

# 大数据技术原理与应用

## 10. Spark

陈建文

电子信息与通信学院

[chenjw@hust.edu.cn](mailto:chenjw@hust.edu.cn)

# 10. Spark

## 10.1 Spark概述

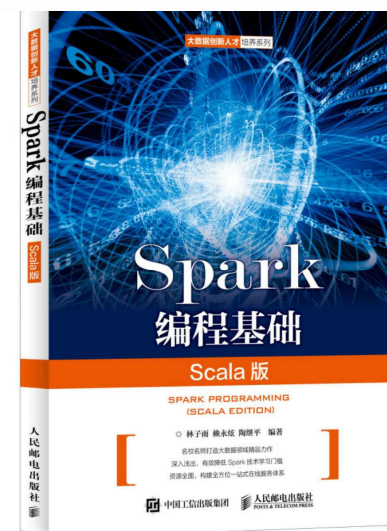
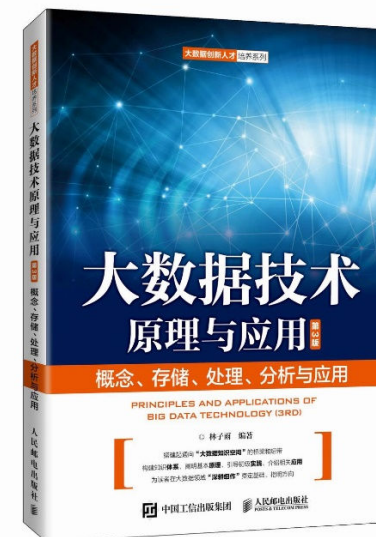
## 10.2 Spark生态系统

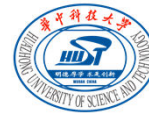
## 10.3 Spark运行架构

## 10.4 Spark SQL

## 10.5 Spark部署和应用方式

## 10.6 Spark编程实践





# 10.1 Spark概述

---

- 10.1.1 Spark介绍
- 10.1.2 Scala简介
- 10.1.3 Spark与Hadoop比较

## 10.1.1 Spark介绍

- **Spark**最初由美国加州伯克利大学（**UCBerkeley**）的**AMP**实验室于**2009**年开发，是基于内存计算的大数据并行计算框架，可用于构建大型的、低延迟的数据分析应用程序；
- **2013**年**Spark**加入**Apache**孵化器项目后发展迅猛，如今已成为**Apache**软件基金会最重要的三大分布式计算系统开源项目之一（**Hadoop**、**Spark**、**Storm**）；
- **Spark**在**2014**年打破了**Hadoop**保持的基准排序纪录
  - **Spark/206**个节点/**23**分钟/**100TB**数据
  - **Hadoop/2000**个节点/**72**分钟/**100TB**数据
  - **Spark**用十分之一的计算资源，获得了比**Hadoop**快**3**倍的速度

## Spark具有如下几个主要特点:

- **运行速度快:** 使用DAG执行引擎以支持循环数据流与内存计算;
- **容易使用:** 支持使用Scala、Java、Python和R语言进行编程, 可以通过Spark Shell进行交互式编程;
- **通用性:** Spark提供了完整而强大的技术栈, 包括SQL查询、流式计算、机器学习和图算法组件;
- **运行模式多样:** 可运行于独立的集群模式中, 可运行于Hadoop中, 也可运行于Amazon EC2等云环境中, 并且可以访问HDFS、Cassandra、HBase、Hive等多种数据源。

**Spark**如今已吸引了国内外各大公司的注意，如腾讯、淘宝、百度、亚马逊等公司均不同程度地使用了**Spark**来构建大数据分析应用，并应用到实际的生产环境中。

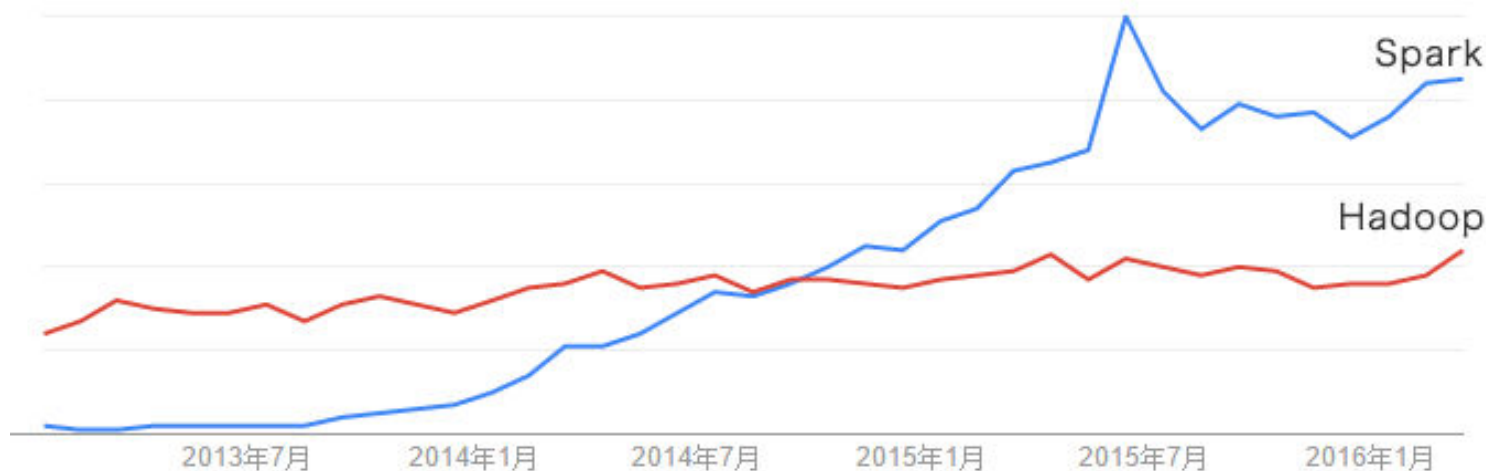


图10-1 谷歌趋势：**Spark**与**Hadoop**对比

## 10.1.2 Scala简介

**Scala**是一门现代的多范式编程语言，运行于**Java**平台（**JVM**，**Java** 虚拟机），并兼容现有的**Java**程序

**Scala**的特性：

- **Scala**具备强大的并发性，支持函数式编程，可以更好地支持分布式系统；
- **Scala**语法简洁，能提供优雅的API，**Scala**兼容**Java**，运行速度快，且能融合到**Hadoop**生态圈中。

**Scala**是**Spark**的主要编程语言，但**Spark**还支持**Java**、**Python**、**R**作为编程语言

**Scala**的优势是提供了**REPL**（**Read-Eval-Print Loop**，交互式解释器），提高程序开发效率

## 10.1.3 Spark与Hadoop对比

Hadoop存在如下一些缺点：

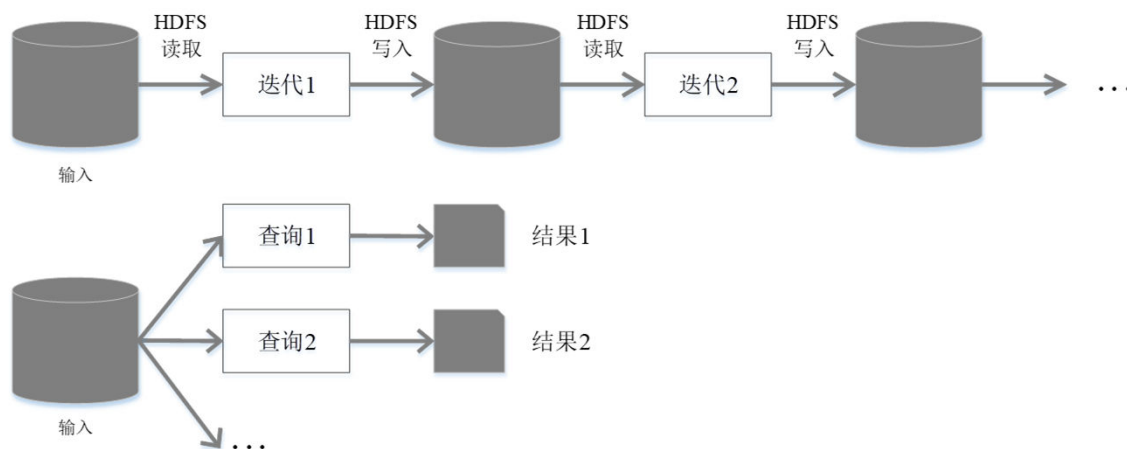
- 表达能力有限
- 磁盘IO开销大
- 延迟高
  - 任务之间的衔接涉及IO开销；
  - 在前一个任务执行完成之前，其他任务就无法开始，难以胜任复杂、多阶段的计算任务。



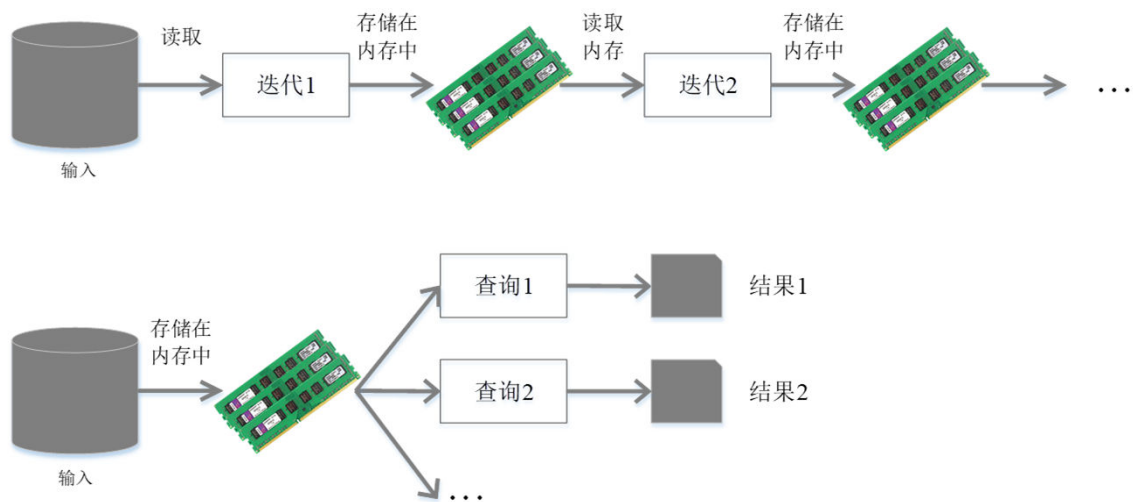
**Spark**在借鉴**Hadoop MapReduce**优点的同时，很好地解决了**MapReduce**所面临的问题。

相比于**Hadoop MapReduce**，**Spark**主要具有如下优点：

- **Spark**的计算模式也属于**MapReduce**，但不局限于**Map**和**Reduce**操作，还提供了多种数据集操作类型，编程模型比**Hadoop MapReduce**更灵活；
- **Spark**提供了内存计算，可将中间结果放到内存中，对于迭代运算效率更高。
- **Spark**基于**DAG**的任务调度执行机制，要优于**Hadoop MapReduce**的迭代执行机制。



(a) Hadoop MapReduce执行流程



(b) Spark执行流程

图10-2. Hadoop与Spark的执行流程对比

- 使用**Hadoop**进行迭代计算非常耗资源；
- **Spark**将数据载入内存后，之后的迭代计算都可以直接使用内存中的中间结果作运算，避免了从磁盘中频繁读取数据。

