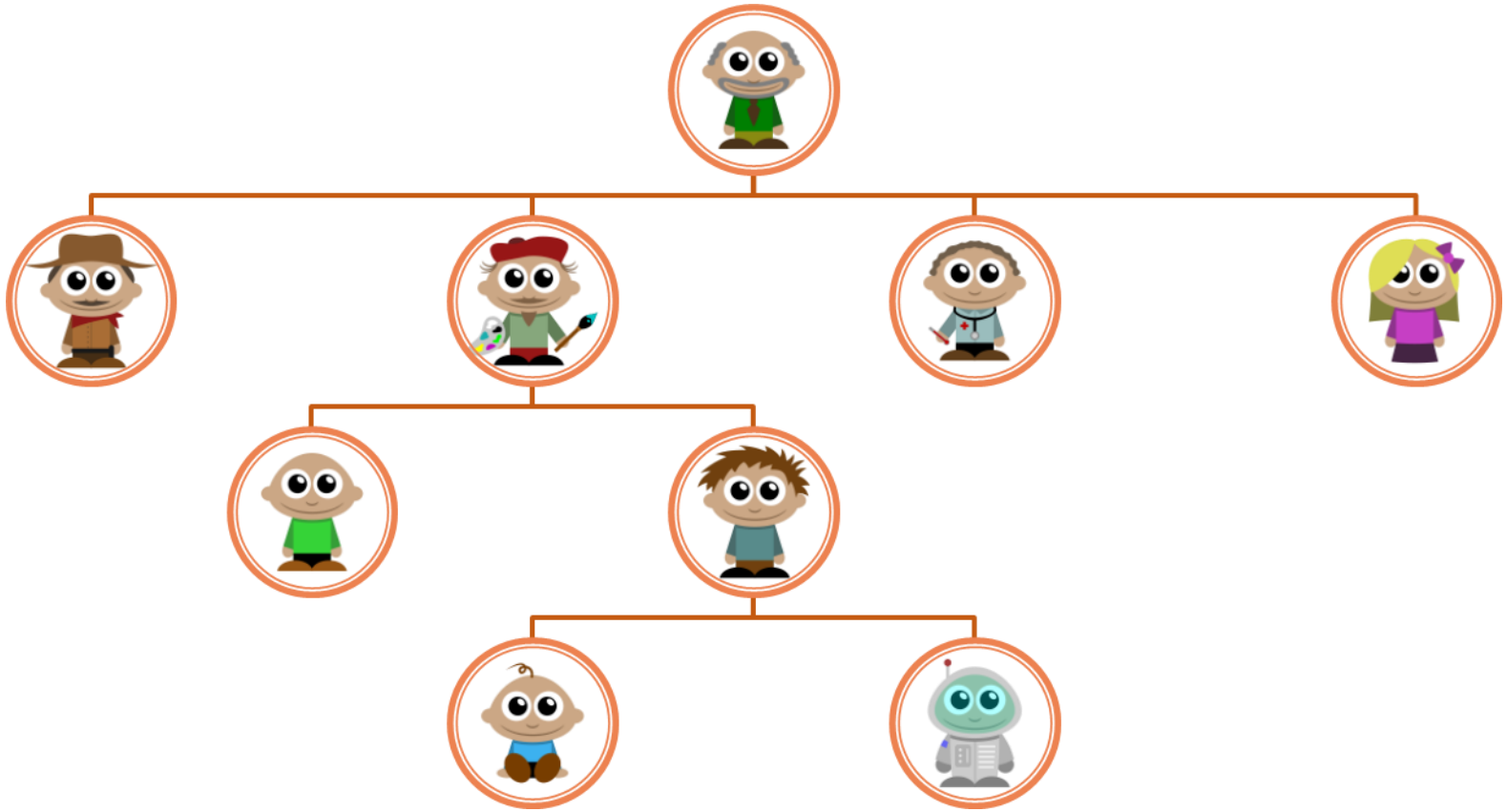


- Spark中受制于某些任务必须比另一些任务较早执行的约束，可排序为一个队列的**任务集合**，该队列可由一个DAG图呈现

- Spark程序的内部执行逻辑由DAG描述，顶点代表**Task任务**，边代表任务间的**依赖约束**

➢ RDD血缘Lineage

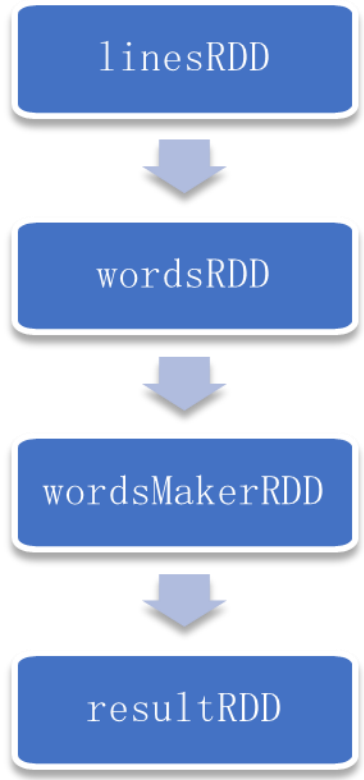
- 血缘关系，集中描述上下游之间相互的依赖关系



➢ RDD血缘Lineage

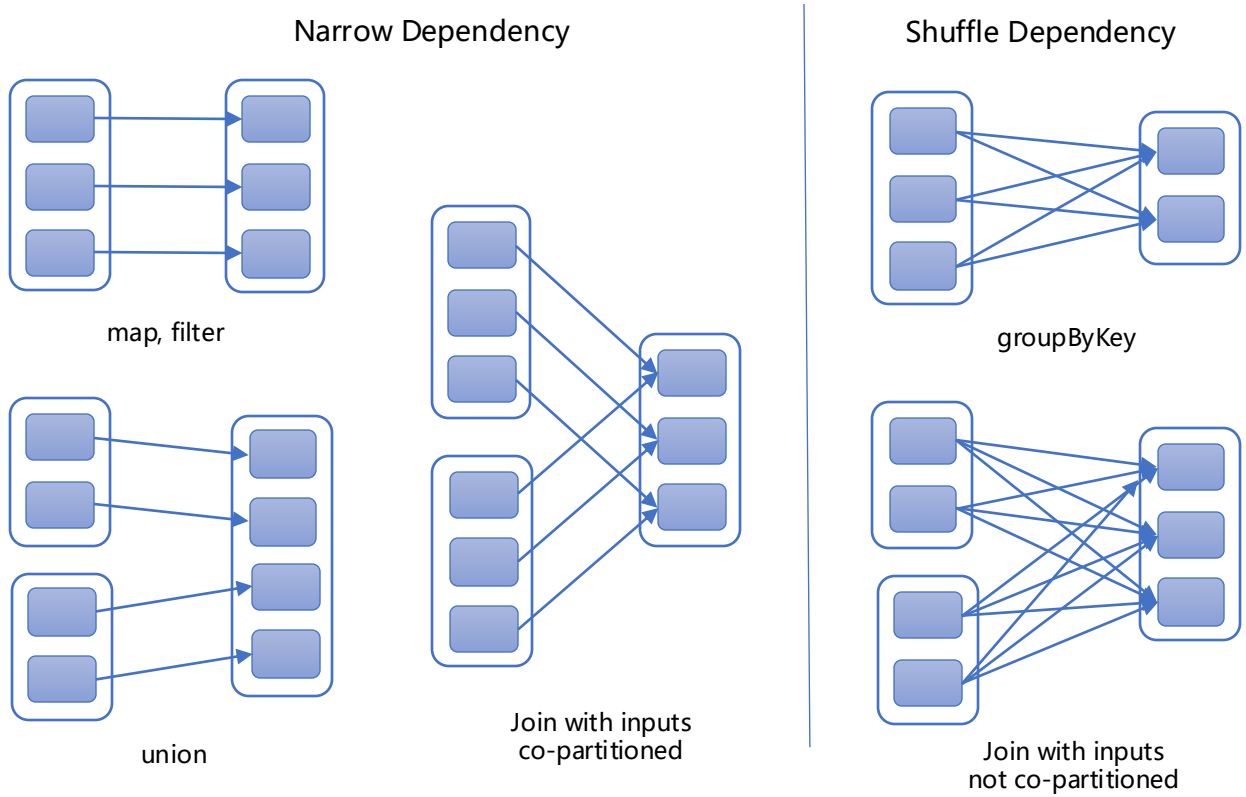
- RDD本质是延迟执行的
  - 例如，RDD之间的转换（flatMap），并不会立即执行
- Lineage记录了RDD之间相互的转换、依赖关系
- 容错机制（粗粒度）
  - 当RDD部分分区数据丢失时，可以进行重新运算、数据恢复

```
val config = new SparkConf()
val sc = new SparkContext(config)
val linesRDD = sc.textFile("input")
val wordsRDD = linesRDD.flatMap(_.split(" "))
val wordsMakerRDD = wordsRDD.map((_,1))
val resultRDD = wordsRemakeRDD.reduceByKey((_+_))
resultRDD.saveAsTextFile(“onput")
```



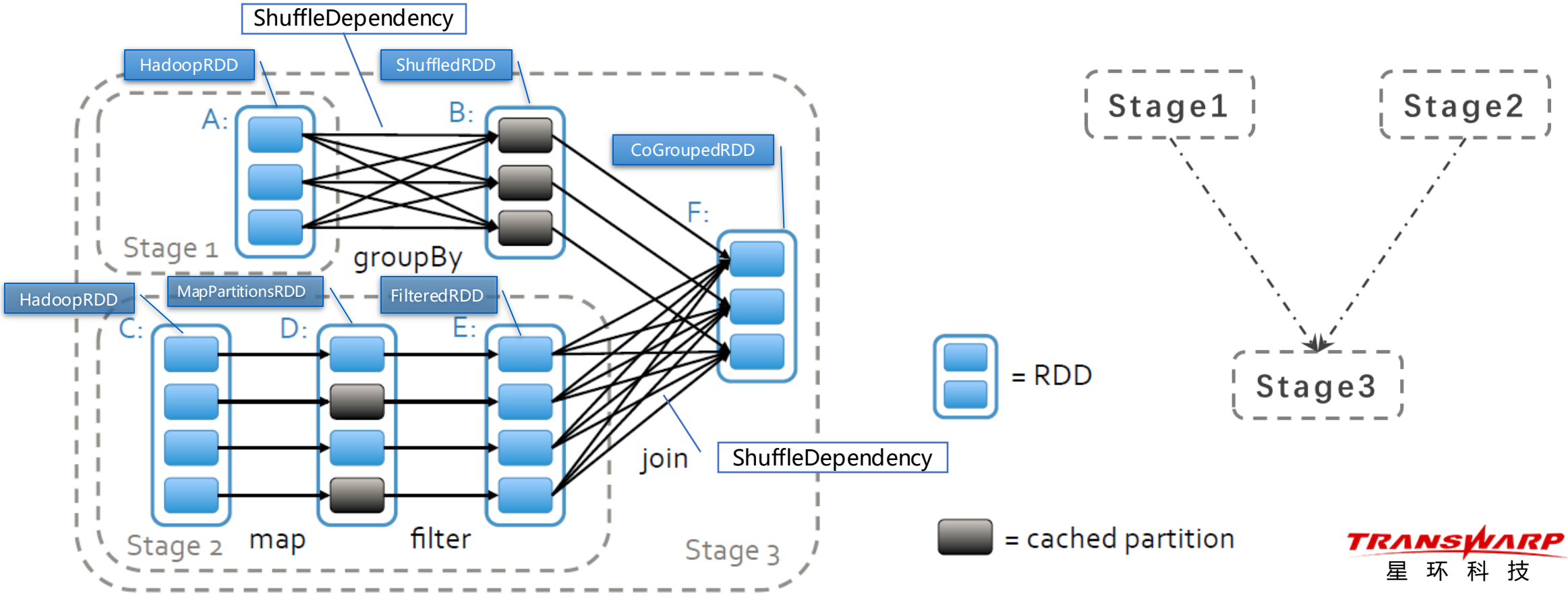
➢ RDD依赖（Dependency）

- 窄依赖（Narrow Dependency）
  - 每个父RDD分区只能为一个子RDD分区供数，子分区所依赖的父分区集合之间没有交集
  - 子RDD分区数据丢失或损坏，从其依赖的父RDD分区重新计算即可，无需Shuffle
  - 例如：map、filter、union
- 宽依赖（Wide/Shuffle Dependency）
  - 每个父RDD分区为所有子RDD分区供数
  - 子RDD区数据丢失或损坏，从所有父RDD分区重新计算，必须Shuffle
  - 相对于窄依赖，宽依赖付出的代价要高很多，尽量避免使用
  - 例如：groupByKey、reduceByKey、sortByKey