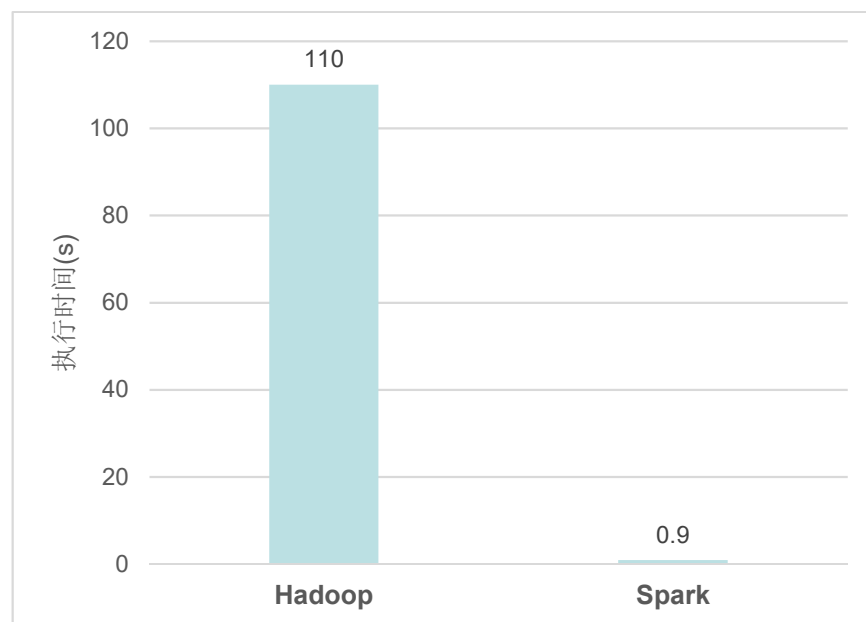


图10-3. Hadoop与Spark执行逻辑回归的时间对比

## 10.2 Spark生态系统

在实际应用中，大数据处理主要包括以下三个类型：

- 复杂的批量数据处理：通常时间跨度在数十分钟到数小时之间
- 基于历史数据的交互式查询：通常时间跨度在数十秒到数分钟之间
- 基于实时数据流的数据处理：通常时间跨度在数百毫秒到数秒之间

当同时存在以上三种场景时，就需要同时部署三种不同的软件

- 比如: **MapReduce / Impala / Storm**

这样做难免会带来一些问题：

- 不同场景之间输入输出数据无法做到无缝共享，通常需要进行数据格式的转换
- 不同的软件需要不同的开发和维护团队，带来了较高的使用成本
- 比较难以对同一个集群中的各个系统进行统一的资源协调和分配

## Spark 具有优势:

- Spark的设计遵循“一个软件栈满足不同应用场景”的理念，逐渐形成了一套完整的生态系统；
- 既能够提供内存计算框架，也可以支持SQL即席查询、实时流式计算、机器学习和图计算等；
- Spark可以部署在资源管理器YARN之上，提供一站式的大数据解决方案；

因此，**Spark**所提供的生态系统足以应对上述三种场景，即同时支持批处理、交互式查询和流数据处理。

**Spark**生态系统已经成为伯克利数据分析软件栈**BDAS** (**Berkeley Data Analytics Stack**) 的重要组成部分。

Access and Interfaces	Spark Streaming	BlinkDB	GraphX	MLBase
		Spark SQL		MLlib
Processing Engine	Spark Core			
Storage	Tachyon			
	HDFS, S3			
Resource Virtualization	Mesos		Hadoop Yarn	

图10-4. BDAS架构

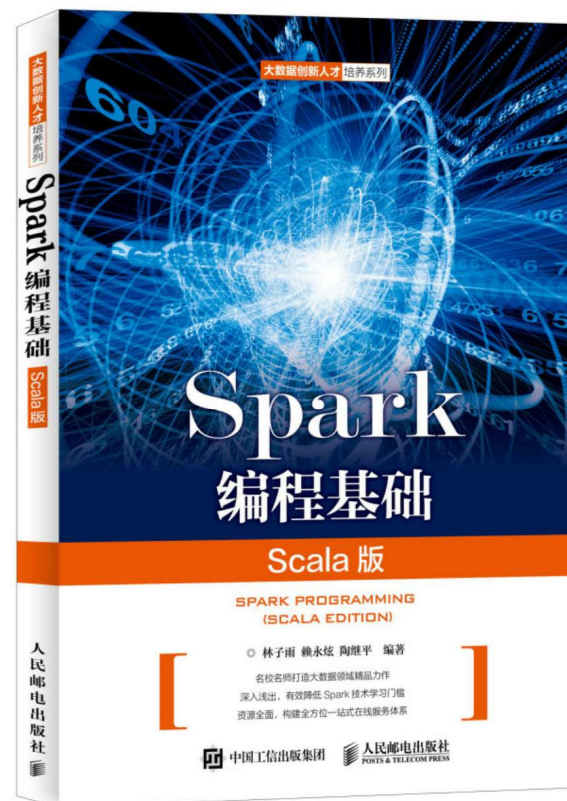
**Spark**的生态系统主要包含了**Spark Core**、**Spark SQL**、**Spark Streaming**、**MLlib**和**GraphX** 等组件。

表1 Spark生态系统组件的应用场景

应用场景	时间跨度	其他框架	Spark生态系统中的组件
复杂的批量数据处理	小时级	MapReduce、Hive	Spark
基于历史数据的交互式查询	分钟级、秒级	Impala、Dremel、Drill	Spark SQL
基于实时数据流的数据处理	毫秒、秒级	Storm、S4	Spark Streaming
基于历史数据的数据挖掘	-	Mahout	MLlib
图结构数据的处理	-	Pregel、Hama	GraphX

## 10.3 Spark运行架构

- 10.3.1 Spark 基本概念
- 10.3.2 Spark 架构设计
- 10.3.3 Spark 运行基本流程
- 10.3.4 RDD 运行原理





## 10.3.1 Spark基本概念

- **RDD:** 是Resilient Distributed Dataset（弹性分布式数据集）的简称，是分布式内存的一个抽象概念，提供了一种高度受限的共享内存模型；
- **DAG:** 是Directed Acyclic Graph（有向无环图）的简称，反映RDD之间的依赖关系；
- **Executor:** 是运行在工作节点（WorkerNode）的一个进程，负责运行Task；
- **Application:** 用户编写的Spark应用程序；
- **Task:** 运行在Executor上的工作单元；
- **Job:** 一个Job包含多个RDD及作用于相应RDD上的各种操作；
- **Stage:** 是Job的基本调度单位，一个Job会分为多组Task，每组Task被称为Stage，或者也被称为TaskSet，代表了一组关联的、相互之间没有Shuffle依赖关系的任务组成的任务集。

## 10.3.2 Spark架构设计

- Spark运行架构包括集群资源管理器（**Cluster Manager**）、运行作业任务的工作节点（**Worker Node**）、每个应用的任务控制节点（**Driver**）和每个工作节点上负责具体任务的执行进程（**Executor**）；
- 资源管理器可以自带或 **Mesos** 或 **YARN** 。

与Hadoop MapReduce计算框架相比，Spark所采用的Executor有两个优点：

- 一是利用多线程来执行具体的任务，减少任务的启动开销；
- 二是Executor中有一个BlockManager存储模块，会将内存和磁盘共同作为存储设备，有效减少IO开销。

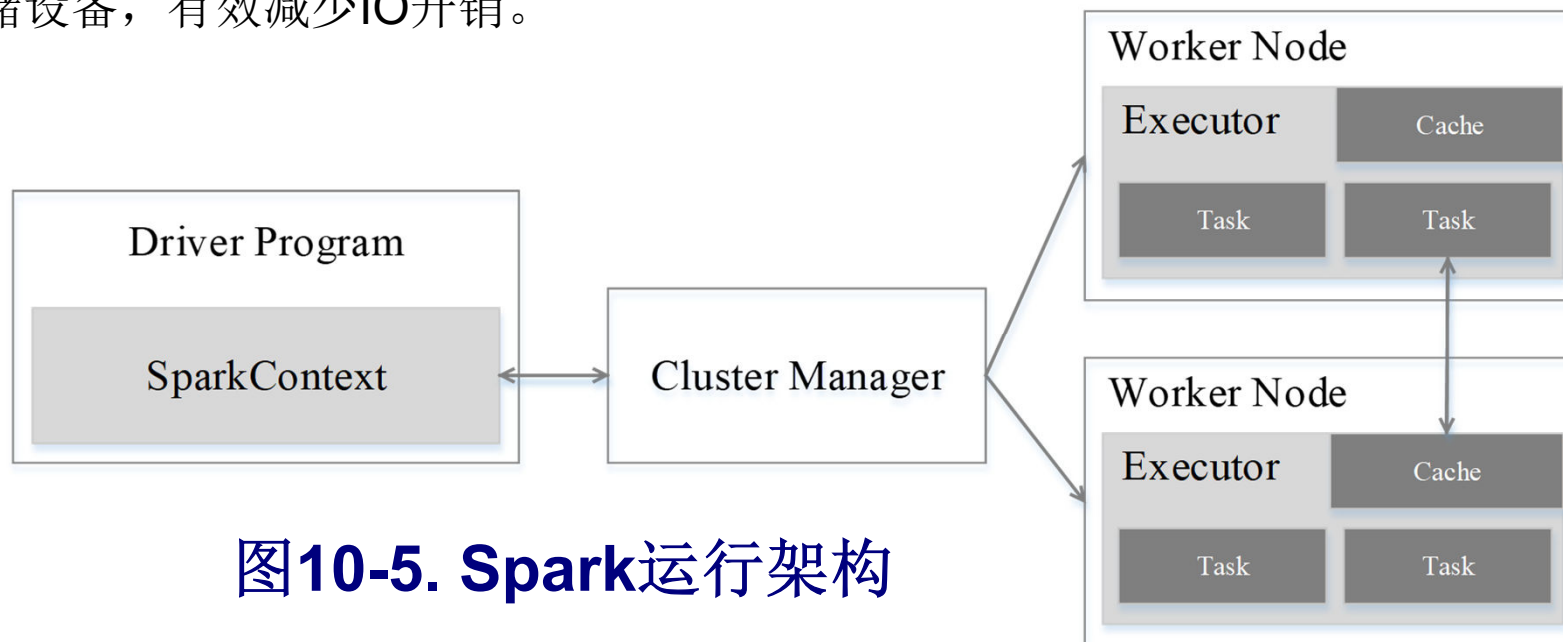


图10-5. Spark运行架构

- 一个 **Application** 由一个 **Driver** 和若干个 **Job** 构成，一个 **Job** 由多个 **Stage** 构成，一个 **Stage** 由多个没有 **Shuffle** 关系的 **Task** 组成；
- 当执行一个 **Application** 时，**Driver** 会向集群管理器申请资源，启动 **Executor**，并向 **Executor** 发送应用程序代码和文件，然后在 **Executor** 上执行 **Task**，运行结束后，执行结果会返回给 **Driver**，或者写到 **HDFS** 或者其他数据库中。

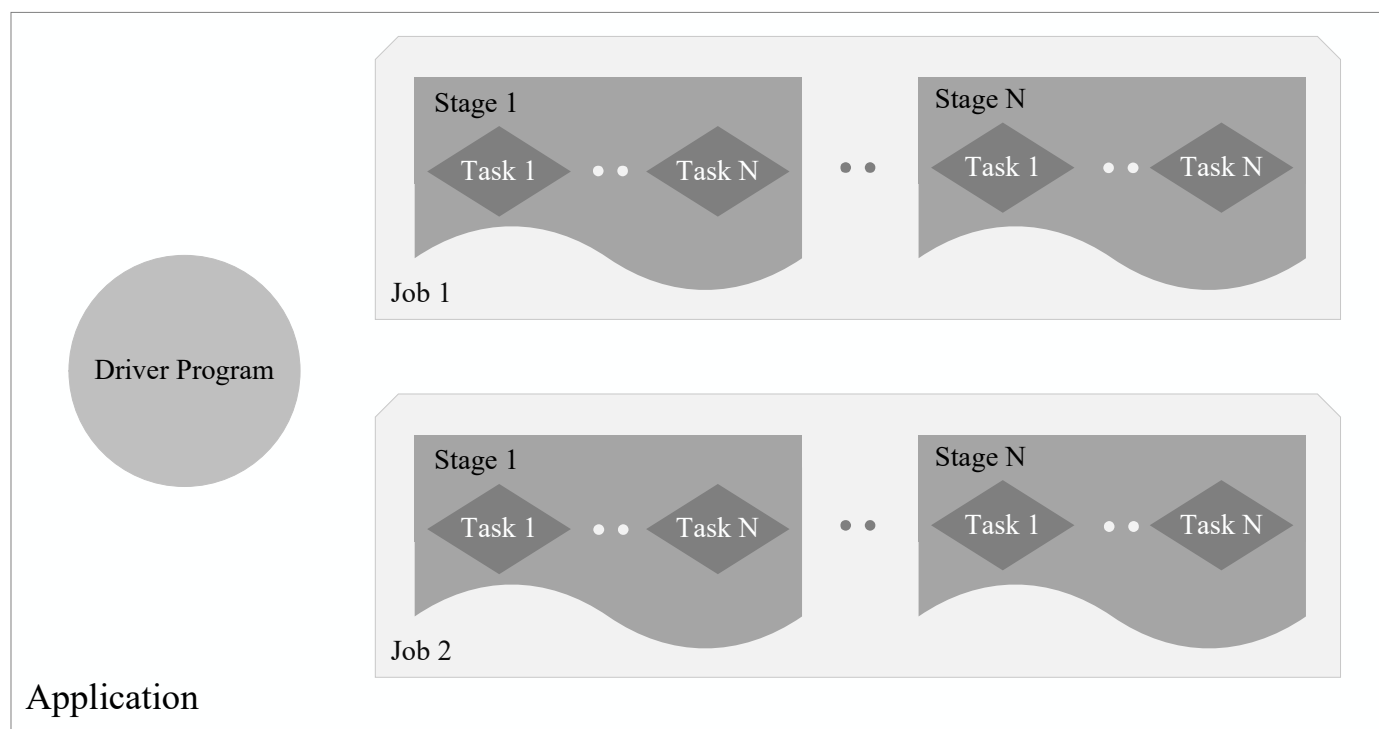


图10-6. Spark中各种概念之间的相互关系

## 10.3.3 Spark运行基本流程

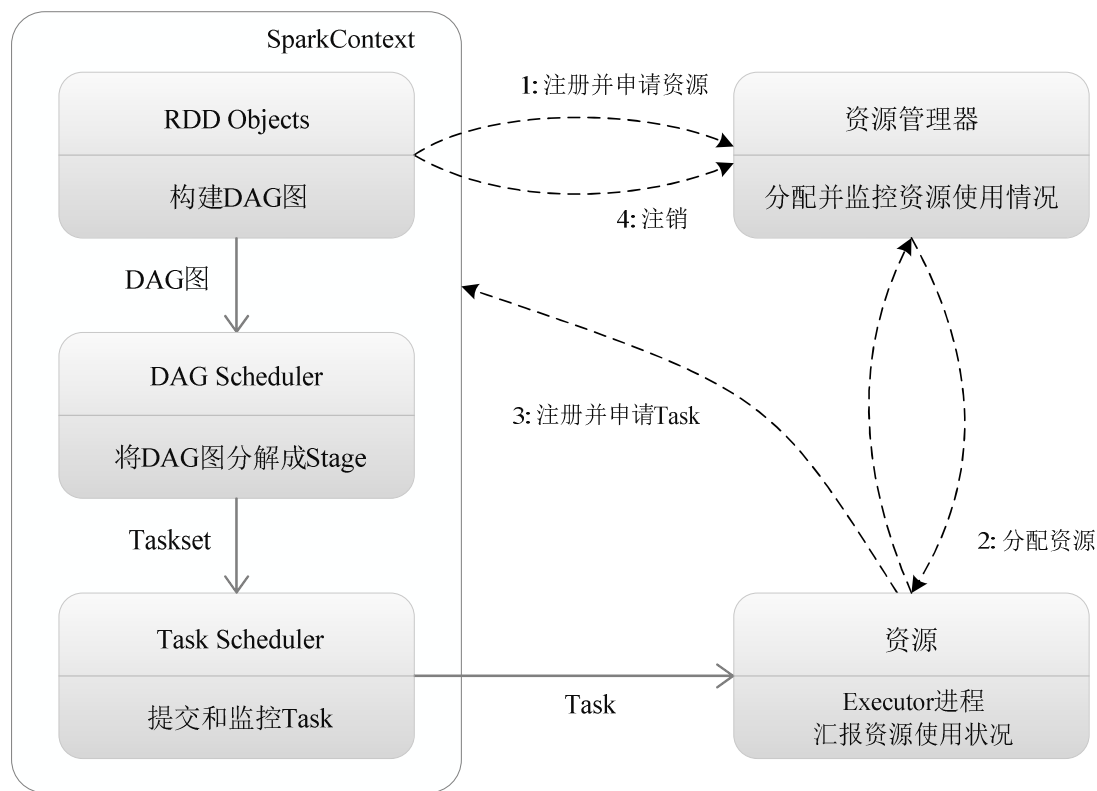


图10-7. Spark运行基本流程图

（1）首先为应用构建起基本的运行环境，即由Driver创建一个SparkContext，进行资源的申请、任务的分配和监控；

（2）资源管理器为Executor分配资源，并启动Executor进程

（3）SparkContext根据RDD的依赖关系构建DAG图，DAG图提交给DAGScheduler解析成Stage，然后把一个个TaskSet提交给底层调度器TaskScheduler处理；Executor向SparkContext申请Task，Task Scheduler将Task发放给Executor运行，并提供应用程序代码

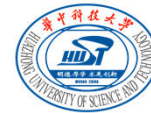
（4）Task在Executor上运行，把执行结果反馈给TaskScheduler，然后反馈给DAGScheduler，运行完毕后写入数据并释放所有资源

总体而言，**Spark**运行架构具有以下特点：

（1）每个**Application**都有自己专属的**Executor**进程，并且该进程在**Application**运行期间一直驻留。**Executor**进程以多线程的方式运行**Task**；

（2）**Spark**运行过程与资源管理器无关，只要能够获取**Executor**进程并保持通信即可；

（3）**Task**采用了数据本地性和推测执行等优化机制。



## 10.3.4 RDD运行原理

- 1. RDD设计背景
- 2. RDD概念
- 3. RDD特性
- 4. RDD之间的依赖关系
- 5. RDD阶段划分
- 6. RDD运行过程

## 1.RDD设计背景

- 许多迭代式算法（比如机器学习、图算法等）和交互式数据挖掘工具，共同之处是，不同计算阶段之间会重用中间结果；
- 目前的**MapReduce**框架都是把中间结果写入到**HDFS**中，带来了大量的数据复制、磁盘**IO**和序列化开销；
- **RDD**就是为了满足这种需求而出现的，它提供了一个抽象的数据架构，我们不必担心底层数据的分布式特性，只需将具体的应用逻辑表达为一系列转换处理，不同**RDD**之间的转换操作形成依赖关系，可以实现管道化，避免中间数据存储。

## 2.RDD概念

- 一个**RDD**就是一个分布式对象集合，本质上是一个只读的分区记录集合，每个**RDD**可分成多个分区，每个分区就是一个数据集片段，并且一个**RDD**的不同分区可以被保存到集群中不同的节点上，从而可以在集群中的不同节点上进行并行计算；
- **RDD**提供了一种高度受限的共享内存模型，即**RDD**是只读的记录分区的集合，不能直接修改，只能基于稳定的物理存储中的数据集创建**RDD**，或者通过在其他**RDD**上执行确定的转换操作（如**map**、**join**和**group by**）而创建得到新的**RDD**；



- **RDD**提供了一组丰富的操作以支持常见的数据运算，分为“动作”（**Action**）和“转换”（**Transformation**）两种类型；
- **RDD**提供的转换接口都非常简单，都是类似**map**、**filter**、**groupBy**、**join**等粗粒度的数据转换操作，而不是针对某个数据项的细粒度修改（不适合网页爬虫）；
- 表面上**RDD**的功能很受限、不够强大，实际上**RDD**已经被实践证明可以高效地表达许多框架的编程模型（比如**MapReduce**、**SQL**、**Pregel**）
- **Spark**用**Scala**语言实现了**RDD**的**API**，程序员可以通过调用**API**实现对**RDD**的各种操作。

## RDD典型的执行过程如下：

- RDD读入外部数据源进行创建
- RDD经过一系列的转换（Transformation）操作，每一次都会产生不同的RDD，供给下一个转换操作使用；
- 最后一个RDD经过“动作”操作进行转换，并输出到外部数据源。

这一系列处理称为一个**Lineage**（血缘关系），即**DAG**拓扑排序的结果。  
优点：惰性调用、管道化、避免同步等待、不需要保存中间结果、每次操作变得简单。

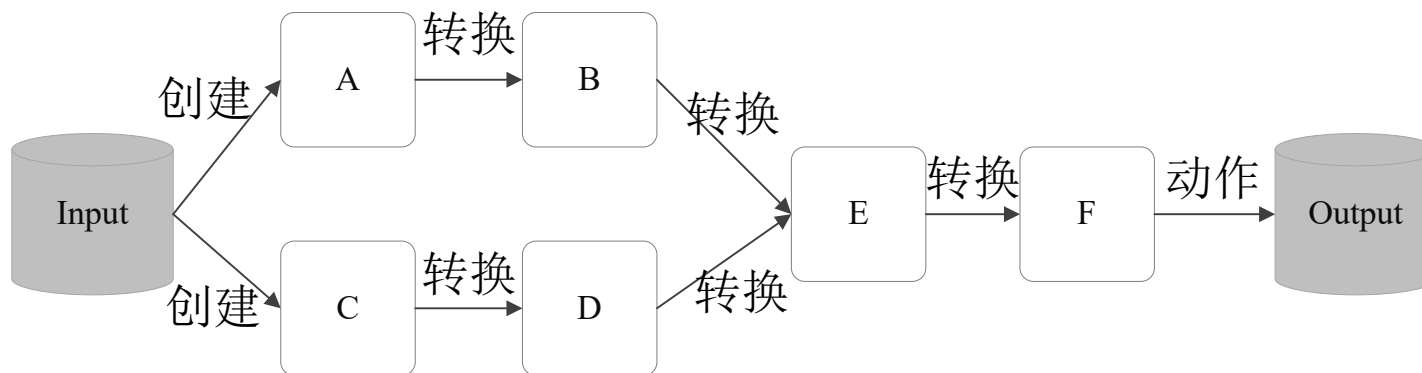


图10-8. RDD执行过程的一个实例

## 3.RDD特性

**Spark**采用**RDD**以后能够实现高效计算的原因主要在于：

(1) 高效的容错性

- 现有容错机制：数据复制或者记录日志；
- **RDD**：血缘关系、重新计算丢失分区、无需回滚系统、重算过程在不同节点之间并行、只记录粗粒度的操作。

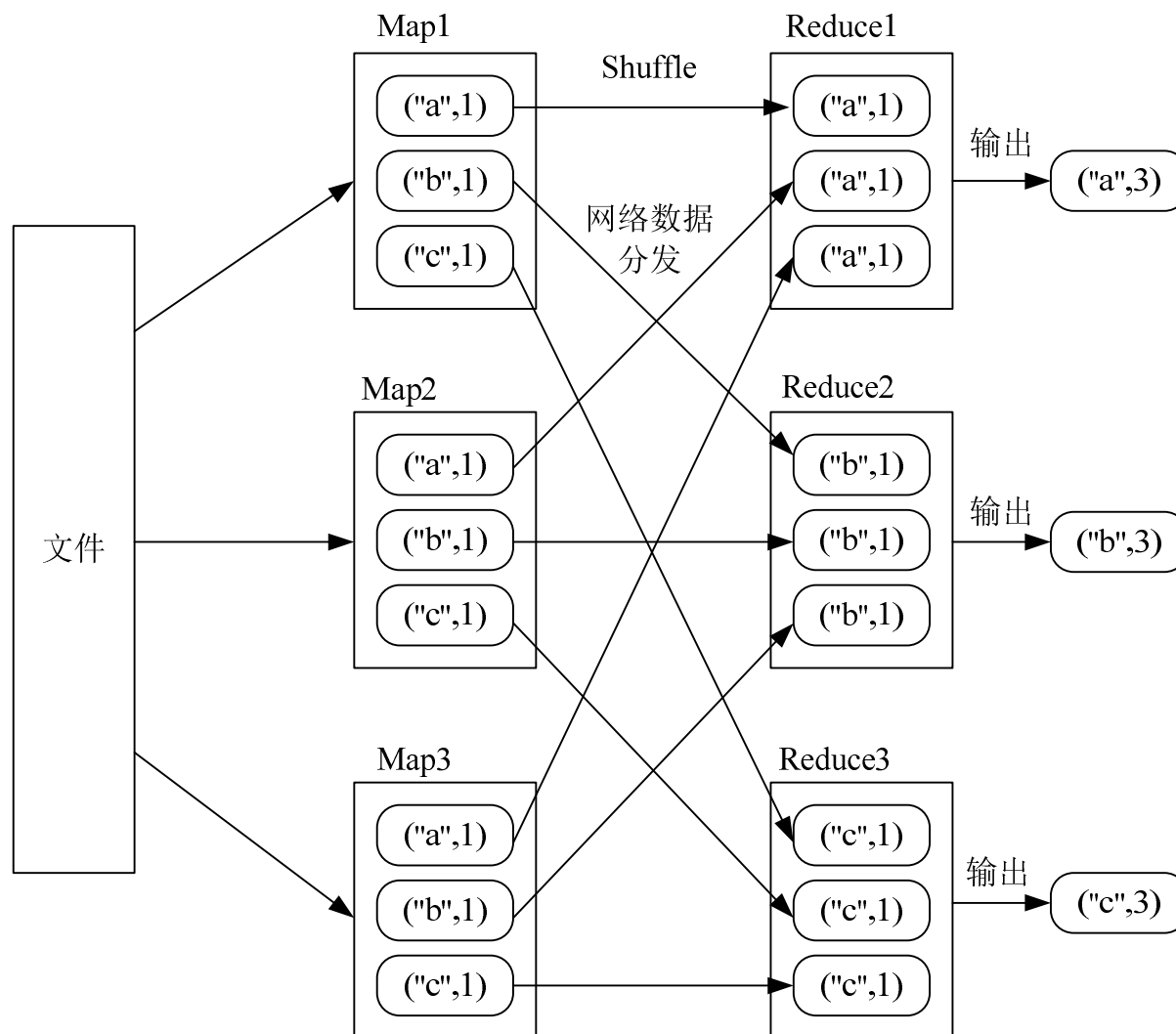
(2) 中间结果持久化到内存，数据在内存中的多个 **RDD**操作之间进行传递，避免了不必要的读写磁盘开销；