➢ Stage

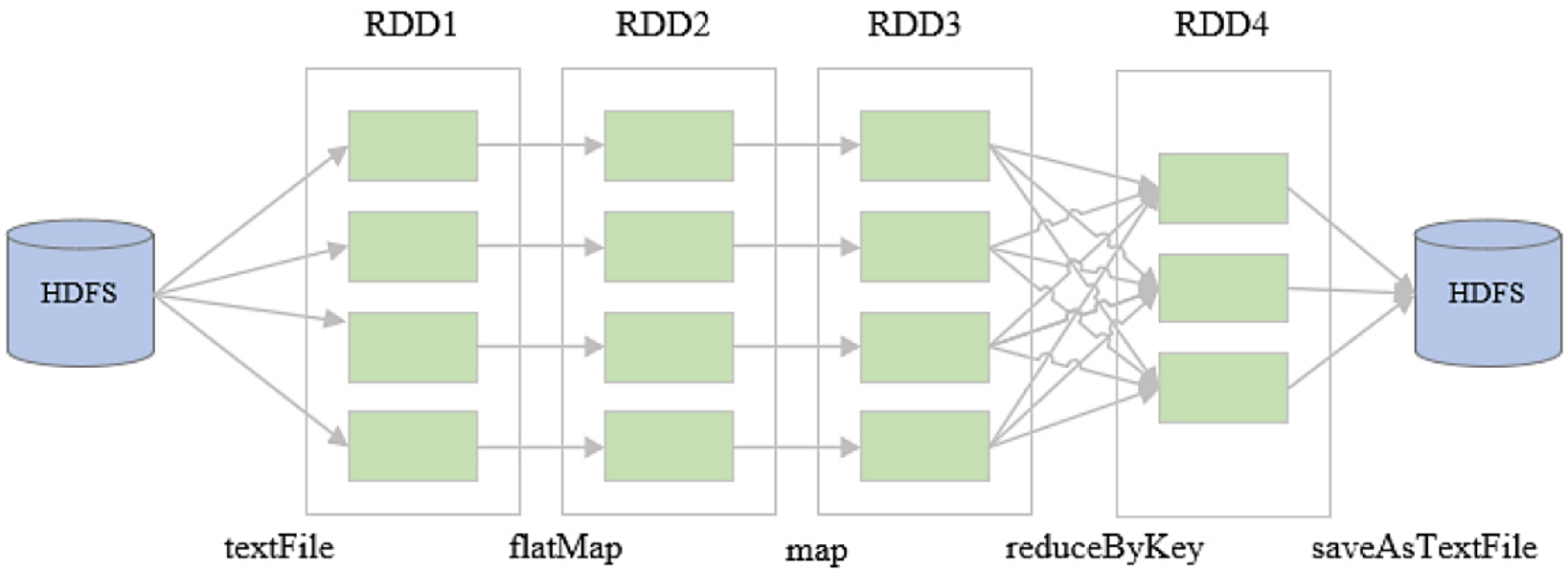
- 根据RDD之间的依赖关系的不同将Job划分成不同的Stage（调度阶段）
- 遇到一个**宽依赖**，则划分一个Stage



➢ Spark执行流程 —— WordCount 示例:

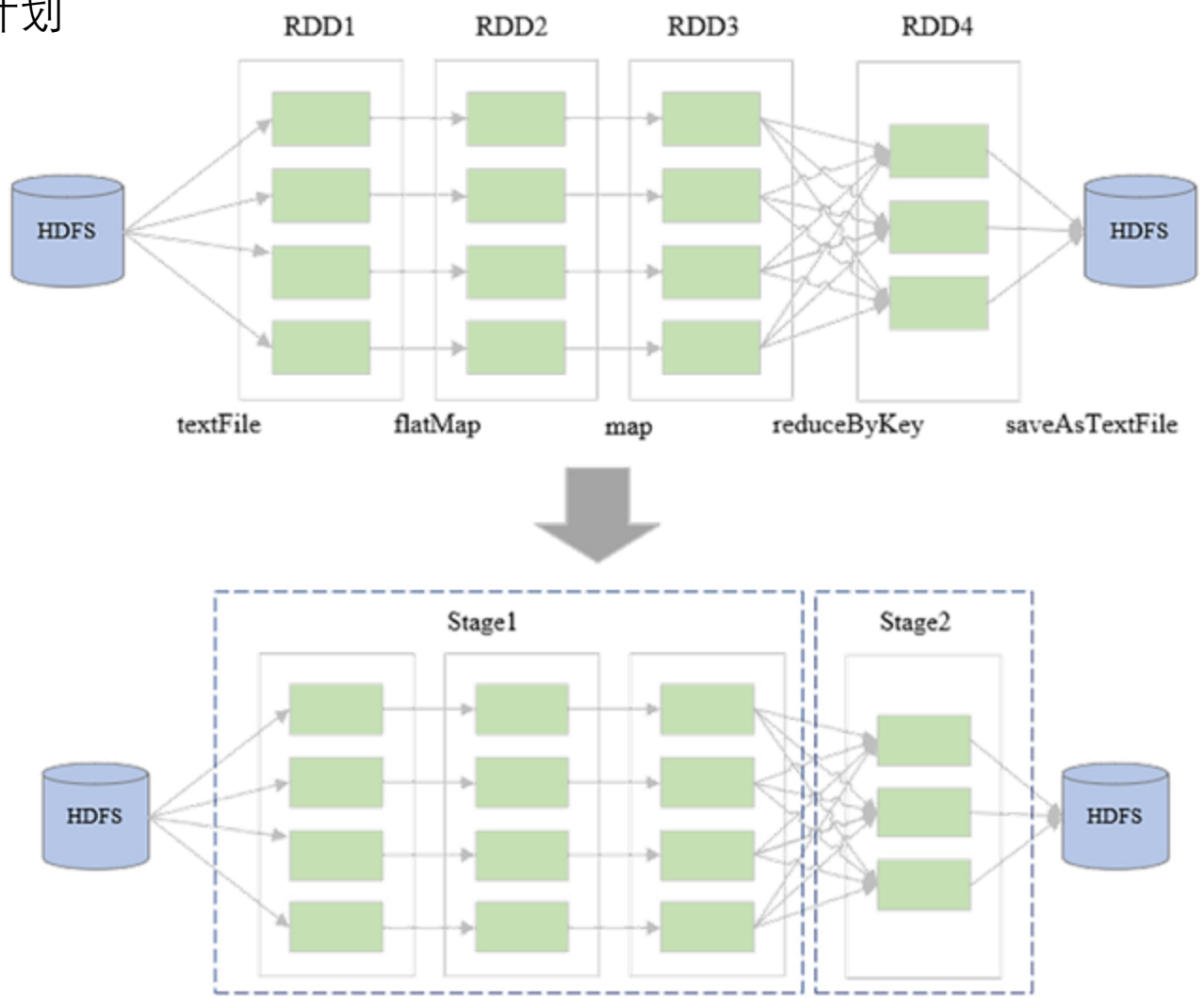
① 生成逻辑执行计划

```
val rdd1 = sc.textFile("hdfs://node01:9000/data/in")
val rdd2 = rdd1.flatMap(_.split("\t"))
val rdd3 = rdd2.map((_,1))
val rdd4 = rdd3.reduceByKey(_+_)
rdd4.saveAsTextFile("hdfs://node01:9000/data/out")
```



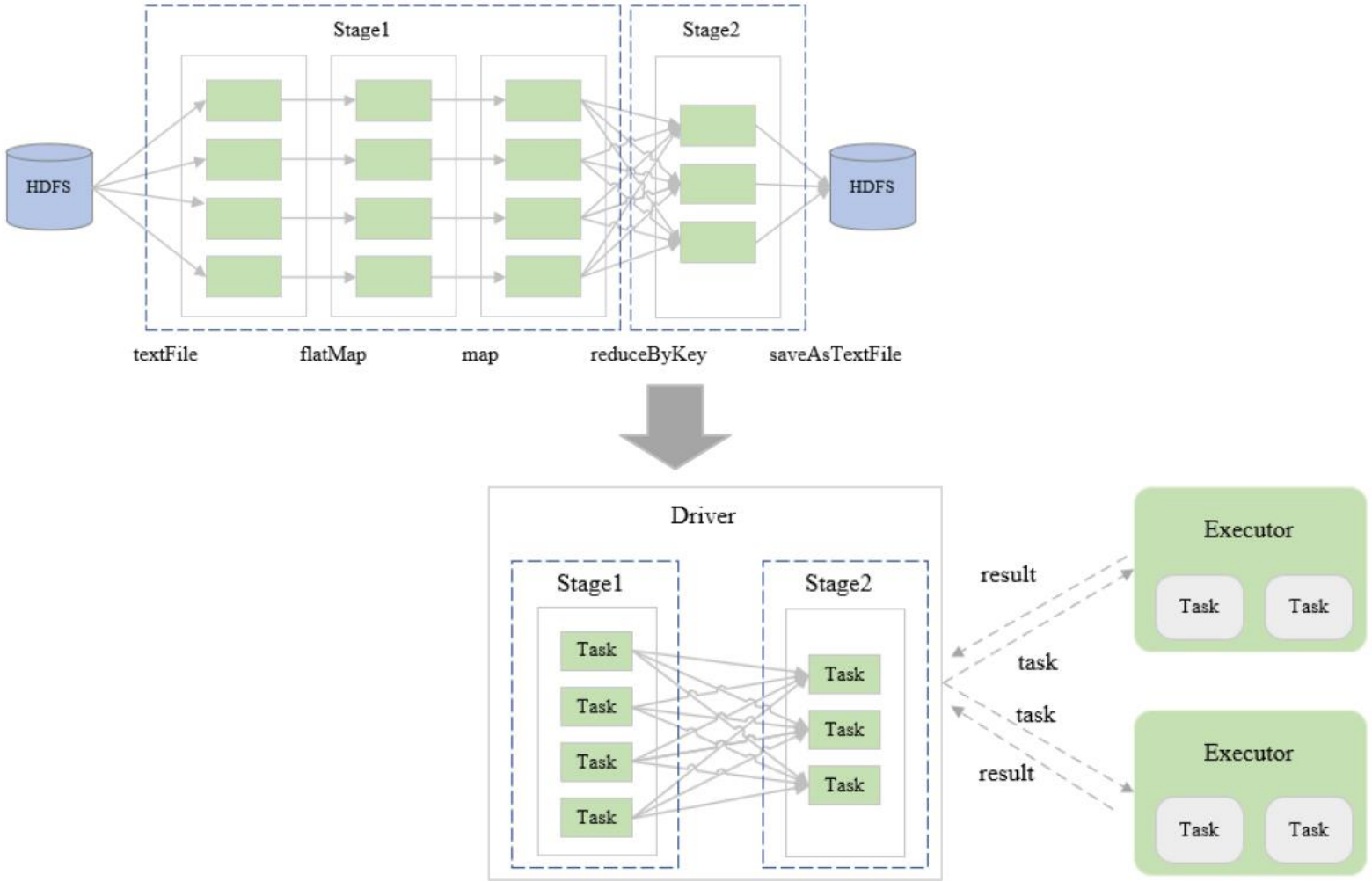
- Spark执行流程 —— WordCount 示例:

② 生成物理执行计划

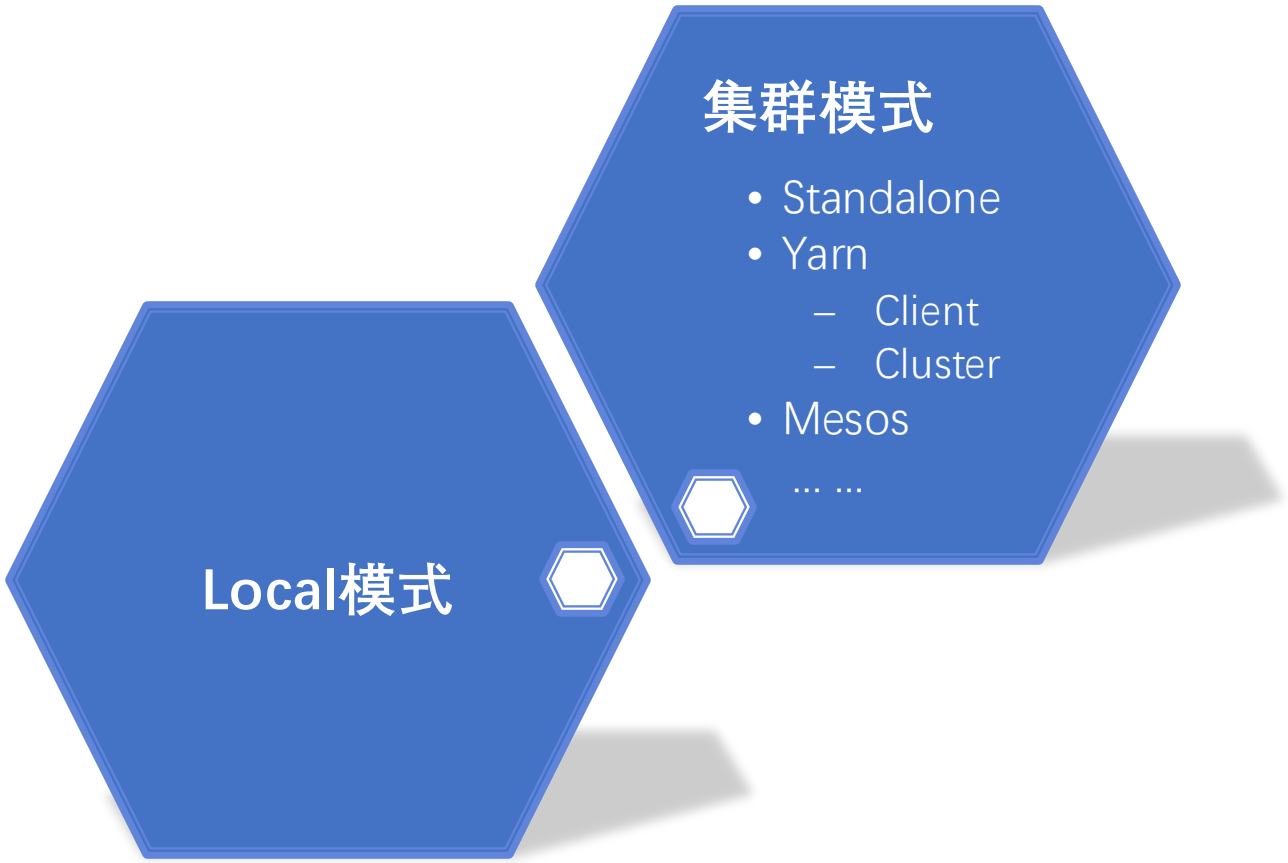


- Spark执行流程 —— WordCount 示例:

③ 任务提交&调度

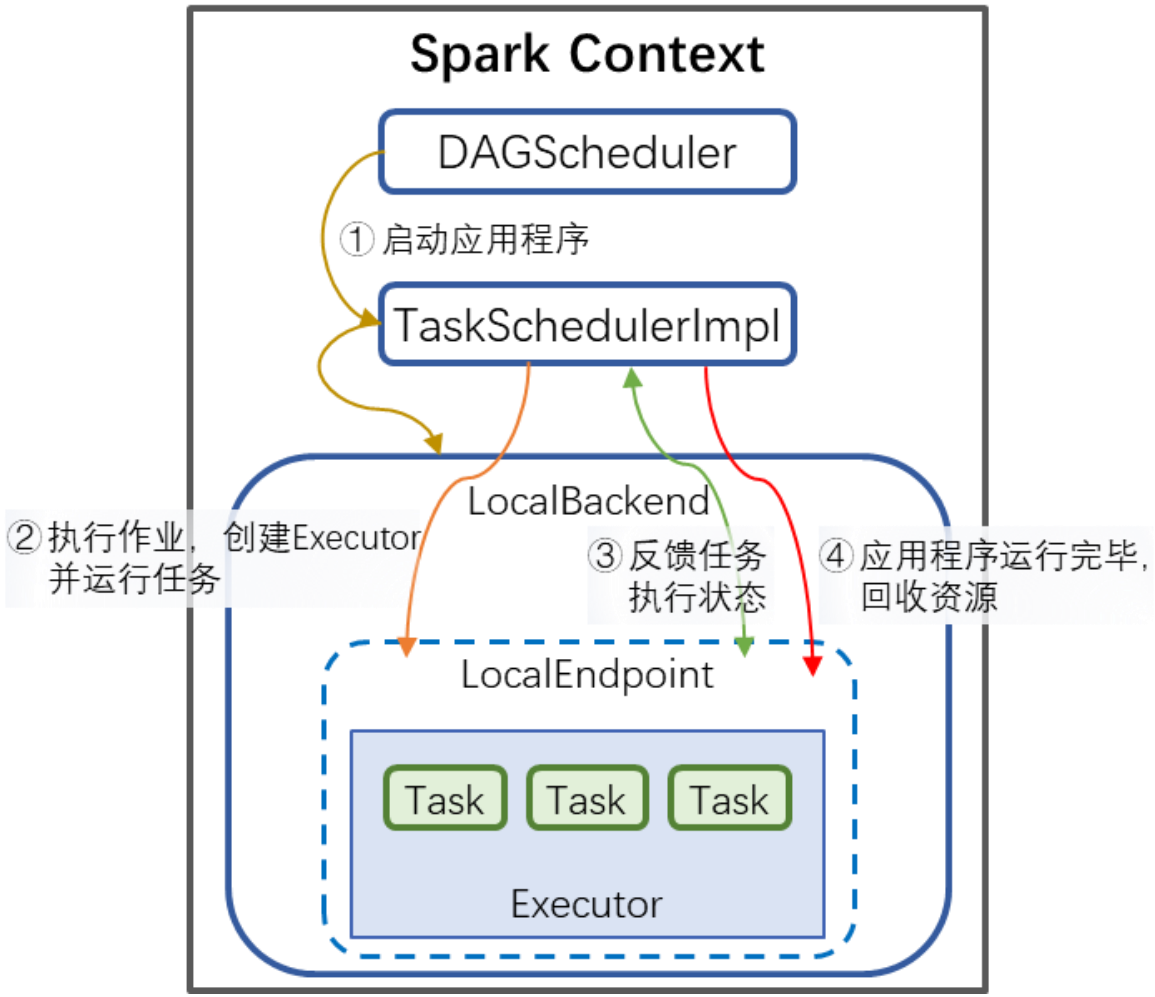


- Spark支持多种部署模式



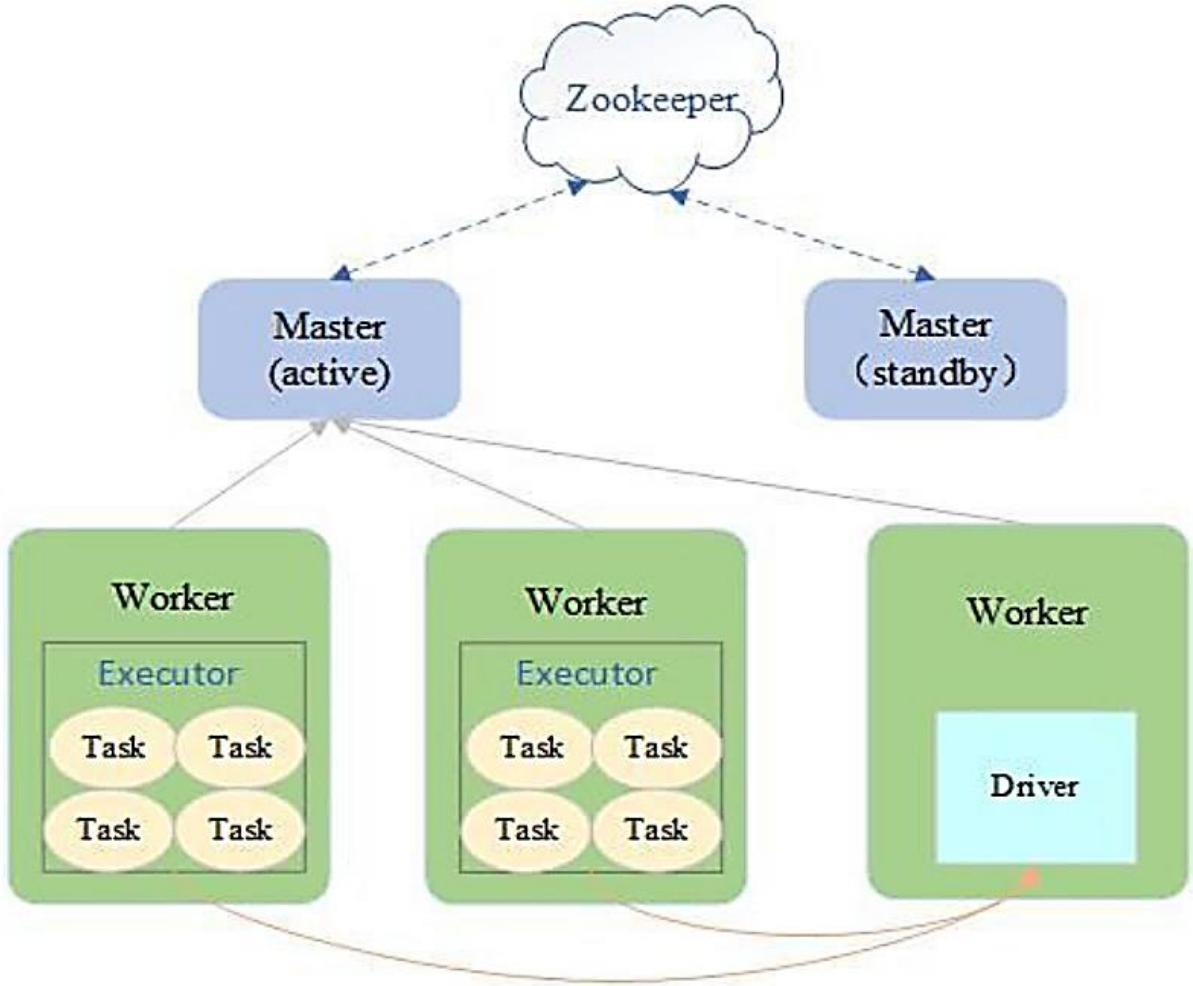
➢ Local模式

- 单机运行，通常用于测试
- Spark程序以多线程方式直接运行在本地，所有进程都运行在一台机器的JVM中



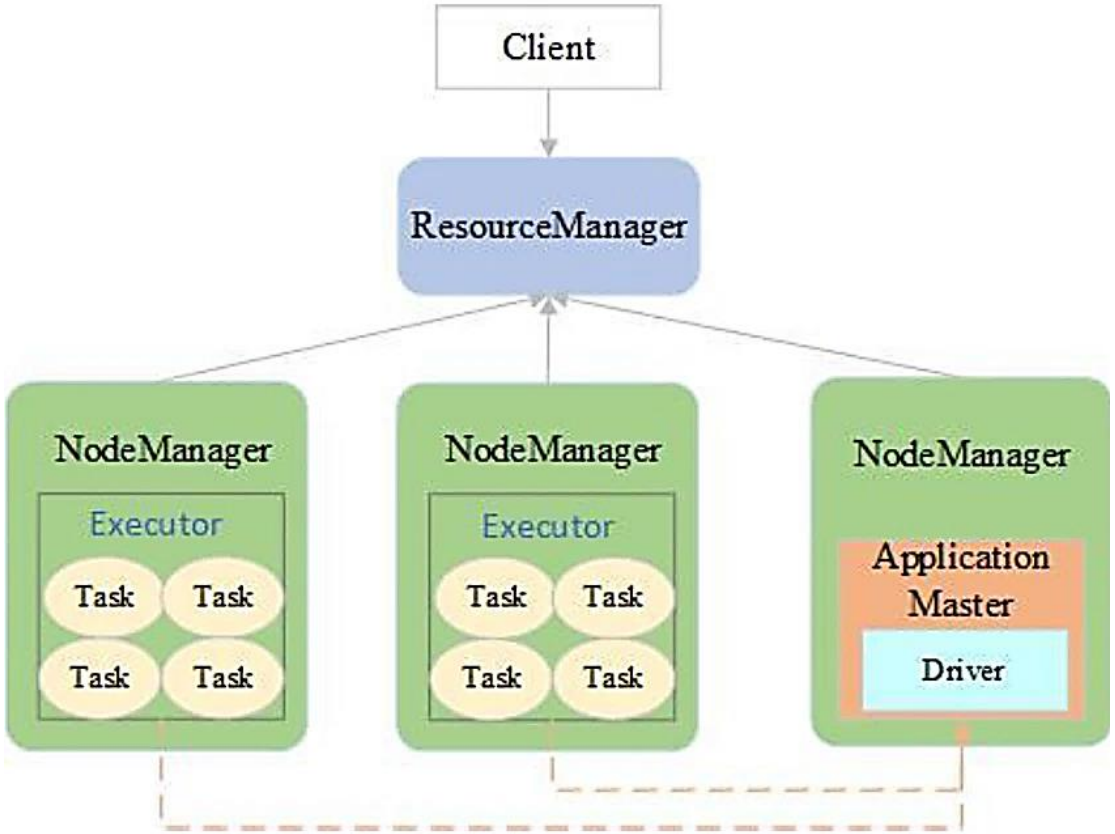
➢ Standalone模式

- Spark集群独立运行，不依赖于第三方资源管理系统，如YARN、Mesos
- 采用Master/Slave架构
  - Master统一管理集群资源
  - Worker负责本地计算
  - Driver一对一管理作业
- ZooKeeper负责Master HA，避免单点故障
- 适用于集群规模和数据量都不大的情况

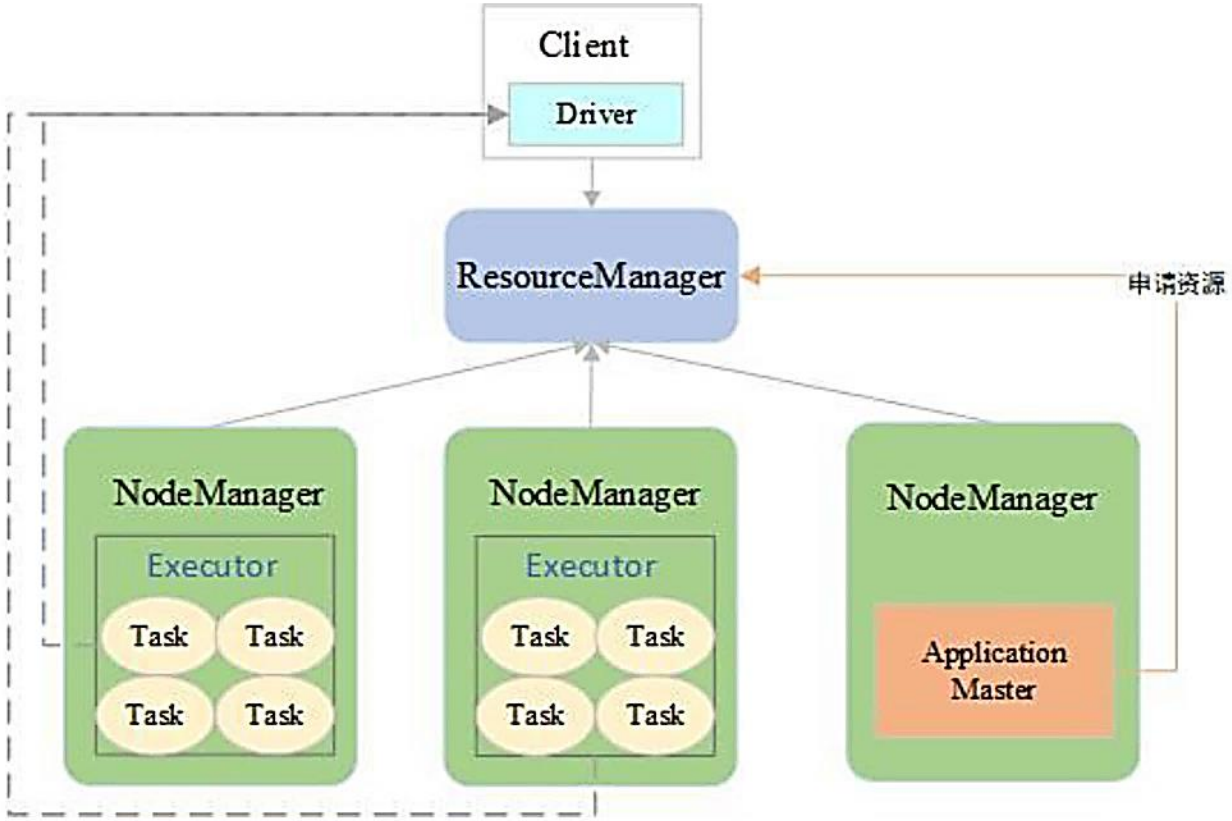


➢ YARN模式

- YARN-Client模式：适用于交互和调试
- YARN-Cluster模式：适用于生产环境



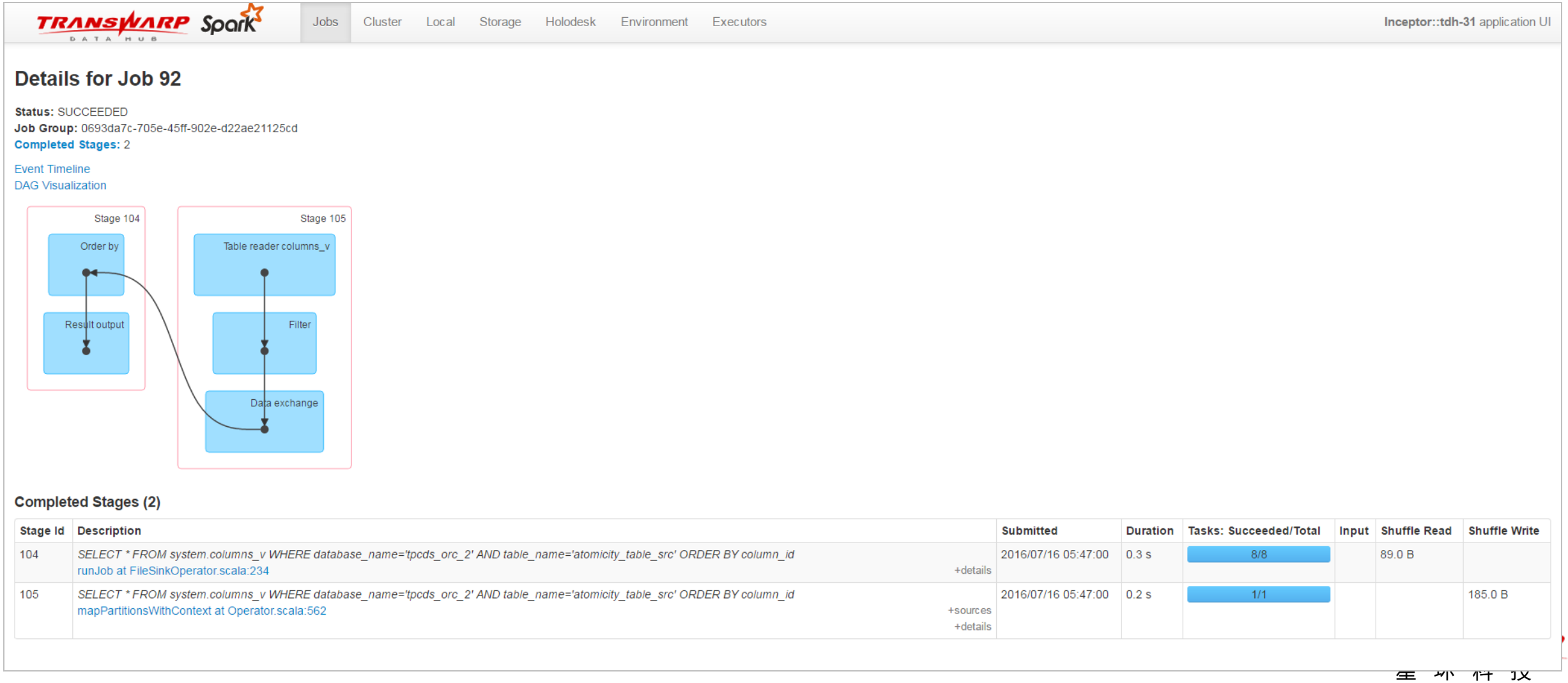
YARN-Cluster模式



YARN-Client模式

2.4 作业监控

- Web监控：http://{master_server_ip}:4040



3
chapter

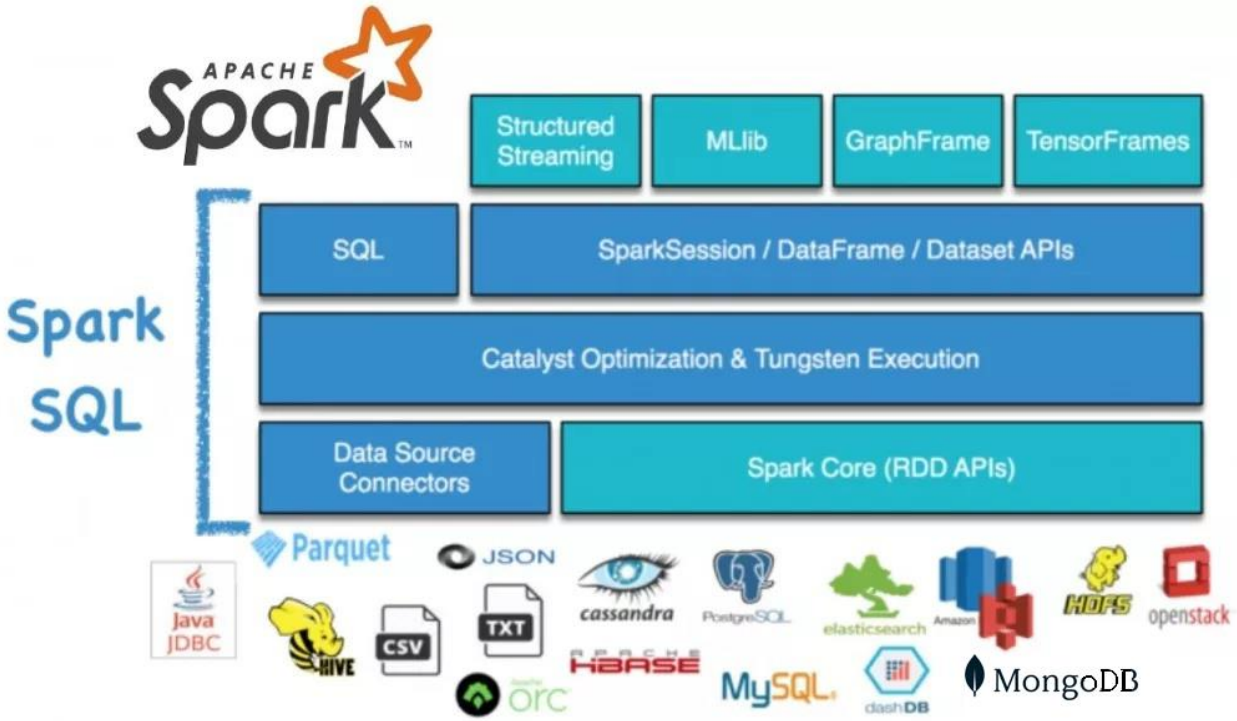
Spark SQL

- ✓ Spark SQL 概述
- ✓ DataFrame
- ✓ DataSet
- ✓ RDD、DataFrame、DataSet

➢ SparkSQL概念:

- Spark SQL 是 Spark 中用来处理**结构化数据**的一个模块，它提供了一个编程抽象（DataFrame），并且可以作为分布式 SQL 的查询引擎。

- Spark SQL 可以将数据的计算任务通过 SQL 的形式转换成 RDD再提交到集群执行计算，类似于 Hive 通过 SQL 的形式将数据的计算任务转换成 MapReduce，大大简化了编写 Spark 数据计算操作程序的复杂性，且执行效率比 MapReduce 这种计算模型高。



Spark架构设计

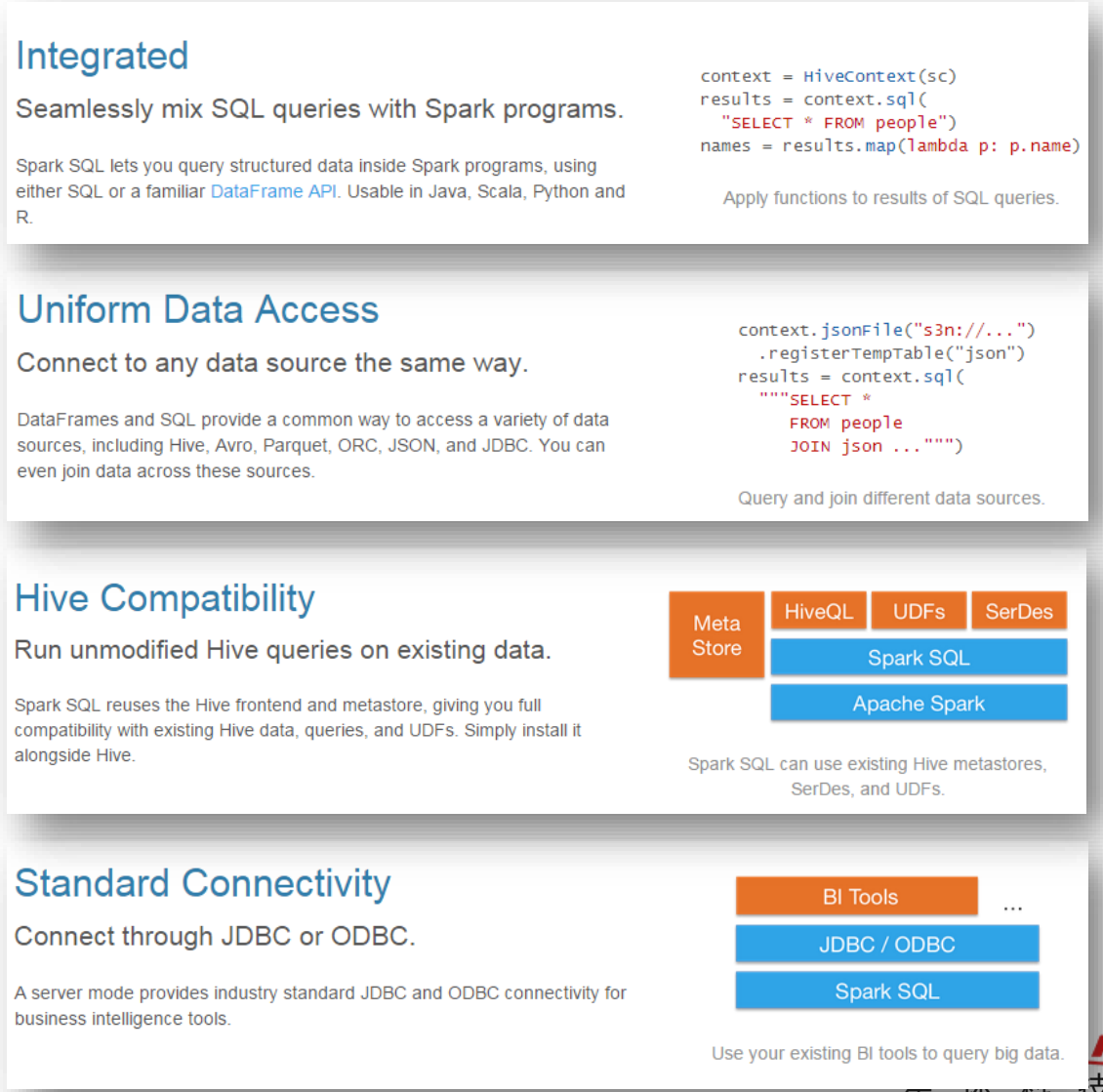
➢ Spark SQL特点

① 和 Spark Core 的无缝集成，可以在写整个 RDD 应用程序时，配置 Spark SQL 来完成逻辑实现。

② 统一的数据访问方式，Spark SQL 提供标准化的 SQL 查询。

③ Hive 的继承，Spark SQL 通过内嵌的 Hive 或者连接外部已经部署好的 Hive 案例，实现了对 Hive 语法的继承和操作。

④ 标准化的连接方式，Spark SQL 可以通过启动 Thrift Server 来支持 JDBC、ODBC 的访问，将自己作为一个 BI Server 使用。

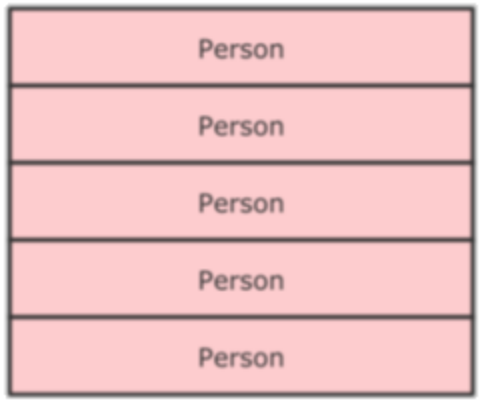


➢ DataFrame概念:

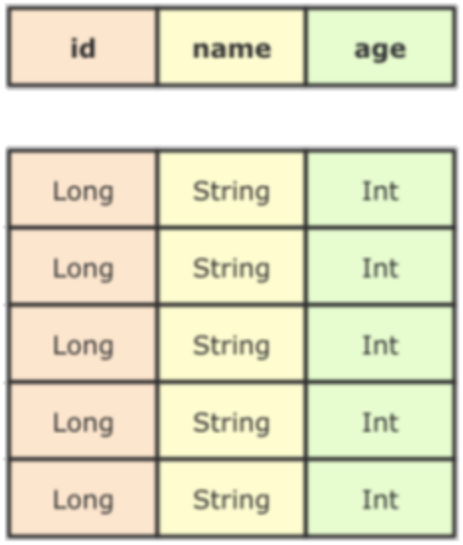
- DataFrame 是一种以 RDD 为基础的的分布式数据集，类似于传统数据库的二维表格。

  可以理解为是SparkSQL中一个表示关系型数据库中表的函数式抽象，主要是为了让Spark处理大规模结构化数据的时候更加容易。

- 一般DataFrame可以处理结构化的数据，或者是半结构化的数据，因为这两类数据中都可以获取到Schema信息。也就是说DataFrame中有Schema信息，可以像操作表一样DataFrame。



RDD 中存储 Person 对象



DataFrame

➢ 构建DataFrame：

- 示例：

```
//读取JSON文件，构建DataFrame
val df = spark.read.json("file:///xxxxx/people.json")

//对DataFrame创建表
df.createGlobalTempView("people")

//使用SQL语句进行全表查询
spark.sql("SELECT age, name FROM global_temp.people").show()

返回输出：
scala> sqlDF.show
+----+-------+
| age |  name  |
+----+-------+
| 23  |  JUDY  |
| 39  |  TOM   |
+----+------+
```

➢ DataFrame常用操作：

① DSL风格语法：

SparkSQL中的DataFrame自身提供了一套自己的Api，可以去使用这套api来做相应的处理。

```
//加载数据
val rdd1=sc.textFile("/person.txt").map(x=>x.split(" "))
//定义一个样例类
case class Person(id:String,name:String,age:Int)
//把rdd与样例类进行关联
val personRDD=rdd1.map(x=>Person(x(0),x(1),x(2).toInt))
//把rdd转换成DataFrame
val personDF=personRDD.toDF

//查询指定的字段
personDF.select("name").show
personDF.select($"name").show
personDF.select(col("name").show

personDF.select($"name",$"age",$"age"+1).show          //实现age+1
personDF.filter($"age" > 30).show                      //实现age大于30过滤
personDF.groupBy("age").count.show                     //按照age分组统计次数
personDF.groupBy("age").count().sort($"count".desc)show  //按照age分组统计次数降序
```

➢ DataFrame常用操作：

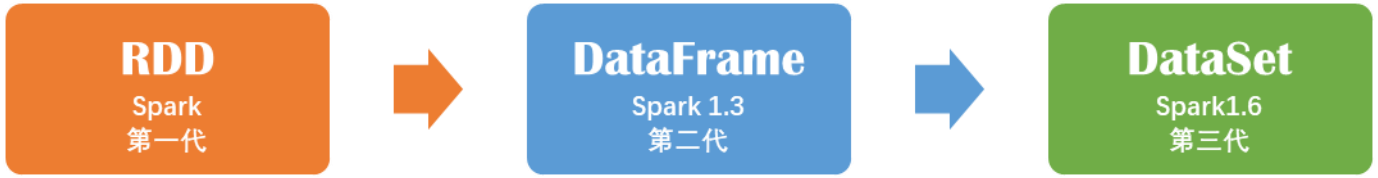
② SQL风格语法（重点）：

将DataFrame注册成一张表，然后通过sparkSession.sql(sql语句)操作。

```
//DataFrame注册成表
personDF.createTempView("person")

//使用SparkSession调用sql方法统计查询
spark.sql("select * from person").show
spark.sql("select name from person").show
spark.sql("select name,age from person").show
spark.sql("select * from person where age >30").show
spark.sql("select count(*) from person where age >30").show
spark.sql("select age,count(*) from person group by age").show
spark.sql("select age,count(*) as count from person group by age").show
spark.sql("select * from person order by age desc").show
```

➢ DataSet概念:

- 从 Spark 1.6 开始出现 DataSet，作为 DataFrame API 的一个扩展，是一个**强类型**的特定领域的对象，这种对象可以函数式或者关系操作并行地转换，结合了 RDD 和 DataFrame 的优点，至 Spark 2.0 中将 DataFrame 与 DataSet 合并。
- 产生的原因（DataFrame限制）：
  ① 编译时类型不安全：DataFrame API 不支持编译时安全性，这限制了在结构不知道时操纵数据，使得在编译期间有效，但执行代码时出现运行时异常
  ② 无法对域对象（丢失域对象）进行操作：将域对象转换为 DataFrame 后，无法从中重新生成它，就是说无法重新生成原始 RDD。
- DataSet API可以理解为是DataFrame的扩展，它提供了一种类型安全的、面向对象的编程接口，它是一个强类型、不可变的对象集合，映射到关系模式。



➢ 构建DataSet：

- 示例：

```
// 1、通过sparkSession调用createDataset方法
val ds=spark.createDataset(1 to 10) //scala集合
val ds=spark.createDataset(sc.textFile("/person.txt"))  //rdd

// 2、使用scala集合和rdd调用toDS方法
sc.textFile("/person.txt").toDS
List(1,2,3,4,5).toDS

// 3、把一个DataFrame转换成DataSet
val dataSet=dataFrame.as[强类型]

// 4、通过一个DataSet转换生成一个新的DataSet
List(1,2,3,4,5).toDS.map(x=>x*10)
```