(3) 存放的数据可以是**Java**对象，避免了不必要的对象序列化和反序列化。

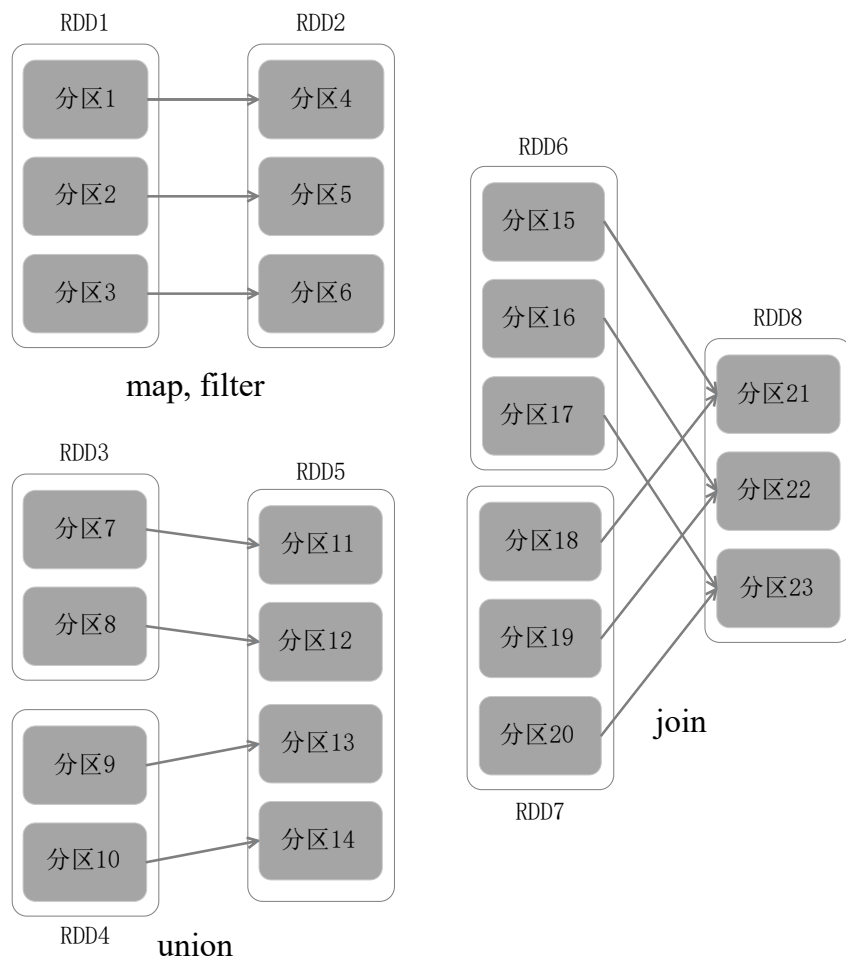
## 4. RDD之间的依赖关系

- **Shuffle**操作  
什么是**Shuffle**操作？
- 窄依赖和宽依赖  
是否包含**Shuffle**操作是区分窄依赖和宽依赖的根据！

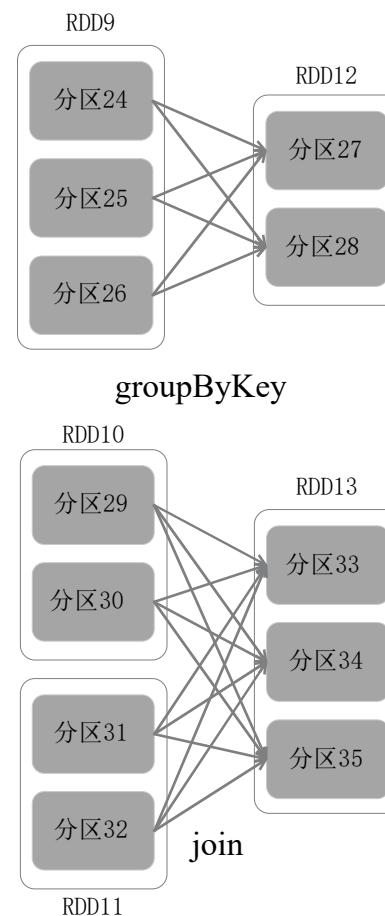
## Shuffle操作



## 窄依赖与宽依赖



(a)窄依赖



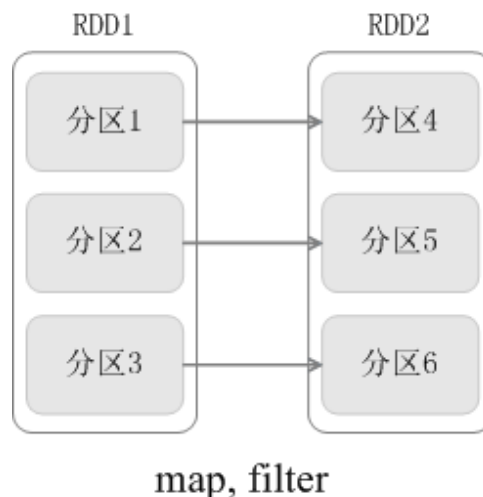
(b)宽依赖

- 窄依赖表现为一个父**RDD**的分区对应于一个子**RDD**的分区或多个父**RDD**的分区对应于一个子**RDD**的分区；
- 宽依赖则表现为存在一个父**RDD**的一个分区对应一个子**RDD**的多个分区。

图10-9. 窄依赖与宽依赖的区别

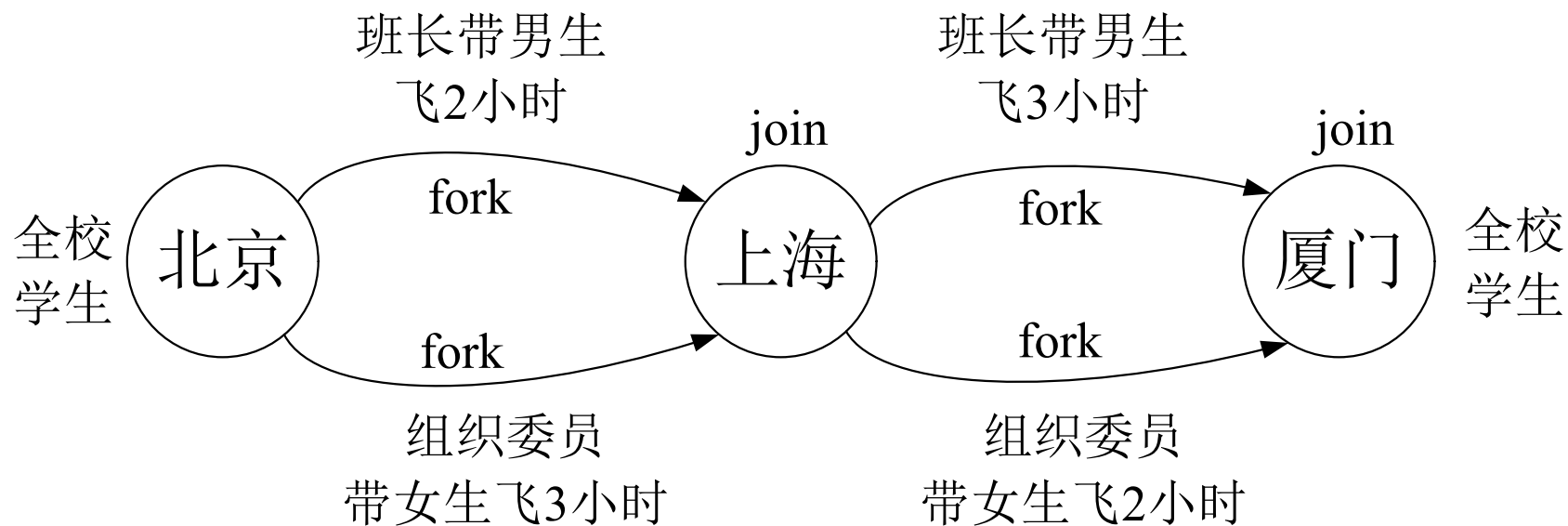
## 5.RDD阶段划分

**Spark** 根据**DAG** 图中的**RDD** 依赖关系，把一个作业分成多个阶段。阶段划分的依据是窄依赖和宽依赖。对于宽依赖和窄依赖而言，窄依赖对于作业的优化很有利，宽依赖无法优化逻辑上，每个**RDD** 操作都是一个**fork/join**（一种用于并行执行任务的框架），把计算**fork** 到每个**RDD** 分区，完成计算后对各个分区得到的结果进行**join** 操作，然后**fork/join** 下一个**RDD** 操作。

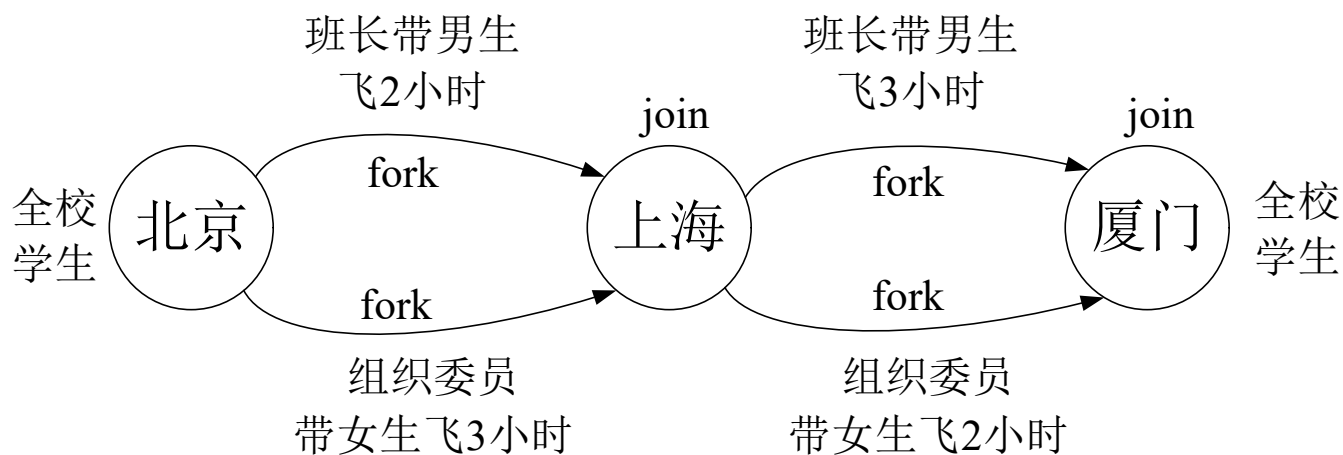


## fork/join的优化原理:

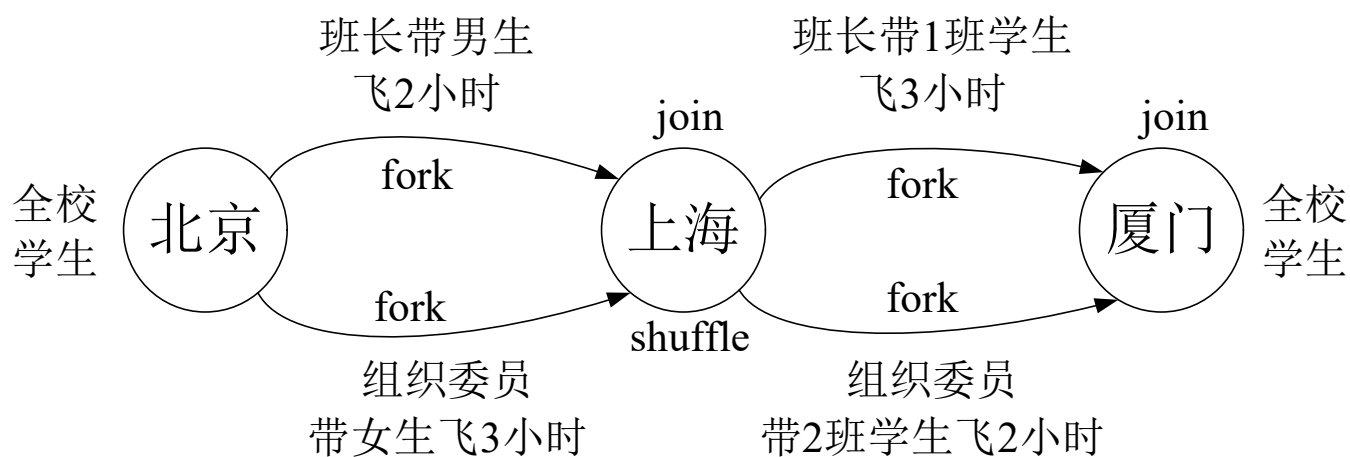
举例：一个学校（含**2**个班级）完成从北京到厦门的长征。





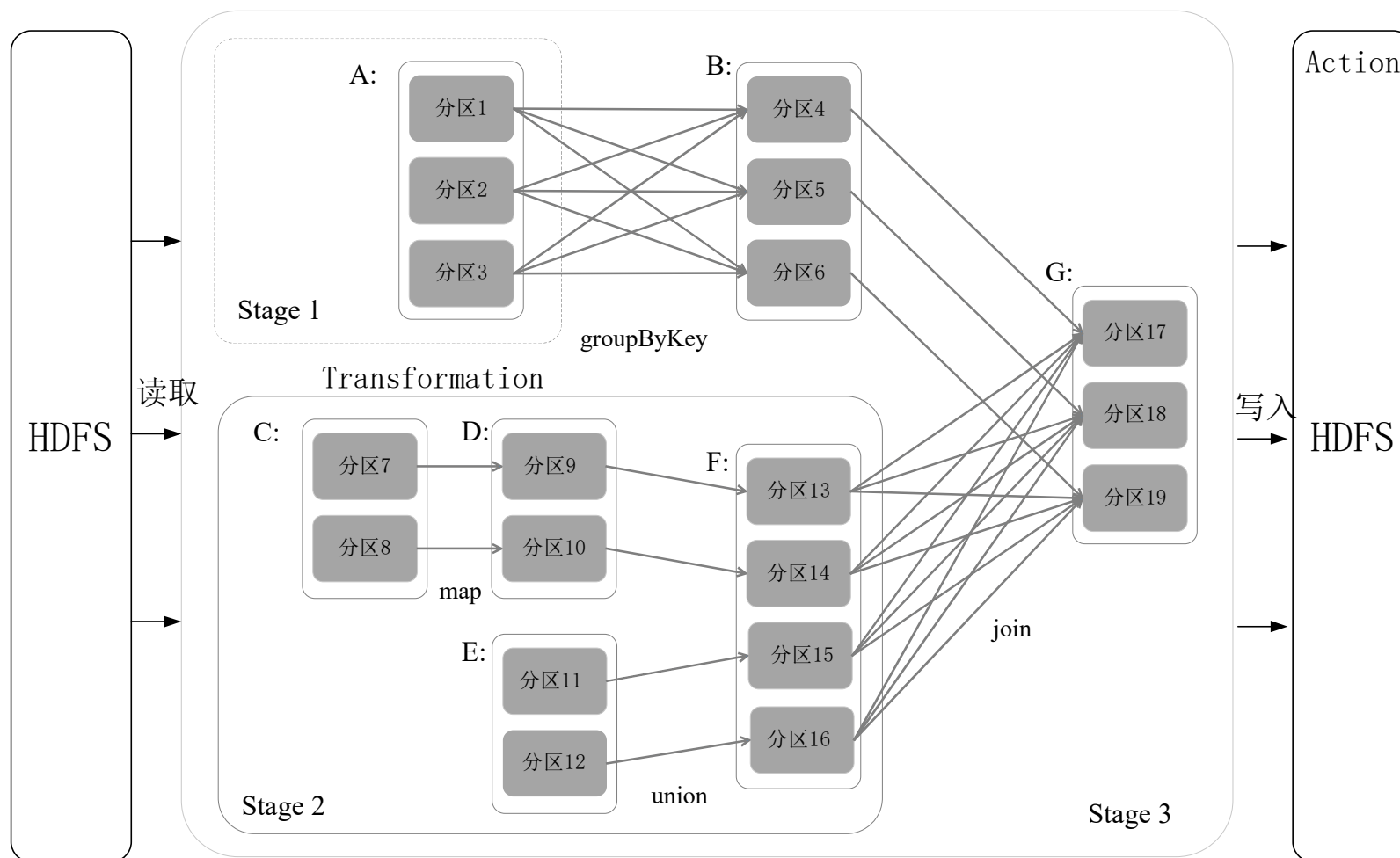


(a) 窄依赖



(a) 宽依赖

## 窄依赖可以实现“流水线”优化 宽依赖无法实现“流水线”优化

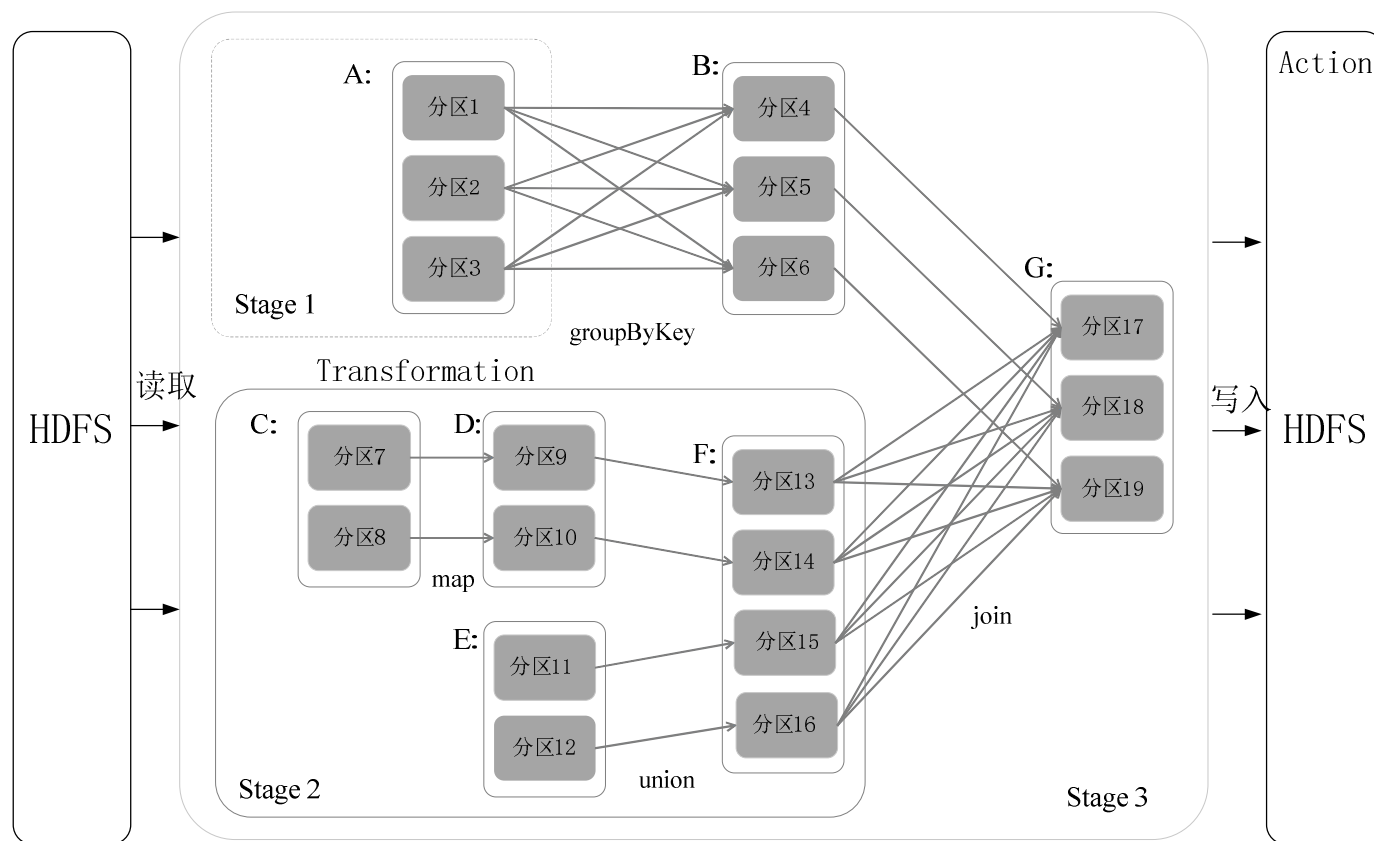


**Spark**根据**DAG**图中的**RDD**依赖关系，把一个作业分成多个阶段。对于宽依赖和窄依赖而言，窄依赖对于作业的优化很有利。只有窄依赖可以实现流水线优化，宽依赖包含**Shuffle**过程，无法实现流水线方式处理。

**Spark**通过分析各个**RDD**的依赖关系生成了**DAG**，再通过分析各个**RDD**中的分区之间的依赖关系来决定如何划分**Stage**，具体划分方法是：

- 在**DAG**中进行反向解析，遇到宽依赖就断开；
- 遇到窄依赖就把当前的**RDD**加入到**Stage**中；
- 将窄依赖尽量划分在同一个**Stage**中，可以实现流水线计算。

流水线操作实例：被分成三个**Stage**，在**Stage2**中，从**map**到**union**都是窄依赖，这两步操作可以形成一个流水线操作。



流水线操作实例：  
分区7通过**map**操作生成的分区9，  
可以不用等待分区8到分区10这个  
**map**操作的计算结束，而是继续进行**union**操作，  
得到分区13，这样流水线执行大大提高了计算的效率。

图10-10.根据RDD分区的依赖关系划分Stage

## 6.RDD运行过程

通过上述对**RDD**概念、依赖关系和**Stage**划分的介绍，结合之前介绍的**Spark**运行基本流程，再总结一下**RDD**在**Spark**架构中的运行过程：

- (1) 创建**RDD**对象；
- (2) **SparkContext**负责计算**RDD**之间的依赖关系，构建**DAG**；
- (3) **DAGScheduler**负责把**DAG**图分解成多个**Stage**，每个**Stage**中包含了多个**Task**，每个**Task**会被**TaskScheduler**分发给各个**WorkerNode**上的**Executor**去执行。