- RDD vs DataFrame vs DataSet：

示例：RDD中2行人员数据：

001	Judy
002	Tom

DataFrame：

ID:String	Name:String
001	Judy
002	Tom

DataSet：

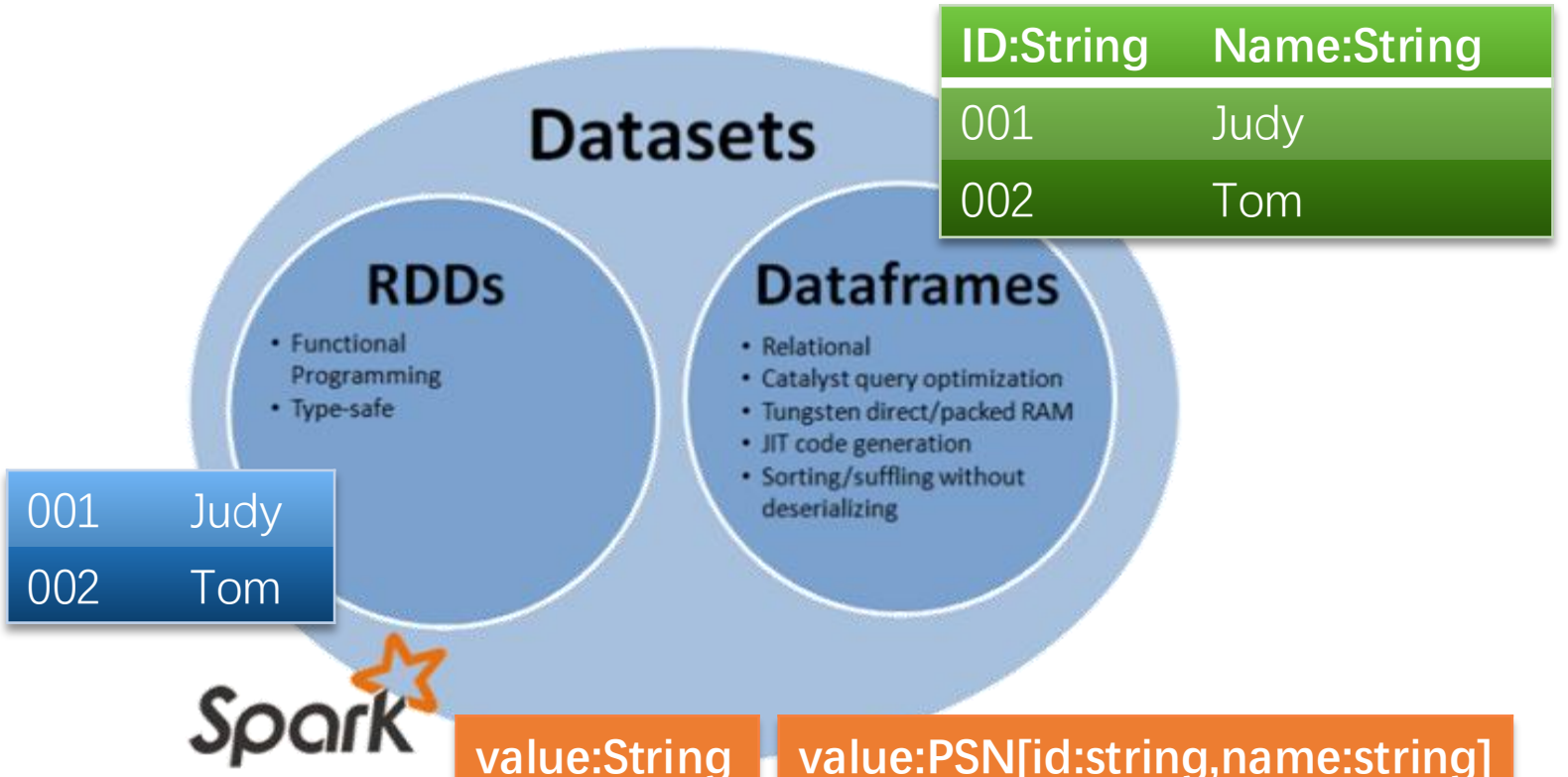
（String或者Object）

value:String
001, Judy
002, Tom

value:PSN[id:string,name:string]
PSN(id="001", name="Judy")
PSN(id="002", name="Tom")

➢ RDD vs DataFrame vs DataSet：

- 三者关系：



ID:String	Name:String
001	Judy
002	Tom

001	Judy
002	Tom

value:String	value:PSN[id:string,name:string]
001, Judy	PSN(id="001", name="Judy")
002, Tom	PSN(id="002", name="Tom")

➢ RDD vs DataFrame vs DataSet:

- 共性:

① RDD、DataFrame、DataSet 都是 Spark 平台下的弹性分布式数据集，为处理大型数据提供便利。

② 三者都有惰性机制，在进行创建、转换等操作时不会立即执行，只有触发行动算子时才会执行。

③ 在对 DataFrame 和 DataSet 进行操作时都需要导入隐式转换的包 “import spark.implicits”。

④ 三者都会根据 Spark 的内存情况自动缓存运算，所以即使数据量很大，也不用担心内存溢出。

⑤ DataFrame 和 DataSet 均可使用模式匹配获取各个字段的值和类型。

⑥ 三者都有 partition 的概念、以及有许多共同的函数，如 filter、排序等。

- 区别:

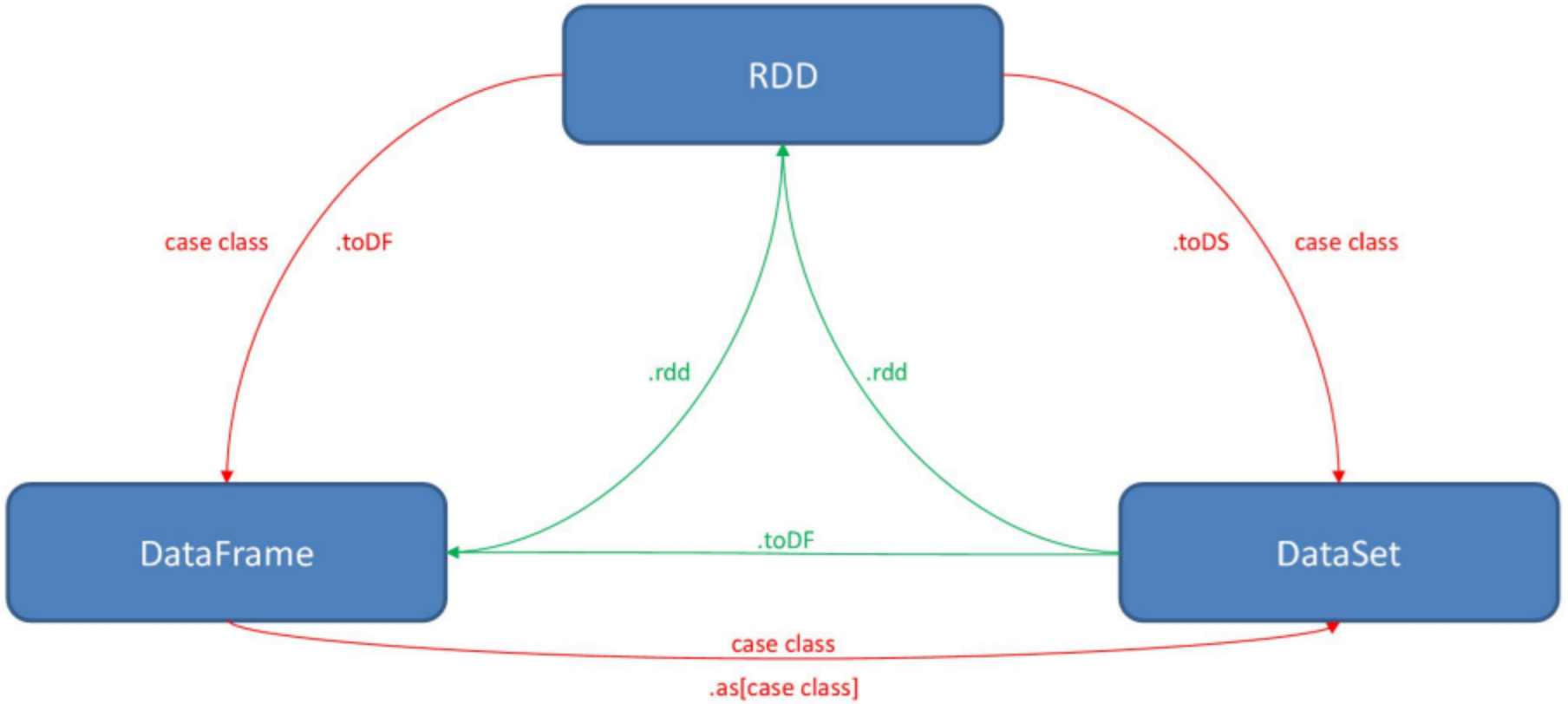
① RDD一般和spark mlib同时使用，且不支持sparksql操作。

② DataFrame与Dataset一般不与spark mlib同时使用，但均支持sparksql的操作，比如select，groupby之类，还能注册临时表/视窗，进行sql语句操作。

③ DataFrame与RDD和Dataset不同，DataFrame每一行的类型固定为Row（即表示为**Dataset[Row]**），每一列的值没法直接访问，只有通过解析才能获取各个字段的值（其只关心数据的结构，RDD只关心数据的值）。

④ Dataset和DataFrame拥有完全相同的成员函数，区别只是每一行的数据类型不同。Dataset在需要访问列中的某个字段时是非常方便的，然而，如果要写一些适配性很强的函数时，如果使用Dataset，行的类型又不确定，可能是各种case class，无法实现适配，这时候用DataFrame即Dataset[Row]就能比较好的解决问题。

- RDD vs DataFrame vs DataSet:
  - 三者相互转换:



➢ RDD转换为DataFrame：

- 需要导入隐式转换并创建一个RDD。

```
val peopleRDD = sc.textFile("examples/src/main/resources/people.txt")

//通过手动确定转换
peopleRDD.map{x=>val para = x.split(",");(para(0),para(1).trim.toInt)}.toDF("name","age")

//通过反射确定（需要用到样例类）
case class People(name:String, age:Int)
peopleRDD.map{ x => val para = x.split(",");People(para(0),para(1).trim.toInt)}.toDF
```

➢ DataFrame转为RDD：

- 调用rdd方法。

```
val df = spark.read.json("/opt/module/spark/examples/src/main/resources/people.json")

//将DataFrame转换为RDD
val dfToRDD = df.rdd
```

➢ RDD转换为DataSet：

- SparkSQL能够自动将包含有case类的RDD转换成DataFrame，case类定义了table的结构，case类属性通过反射变成了表的列名。

```
val peopleRDD = sc.textFile("examples/src/main/resources/people.txt")

//创建一个样例类
case class Person(name: String, age: Long)

//将RDD转化为DataSet
peopleRDD.map(line => {val para = line.split(",");Person(para(0),para(1).trim.toInt)}).toDS()
```

➢ DataSet转为RDD：

- 调用rdd方法。

```
val DS = Seq(Person("Andy", 32)).toDS()

//将DataSet转换为RDD
DS.rdd
```

➢ RDD、DataFrame、DataSet相互转换：

- 案例：

```
import org.apache.spark.rdd.RDD
import org.apache.spark.{SparkConf, SparkContext}
import org.apache.spark.sql.{DataFrame, Dataset, Row, SparkSession}

//定义一个样例类
case class User(id: String, name: String, age: Int)

//todo: RDD、DataFrame、DataSet互相转换案例
object SparkConversion {
  def main(args: Array[String]): Unit = {

    val sparkConf: SparkConf = new SparkConf().setMaster("local[2]").setAppName("SparkConversion")
    val spark: SparkSession = SparkSession.builder().config(sparkConf).getOrCreate()
    val sc: SparkContext = spark.sparkContext
    sc.setLogLevel("WARN")

    //todo: 隐式转换
    import spark.implicits._

    //todo: 加载数据
    val rdd = sc.textFile("D:\\person.txt").map(x => x.split(" "))
```

➢ RDD、DataFrame、DataSet相互转换：

- 案例：

```
//todo: 把rdd与样例类进行关联
val userRDD = rdd.map(x => User(x(0), x(1), x(2).toInt))

//1. RDD转DF
val df1: DataFrame = userRDD.toDF
df1.show

//2. RDD转DS
val ds1: Dataset[User] = userRDD.toDS
ds1.show

//3. DF转RDD
val rdd1: RDD[Row] = df1.rdd
println(rdd1.collect.toList)

//4. DS转RDD
val rdd2: RDD[User] = ds1.rdd
println(rdd2.collect.toList)
```

```
  //5. DS转DF
  val df2: DataFrame = ds1.toDF
  df2.show()

  //6. DF转DS
  val ds2: Dataset[User] = df2.as[User]
  ds2.show()

  spark.stop()
 }
}
```

4
chapter

Spark Streaming

- ✓ Spark Streaming 概述
- ✓ Spark Streaming 开发

➢ SparkStreaming概念

- Spark Streaming是构建在Spark上的实时计算框架，它扩展了Spark处理大规模流式数据的能力。可结合批处理和交互查询，适合一些需要对历史数据和实时数据进行结合分析的应用场景。
- Spark Streaming可整合多种输入数据源，如Kafka、Flume、HDFS，甚至是普通的TCP套接字。经处理后的数据可存储至文件系统、数据库，或显示在仪表盘里。

➢ SparkStreaming特点

- ✓ 易用
- ✓ 容错
- ✓ 易整合到Spark体系

Ease of Use

Build applications through high-level operators.