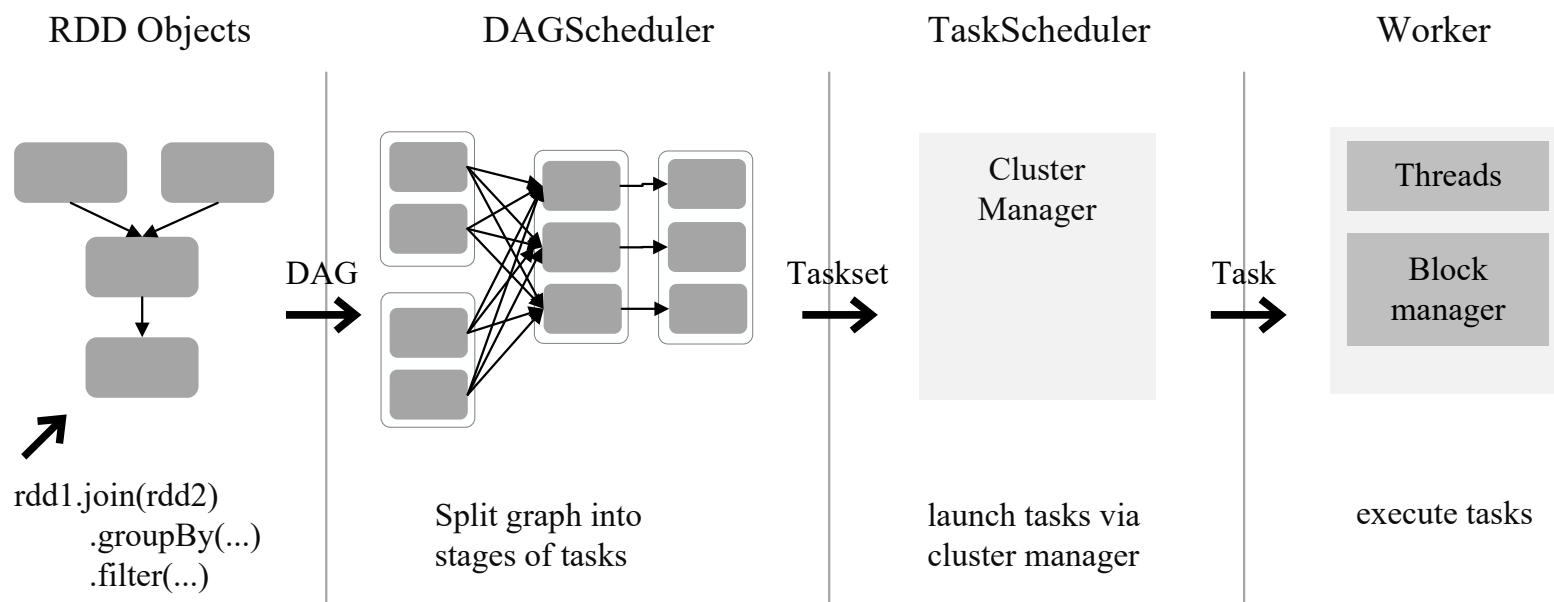
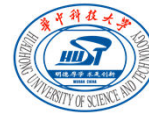


图10-11. RDD在Spark中的运行过程



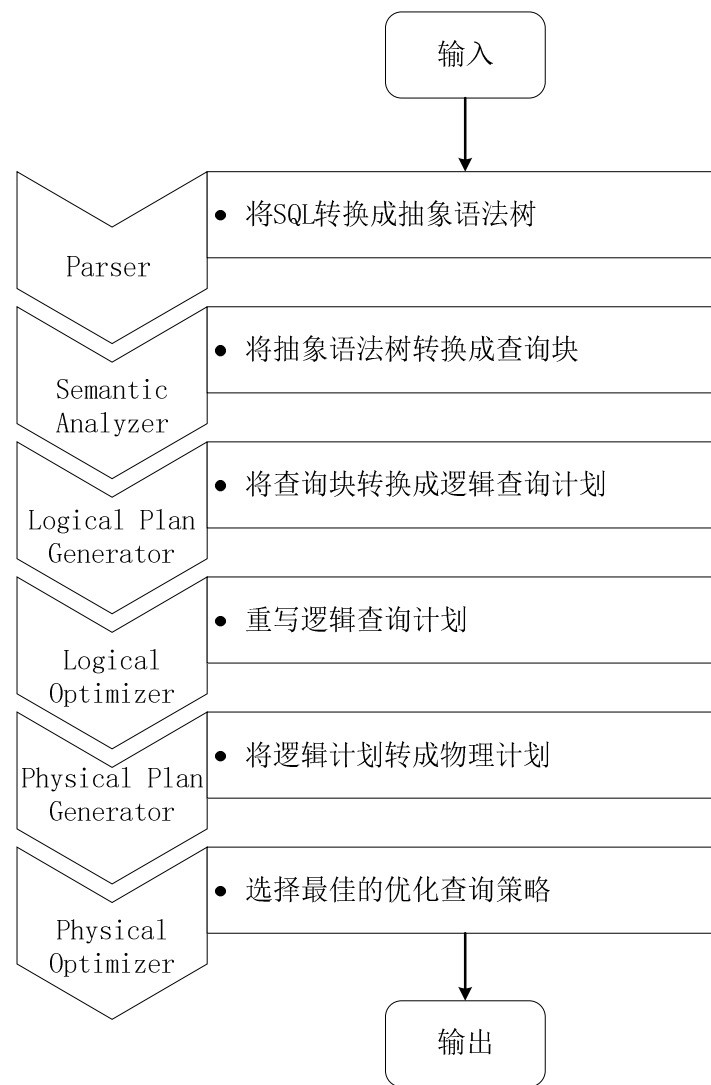
# 10.4 Spark SQL

---

- 10.4.1 从Shark说起
- 10.4.2 Spark SQL设计

## 10.4.1 从Shark说起

- Shark即Hive on Spark，为了实现与Hive兼容，Shark在HiveQL方面重用了Hive中HiveQL的解析、逻辑执行计划翻译、执行计划优化等逻辑，可以近似认为仅将物理执行计划从MapReduce作业替换成了Spark作业，通过Hive的HiveQL解析，把HiveQL翻译成Spark上的RDD操作。
- Shark的设计导致了两个问题：
  - 一是执行计划优化完全依赖于Hive，不方便添加新的优化策略；
  - 二是因为Spark是线程级并行，而MapReduce是进程级并行，因此，Spark在兼容Hive的实现上存在线程安全问题，导致Shark不得不使用另外一套独立维护的打了补丁的Hive源码分支



Hive中SQL查询的MapReduce作业转化过程

## 10.4.2 Spark SQL设计

**Spark SQL**在**Hive**兼容层面仅依赖**HiveQL**解析、**Hive**元数据，也就是说，从**HQL**被解析成抽象语法树（**AST**）起，就全部由**Spark SQL**接管了。**Spark SQL**执行计划生成和优化都由**Catalyst**（函数式关系查询优化框架）负责

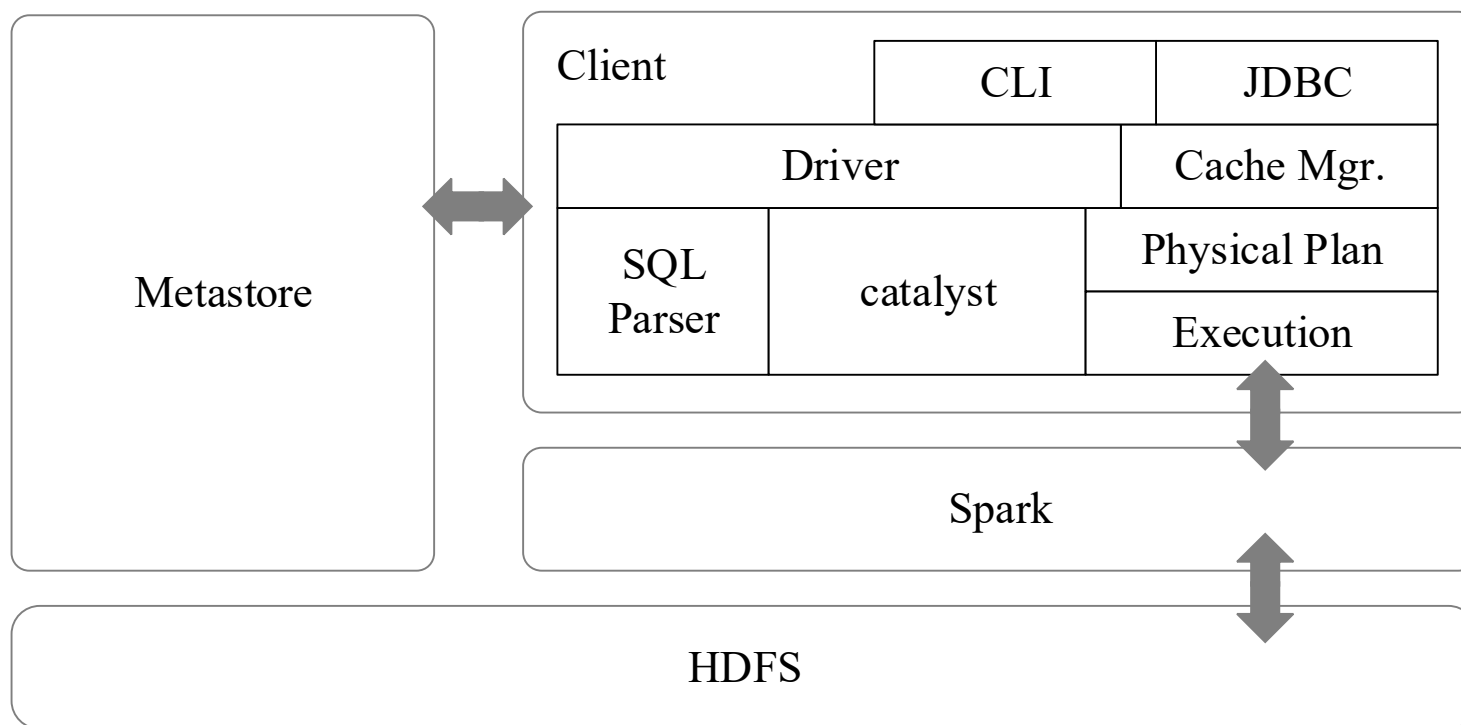


图10-12 Spark SQL架构



- **Spark SQL**增加了**SchemaRDD**（即带有**Schema**信息的**RDD**），使用户可以在**Spark SQL**中执行**SQL**语句，数据既可以来自**RDD**，也可以是**Hive**、**HDFS**、**Cassandra**等外部数据源，还可以是**JSON**格式的数据
- **Spark SQL**目前支持**Scala**、**Java**、**Python**三种语言，支持**SQL-92**规范

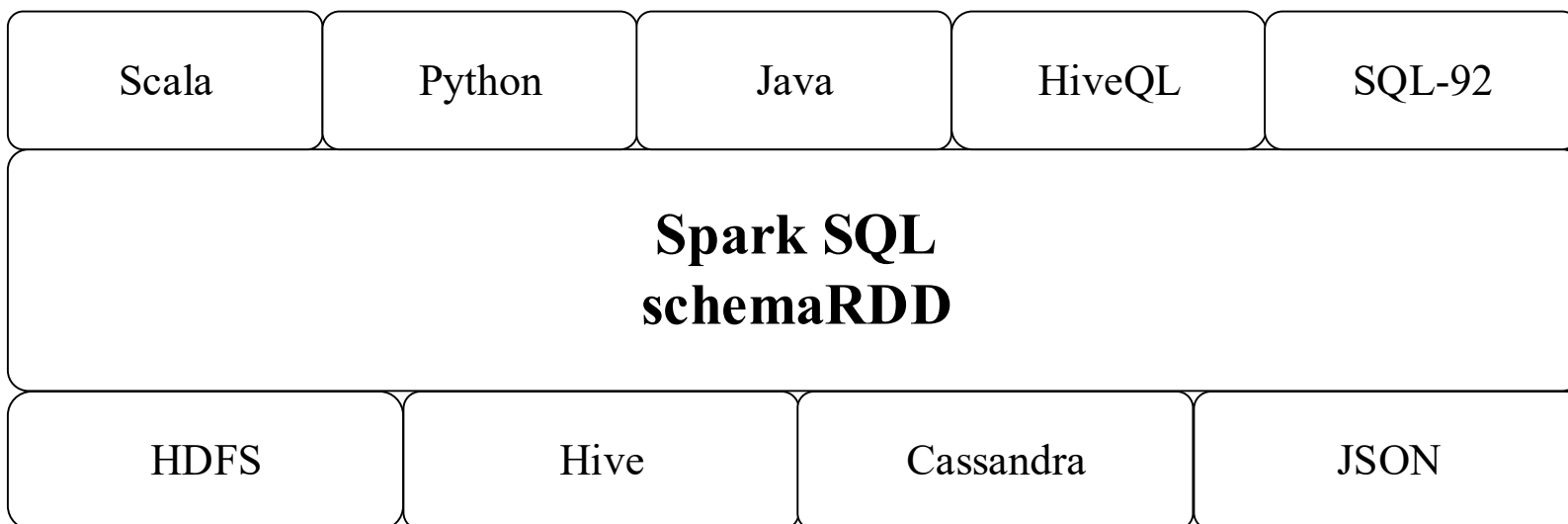
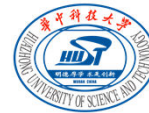