Spark Streaming brings Spark's language-integrated API to stream processing, letting you write streaming jobs the same way you write batch jobs. It supports Java, Scala and Python.

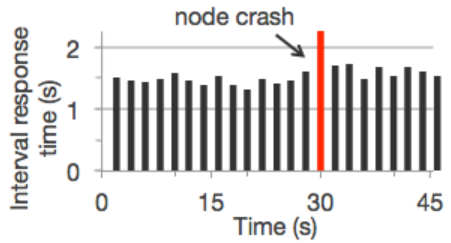
```
TwitterUtils.createStream(...)
    .filter(_.getText.contains("Spark"))
    .countByWindow(Seconds(5))
```

Counting tweets on a sliding window

Fault Tolerance

Stateful exactly-once semantics out of the box.

Spark Streaming recovers both lost work and operator state (e.g. sliding windows) out of the box, without any extra code on your part.



Spark Integration

Combine streaming with batch and interactive queries.

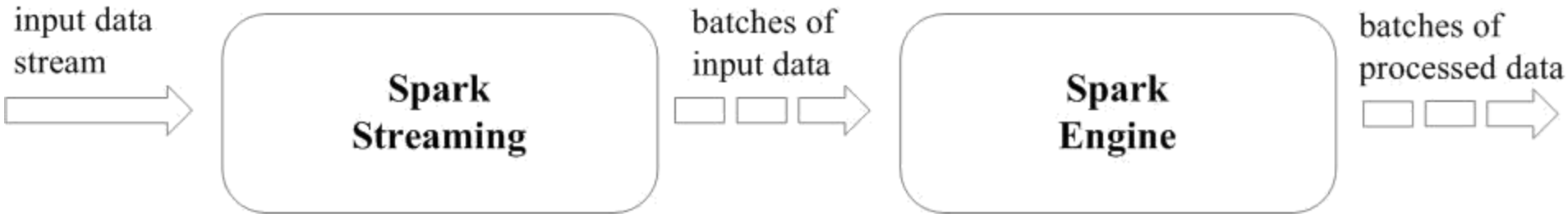
By running on Spark, Spark Streaming lets you reuse the same code for batch processing, join streams against historical data, or run ad-hoc queries on stream state. Build powerful interactive applications, not just analytics.

```
stream.join(historicCounts).filter {
  case (word, (curCount, oldCount)) =>
    curCount > oldCount
}
```

Find words with higher frequency than historic data

➢ SparkStreaming运行流程

- 实时输入数据流以时间片（秒级）为单位进行拆分，然后经Spark引擎以类似批处理的方式处理每个时间片数据。



➢ Dstream

- Spark Streaming最主要的抽象是DStream（Discretized Stream，离散化数据流），表示连续不断的数据流。在内部实现上，Spark Streaming的输入数据按照时间片（如1秒）分成一段一段的DStream，每一段数据转换为Spark中的RDD，并且对DStream的操作都最终转变为对相应的RDD的操作。

➢ WordCount：

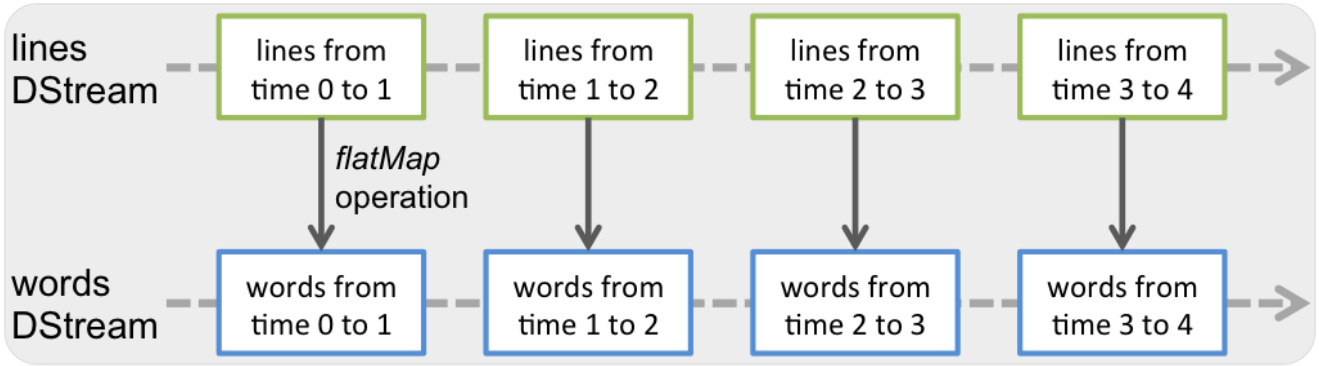
- 示例：

```
object StreamWordCount {
  def main(args: Array[String]): Unit = {

    val sparkConf = new SparkConf().setMaster("local[*]").setAppName("StreamWordCount")   //1.初始化Spark配置信息

    val ssc = new StreamingContext(sparkConf, Seconds(5))                                  //2.初始化SparkStreamingContext

    val lineStreams = ssc.socketTextStream("hadoop102", 9999) //3.通过监控端口创建DStream，读进来的数据为一行行

    val wordStreams = lineStreams.flatMap(_.split(" "))                                    //将每一行数据做切分，形成一个个单词

    val wordAndOneStreams = wordStreams.map((_, 1))                                        //将单词映射成元组（word,1）

    val wordAndCountStreams = wordAndOneStreams.reduceByKey(_+_)                           //将相同的单词次数做统计

    wordAndCountStreams.print()                                                            //打印

    ssc.start()                                                                            //启动SparkStreamingContext
    ssc.awaitTermination()
  }
}
```

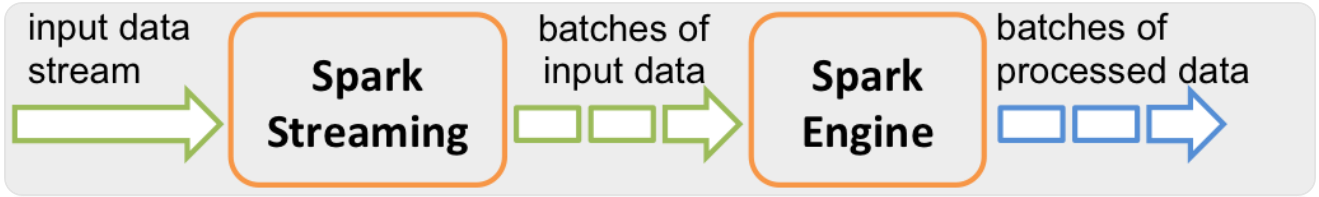
➢ WordCount:

✓ 在内部实现上，DStream是一系列连续的RDD来表示。每个RDD含有一段时间间隔内的数据:

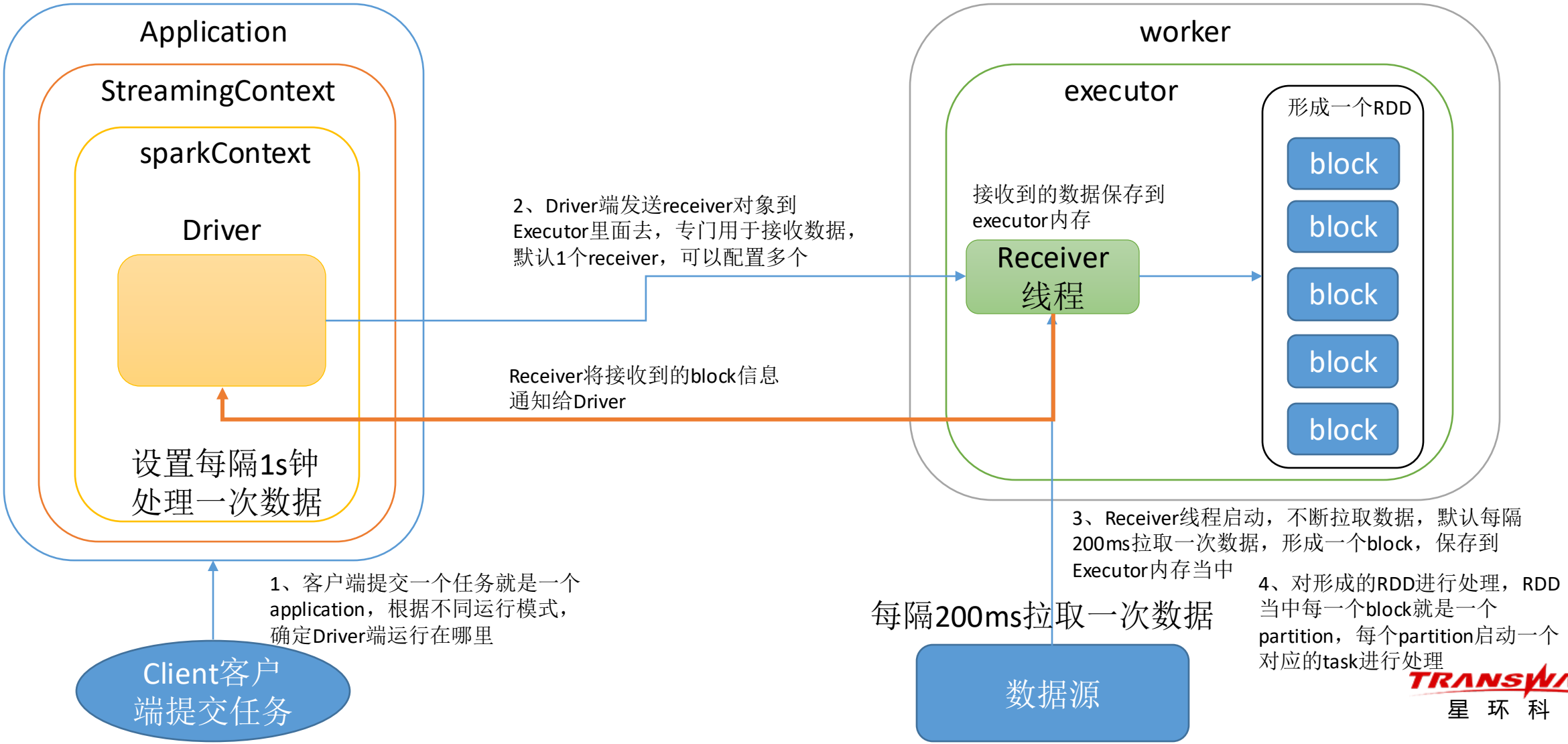
✓ 对数据的操作也是按照RDD为单位来进行的:



✓ 计算过程由Spark engine来完成:



➢ SparkStreaming架构流程：



➢ DStream创建

- Socket数据源

  实时接收socket数据

```
object SocketWordCount {
  def main(args: Array[String]): Unit = {
    // todo: 1、创建SparkConf对象
    val sparkConf: SparkConf = new SparkConf().setAppName("TcpWordCount").setMaster("local[2]")

    // todo: 2、创建StreamingContext对象
    val ssc = new StreamingContext(sparkConf,Seconds(2))

    //todo: 3、接受socket数据
    val socketTextStream: ReceiverInputDStream[String] = ssc.socketTextStream("node01",9999)

    //todo: 4、对数据进行处理
    val result: DStream[(String, Int)] = socketTextStream.flatMap(_.split(" ")).map((_,1)).reduceByKey(_+_)

    //todo: 5、打印结果
    result.print()

    //todo: 6、开启流式计算
    ssc.start()
    ssc.awaitTermination()  } }
```

➢ DStream创建

- Kafka数据源

实时接收Kafka数据

```
//1、创建StreamingContext对象
val sparkConf= new SparkConf().setAppName("KafkaReceiver").setMaster("local[2]")
                              //开启WAL机制
                             .set("spark.streaming.receiver.writeAheadLog.enable", "true")
val ssc = new StreamingContext(sparkConf,Seconds(2))
//需要设置checkpoint,将接受到的数据持久化写入到hdfs上
ssc.checkpoint("hdfs://node01:8020/wal")

//2、接受kafka数据
val zkQuorum="node01:2181,node02:2181,node03:2181"
val groupid="KafkaReceiver"
val topics=Map("test" ->1)

//(String, String) 元组的第一位是消息的key，第二位表示消息的value
val receiverDstream: ReceiverInputDStream[(String, String)] = KafkaUtils.createStream(ssc,zkQuorum,groupid,topics)

//3、获取kafka的topic数据
val data: DStream[String] = receiverDstream.map(_._2)

//4、单词计数
val result: DStream[(String, Int)] = data.flatMap(_.split(" ")).map((_,1)).reduceByKey(_+_)
```

➢ 基础算子

- Transformations

实现将一个DStream转换生成一个新的DStream，延迟加载不会触发任务的执行

Transformation	含义
map(func)	对DStream中的各个元素进行func函数操作，然后返回一个新的DStream
flatMap(func)	与map方法类似，只不过各个输入项可以被输出为零个或多个输出项
filter(func)	过滤出所有函数func返回值为true的DStream元素并返回一个新的DStream
repartition(numPartitions)	增加或减少DStream中的分区数，从而改变DStream的并行度
union(otherStream)	将源DStream和输入参数为otherDStream的元素合并，并返回一个新的DStream.
count()	通过对DStream中的各个RDD中的元素进行计数，然后返回只有一个元素的RDD构成的DStream
reduce(func)	对源DStream中的各个RDD中的元素利用func进行聚合操作，然后返回只有一个元素的RDD构成的新的DStream.
countByValue()	对于元素类型为K的DStream，返回一个元素为（K,Long）键值对形式的新的DStream，Long对应的值为源DStream中各个RDD的key出现的次数
reduceByKey(func, [numTasks])	利用func函数对源DStream中的key进行聚合操作，然后返回新的（K，V）对构成的DStream
join(otherStream, [numTasks])	输入为（K,V)、（K,W）类型的DStream，返回一个新的（K，（V，W））类型的DStream
cogroup(otherStream, [numTasks])	输入为（K,V)、（K,W）类型的DStream，返回一个新的 (K, Seq[V], Seq[W]) 元组类型的DStream
reduceByKeyAndWindow	窗口函数操作，实现按照window窗口大小来进行计算

➢ 基础算子

- Output Operations

  输出算子操作，触发任务的真正运行

Output Operation	含义
print()	打印到控制台
saveAsTextFiles(prefix, [suffix])	保存流的内容为文本文件，文件名为"prefix-TIME_IN_MS[.suffix]"
saveAsObjectFiles(prefix, [suffix])	保存流的内容为SequenceFile，文件名为 "prefix-TIME_IN_MS[.suffix]"
saveAsHadoopFiles(prefix, [suffix])	保存流的内容为hadoop文件，文件名为 "prefix-TIME_IN_MS[.suffix]"

➢ 特殊算子

① window

窗口计算。属于有状态的转换。

② updateStateByKey

根据key的之前状态值和key的新值，对key进行更新，返回一个新状态的DStream，用于记录历史记录。属于有状态的转换。

③ mapWithState

类比updateStateByKey，均是用来统计全局key的状态的变化；但mapWithState可以只返回变化后的key的值，且不需要做checkpoint，不会因此占用大量的数据量、影响性能。新版本中推荐使用mapWithState用于延迟低、实时性要求更高的场景。

④ transform

通过RDD-to-RDD函数作用于DStream中的各个RDD，可以是任意的RDD操作，从而返回一个新的RDD。

⑤ foreachRDD

对Dstream里面的每个RDD执行func。

➢ 算子示例：

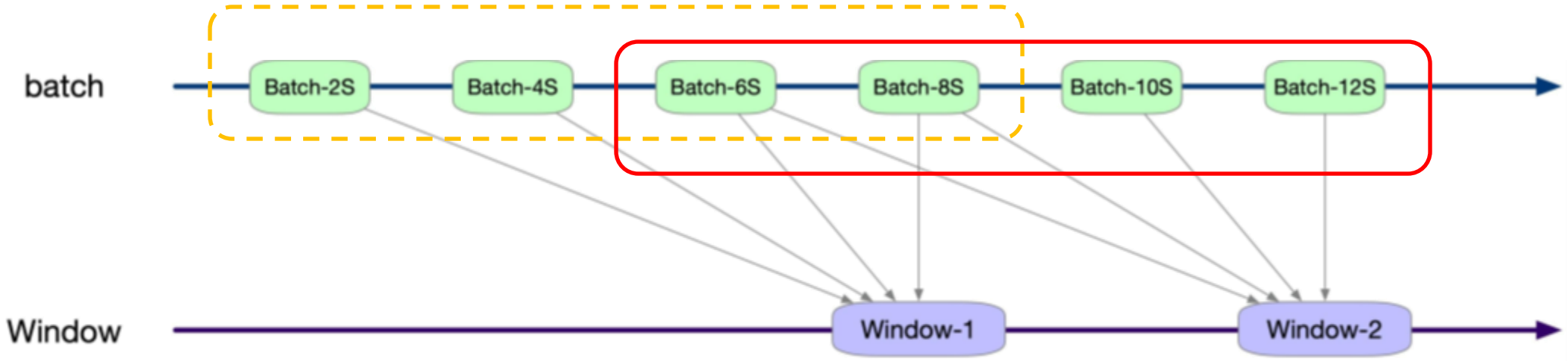
◘ window

计算一个窗口时间内的所有RDD

window算子有两个参数：

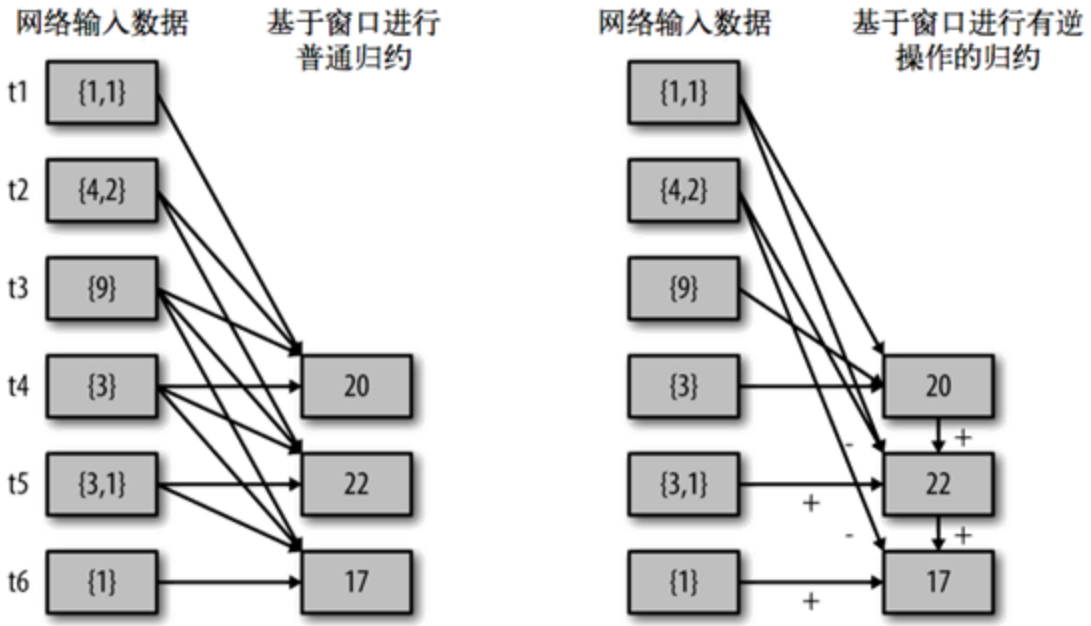
a) windowDuration窗口时间
b) slideDuration 滑动时间

```
//批次时间为2S
val ssc = new StreamingContext(sparkConf, Seconds(2))
// consume from rddQueue
val lines = ssc.queueStream(rddQueue)
// 第一个参数为window：8S
// 第二个参数为slide：4S（选填，默认为批次时间2S）
lines.window(Seconds(8),Seconds(4)).print()
```



Batch:2S　　Slide:4S　　Window:8S

- 算子示例：
  - window的操作原语 示例：




```
val ipDStream = accessLogsDStream.map(logEntry => (logEntry.getIpAddress(), 1))
val ipCountDStream = ipDStream.reduceByKeyAndWindow(
  {(x, y) => x + y},
  {(x, y) => x - y},
  Seconds(30),
  Seconds(10))
  //加上新进入窗口的批次中的元素 // 移除离开窗口的老批次中的元素 // 窗口时长// 滑动步长
```

➢ 算子示例：

◘ window的操作原语：

① window(windowLength, slideInterval): 基于对源DStream窗化的批次进行计算返回一个新的Dstream

② countByWindow(windowLength, slideInterval)：返回一个滑动窗口计数流中的元素。

③ reduceByWindow(func, windowLength, slideInterval)：通过使用自定义函数整合滑动区间流元素来创建一个新的单元素流。

④ reduceByKeyAndWindow(func, windowLength, slideInterval, [numTasks])：当在一个(K,V)对的DStream上调用此函数，会返回一个新(K,V)对的DStream，此处通过对滑动窗口中批次数据使用reduce函数来整合每个key的value值。Note:默认情况下，这个操作使用Spark的默认数量并行任务(本地是2)，在集群模式中依据配置属性(spark.default.parallelism)来做grouping。你可以通过设置可选参数numTasks来设置不同数量的tasks。

⑤ reduceByKeyAndWindow(func, invFunc, windowLength, slideInterval, [numTasks])：这个函数是上述函数的更高效版本，每个窗口的reduce值都是通过用前一个窗的reduce值来递增计算。通过reduce进入到滑动窗口数据并”反向reduce”离开窗口的旧数据来实现这个操作。一个例子是随着窗口滑动对keys的“加”“减”计数。通过前边介绍可以想到，这个函数只适用于”可逆的reduce函数”，也就是这些reduce函数有相应的”反reduce”函数(以参数invFunc形式传入)。如前述函数，reduce任务的数量通过可选参数来配置。注意：为了使用这个操作，检查点必须可用。

⑥ countByValueAndWindow(windowLength,slideInterval, [numTasks])：对(K,V)对的DStream调用，返回(K,Long)对的新DStream，其中每个key的值是其在滑动窗口中频率。如上，可配置reduce任务数量。

Q&A

TRANSWARP
星 环 科 技

温故知新

- RDD的“弹性”主要体现在哪里?
- RDD宽依赖为什么又称为Shuffle依赖?
- Spark运行模式有几种? Driver的主要功能是什么?
- 简述Spark的程序执行过程。
- DAGScheduler是如何划分Task的?