图10-13 Spark SQL支持的数据格式和编程语言



## 10.5 Spark部署和应用

---

- 10.5.1 Spark三种部署方式
- 10.5.2 从Hadoop+Storm架构转向Spark架构
- 10.5.3 Hadoop和Spark的统一部署

## 10.5.1 Spark三种部署方式

Spark支持三种不同类型的部署方式，包括：

- **Standalone**（类似于MapReduce1.0，slot为资源分配单位）
- **Spark on Mesos**（和Spark有血缘关系，更好支持Mesos）
- **Spark on YARN**

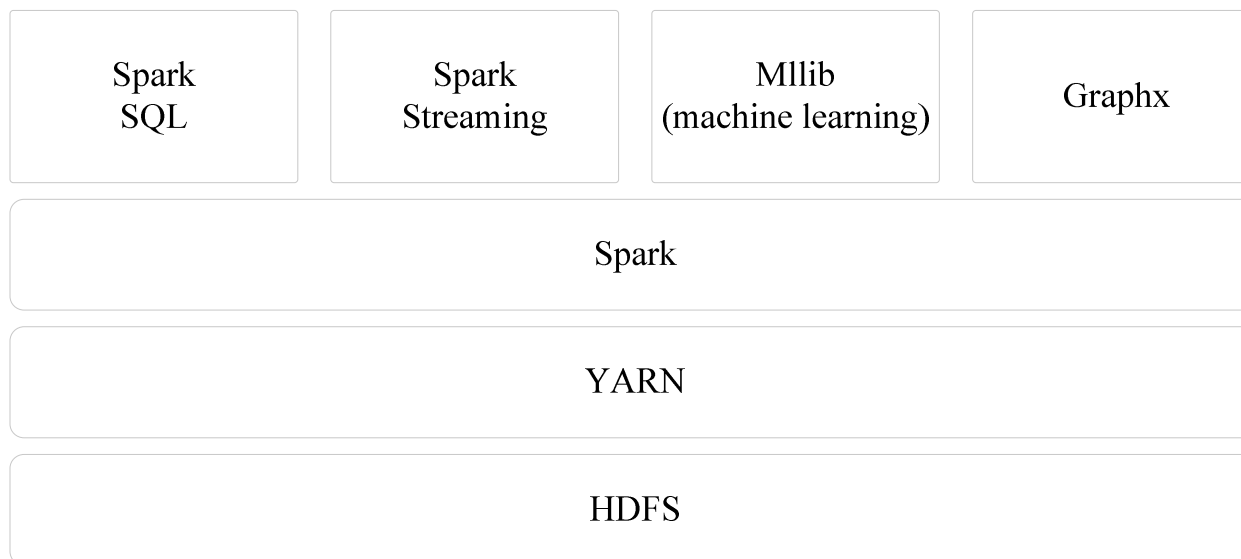


图9-15. Spark on Yarn架构

## 10.5.2 从Hadoop+Storm架构转向Spark架构

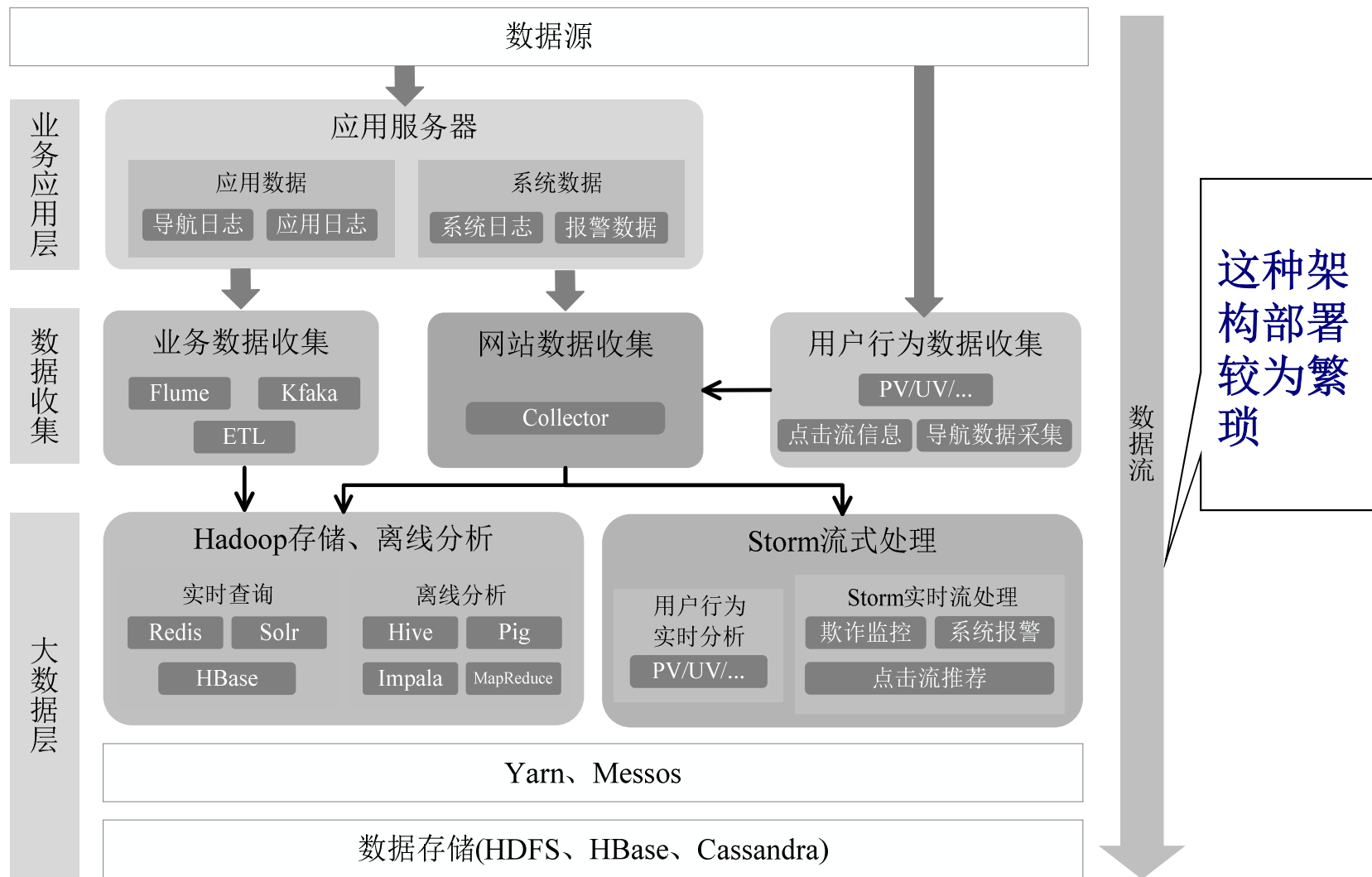


图9-16. 采用Hadoop+Storm部署方式的一个案例

用**Spark**架构具有如下优点：

- 实现一键式安装和配置、线程级别的任务监控和告警
- 降低硬件集群、软件维护、任务监控和应用开发的难度
- 便于做成统一的硬件、计算平台资源池

需要说明的是，**Spark Streaming**无法实现毫秒级的流计算，因此，对于需要毫秒级实时响应的企业应用而言，仍然需要采用流计算框架（如**Storm**）

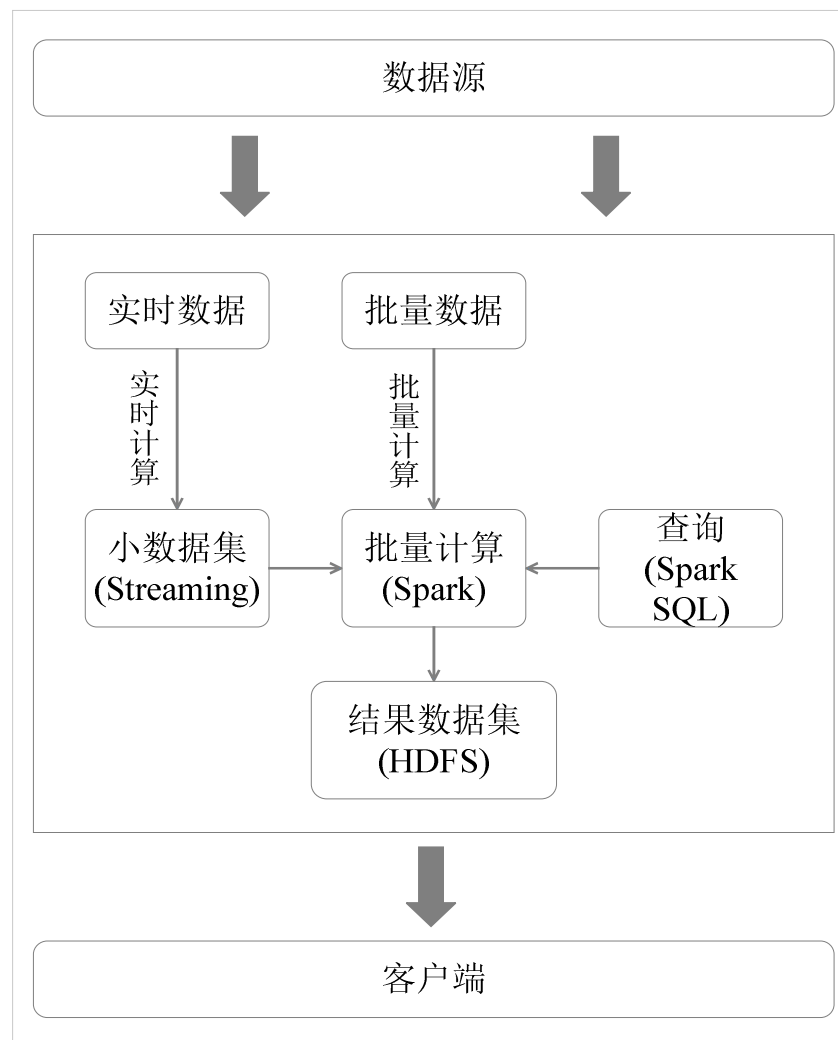


图10-19 用Spark架构满足批处理和流处理需求

## 10.5.3 Hadoop和Spark统一部署

- 由于Hadoop生态系统中的一些组件所实现的功能，目前还是无法由Spark取代的，比如，Storm
- 现有的Hadoop组件开发的应用，完全转移到Spark上需要一定的成本

不同的计算框架统一运行在YARN中，可以带来如下好处：

- 计算资源按需伸缩
- 不用负载应用混搭，集群利用率高
- 共享底层存储，避免数据跨集群迁移

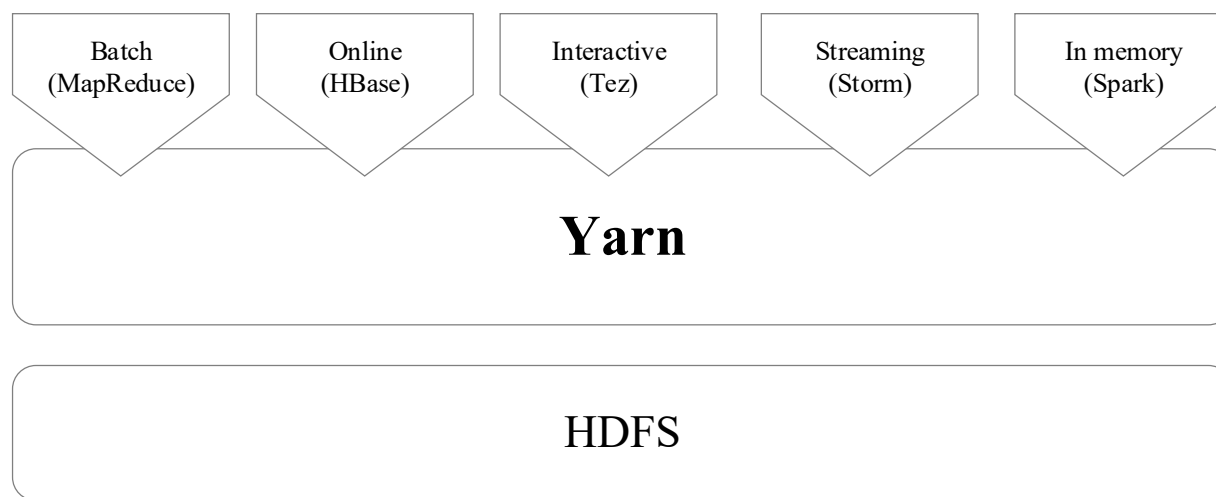
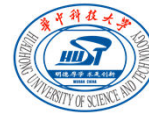


图10-20 Hadoop和Spark的统一部署



## 10.6 Spark编程实践

---

**10.6.1 Spark安装**

**10.6.2 启动Spark Shell**

**10.6.3 Spark RDD基本操作**

**10.6.4 Spark应用程序**








## 10.6.1 Spark安装

注意：安装Spark之前需要安装Java环境和Hadoop环境。

Spark下载地址：<http://spark.apache.org>

登录Linux系统（本教程统一采用hadoop用户登录），打开浏览器，访问Spark官网

（<https://archive.apache.org/dist/spark/spark-2.4.0/>）（如下图所示），在页面中选择下载“**spark-2.4.0-bin-without-hadoop.tgz**”。假设下载后的文件被保存在“~/Downloads”目录下。

	<a href="#">spark-2.4.0-bin-without-hadoop-scala-2.12.tgz</a>	2018-10-29 07:10	133M
	<a href="#">spark-2.4.0-bin-without-hadoop-scala-2.12.tgz.asc</a>	2018-10-29 07:10	819
	<a href="#">spark-2.4.0-bin-without-hadoop-scala-2.12.tgz.sha512</a>	2018-10-29 07:10	193
	<a href="#">spark-2.4.0-bin-without-hadoop.tgz</a> 	2018-10-29 07:10	153M
	<a href="#">spark-2.4.0-bin-without-hadoop.tgz.asc</a>	2018-10-29 07:10	819
	<a href="#">spark-2.4.0-bin-without-hadoop.tgz.sha512</a>	2018-10-29 07:10	288



- 解压安装包**spark-2.4.0-bin-without-hadoop.tgz**至路径 **/usr/local**:

```
$ sudo tar -zxvf ~/Downloads/spark-2.4.0-bin-without-hadoop.tgz -C /usr/local/  
  
$ cd /usr/local  
$ sudo mv ./spark-2.4.0-bin-without-hadoop/ ./spark # 更改文件夹名  
$ sudo chown -R hadoop ./spark # 此处的 hadoop 为系统用户名
```

- 配置**Spark** 的**Classpath**

```
$ cd /usr/local/spark  
$ cp ./conf/spark-env.sh.template ./conf/spark-env.sh #拷贝配置文件
```

编辑该配置文件，在文件最后面加上如下一行内容：

```
export SPARK_DIST_CLASSPATH=$(/usr/local/hadoop/bin/hadoop classpath)
```

保存配置文件后，就可以启动、运行**Spark**了。**Spark**包含多种运行模式：单机模式、伪分布式模式、完全分布式模式。本章使用单机模式运行**Spark**。若需要使用**HDFS**中的文件，则在使用**Spark**前需要启动**Hadoop**。

## 10.6.2 启动Spark Shell

- **Spark Shell** 提供了简单的方式来学习**Spark API**
- **Spark Shell**可以以实时、交互的方式来分析数据
- **Spark Shell**支持**Scala**和**Python**

本章节内容选择使用**Scala**进行编程实践，了解**Scala**有助于更好地掌握**Spark**。执行如下命令启动**Spark Shell**：

```
$ ./bin/spark-shell
```

启动**Spark Shell**成功后在输出信息的末尾可以看到“**Scala >**”的命令提示符，如下图所示。

```
Welcome to
 _ _ _ _ _
|_/_/_/_/_| version 2.4.0
|_/_/_/_/_|
|_/_/_/_/_|

Using Scala version 2.11.12 (Java HotSpot(TM) 64-Bit Server VM, Java 1.8.0_162)
Type in expressions to have them evaluated.
Type :help for more information.

scala> |
```

## 10.6.3 Spark RDD基本操作

- **Spark**的主要操作对象是**RDD**，**RDD**可以通过多种方式灵活创建，可通过导入外部数据源建立，或者从其他的**RDD**转化而来。
- 在**Spark**程序中必须创建一个**SparkContext**对象，该对象是**Spark**程序的入口，负责创建**RDD**、启动任务等。在启动**Spark Shell**后，该对象会自动创建，可以通过变量**sc**进行访问。

作为示例，我们选择以**Spark**安装目录中的“**README.md**”文件作为数据源新建一个**RDD**，代码如下：

```
Scala > val textFile = sc.textFile("file:///usr/local/spark/README.md")  
// 通过file:前缀指定读取本地文件
```

**Spark RDD**支持两种类型的操作：

动作（**action**）：在数据集上进行运算，返回计算值

转换（**transformation**）：基于现有的数据集创建一个新的数据集

**Spark**提供了非常丰富的**API**，下面两表格列出了几个常用的动作、转换**API**，更详细的**API**及说明可查阅官方文档。

表10-1常用的几个Action API介绍

Action API	说明
count()	返回数据集中的元素个数
collect()	以数组的形式返回数据集中的所有元素
first()	返回数据集中的第一个元素
take(n)	以数组的形式返回数据集中的前n个元素
reduce(func)	通过函数func（输入两个参数并返回一个值）聚合数据集中的元素
foreach(func)	将数据集中的每个元素传递到函数func中运行

表10-2 常用的几个Transformation API介绍

Transformation API	说明
filter(func)	筛选出满足函数func的元素，并返回一个新的数据集
map(func)	将每个元素传递到函数func中，并将结果返回为一个新的数据集
flatMap(func)	与map()相似，但每个输入元素都可以映射到0或多个输出结果
groupByKey()	应用于(K,V)键值对的数据集时，返回一个新的(K, Iterable<V>)形式的数据集
reduceByKey(func)	应用于(K,V)键值对的数据集时，返回一个新的(K, V)形式的数据集，其中的每个值是将每个key传递到函数func中进行聚合

- 使用**action API - count()**可以统计该文本文件的行数，命令如下：

```
Scala > textFile.count()
```

输出结果 **Long = 95**（“**Long=95**”表示该文件共有**95**行内容）。

- 使用**transformation API - filter()**可以筛选出只包含**Spark**的行，命令如下：

```
Scala > val linesWithSpark = textFile.filter(line => line.contains("Spark"))  
Scala > linesWithSpark.count()
```

第一条命令会返回一个新的**RDD**；

输出结果**Long=17**（表示该文件中共有**17**行内容包含“**Spark**”）。

也可以在同一条代码中同时使用多个**API**，连续进行运算，称为链式操作。不仅可以使**Spark**代码更加简洁，也优化了计算过程。如上述两条代码可合并为如下一行代码：

```
Scala > val linesCountWithSpark  
= textFile.filter(line => line.contains("Spark")).count()
```

假设我们只需要得到包含“**Spark**”的行数，那么存储筛选后的文本数据是多余的，因为这部分数据在计算得到行数后就不再使用到了。**Spark**基于整个操作链，仅储存、计算所需的数据，提升了运行效率。

**Spark**属于**MapReduce**计算模型，因此也可以实现**MapReduce**的计算流程，如实现单词统计，可以使用如下的命令实现：

```
Scala > val wordCounts = textFile.flatMap(line => line.split(" ")).map(word =>
(word, 1)).reduceByKey((a, b) => a + b)
Scala > wordCounts.collect() // 输出单词统计结果
// Array[(String, Int)] = Array((package,1), (For,2), (Programs,1), (processing.,1),
(Because,1), (The,1)...)

```

- 首先使用**flatMap()**将每一行的文本内容通过空格进行划分为单词；
- 再使用**map()**将单词映射为(**K,V**)的键值对，其中**K**为单词，**V**为**1**；
- 最后使用**reduceByKey()**将相同单词的计数进行相加，最终得到该单词总的出现的次数。

输出单词统计结果： ((package,1), (For,2), (Programs,1), (processing.,1), (Because,1), (The,1)...) 。

## 10.6.4 Spark 应用程序

- 在**Spark Shell**中进行编程主要是方便对代码进行调试，但需要以逐行代码的方式运行。一般情况下，会选择将调试后代码打包成独立的**Spark**应用程序，提交到**Spark**中运行。

采用**Scala**编写的程序需要使用**sbt**（**Simple Build Tool**）进行打包，**sbt**的安装配置步骤如下：

- 下载**sbt**安装包文件**sbt-1.3.8.tgz**（下载地址：<http://www.scala-sbt.org>）

将下载后的文件拷贝至安装目录**/usr/local/sbt**中，命令如下：

```
$ sudo mkdir /usr/local/sbt           # 创建安装目录
$ cd ~/Downloads
$ sudo tar -zxvf ./sbt-1.3.8.tgz -C /usr/local
$ cd /usr/local/sbt
$ sudo chown -R hadoop /usr/local/sbt   # 此处的hadoop为系统当前用户名
$ cp ./bin/sbt-launch.jar ./          #把bin目录下的sbt-launch.jar复制到sbt安装目录下
```

- 接着在安装目录中使用下面命令创建一个**Shell**脚本文件，用于启动**sbt**:

```
$ vim /usr/local/sbt/sbt
```

该脚本文件中的代码如下:

```
#!/bin/bash
SBT_OPTS="-Xms512M -Xmx1536M -Xss1M -XX:+CMSClassUnloadingEnabled -
XX:MaxPermSize=256M"
java $SBT_OPTS -jar `dirname $0`/sbt-launch.jar "$@"
```

保存后，还需要为该**Shell**脚本文件增加可执行权限:

```
chmod u+x /usr/local/sbt/sbt
```



然后，可以使用如下命令查看**sbt**版本信息：

```
$ cd /usr/local/sbt
$ ./sbt sbtVersion
Java HotSpot(TM) 64-Bit Server VM warning: ignoring option MaxPermSize=256M;
support was removed in 8.0
[warn] No sbt.version set in project/build.properties, base directory: /usr/local/sbt
[info] Set current project to sbt (in build file:/usr/local/sbt/)
[info] 1.3.8
```

上述查看版本信息的命令，可能需要执行几分钟，执行成功以后就可以看到版本为**1.3.8**。

我们以一个简单的程序为例，介绍如何打包并运行**Spark**程序，该程序的功能是统计文本文件中包含字母**a**和字**b**的各有多少行，具体步骤如下：

1. 创建程序根目录，并创建程序所需的文件夹结构，命令如下：

```
mkdir ~/sparkapp          # 创建程序根目录
mkdir -p ~/sparkapp/src/main/scala # 创建程序所需的文件夹结构
```

2. 创建一个**SimpleApp.scala**文件（文件路径：**~/sparkapp/src/main/scala/SimpleApp.scala**），文件中的代码如下：

```
import org.apache.spark.SparkContext
import org.apache.spark.SparkContext._
import org.apache.spark.SparkConf

object SimpleApp {
  def main(args: Array[String]) {
    val logFile = "file:///usr/local/spark/README.md" // 用于统计的文本文件
    val conf = new SparkConf().setAppName("Simple Application")
    val sc = new SparkContext(conf)
    val logData = sc.textFile(logFile, 2).cache()
    val numAs = logData.filter(line => line.contains("a")).count()
    val numBs = logData.filter(line => line.contains("b")).count()
    println("Lines with a: %s, Lines with b: %s".format(numAs, numBs))
  }
}
```

3. 然后创建一个**simple.sbt**文件（文件路径：**~/sparkapp/simple.sbt**），用于声明该应用程序的信息以及与**Spark**的依赖关系，具体内容如下：

```
name := "Simple Project"
version := "1.0"
scalaVersion := "2.11.12"
libraryDependencies += "org.apache.spark" %% "spark-core" % "2.4.0"
```

4. 使用**sbt**对该应用程序进行打包，命令如下：

```
cd ~/sparkapp
/usr/local/sbt/sbt package
```

打包成功后，会输出程序**jar**包的位置以及“**Done Packaging**”的提示，如下所示。

```
$~/sparkapp$ /usr/local/sbt/sbt package
[info] Set current project to Simple Project
[info] Updating {file:/home/hadoop/sparkapp/}sparkapp...
[info] Done updating.
[info] Compiling 1 Scala source to /home/hadoop/sparkapp/target/...
[info] Packaging /home/hadoop/sparkapp/target/scala-2.11/...
[info] Done packaging.
[success] Total time: 17 s, completed 2020-1-27 16:13:56
```

有了最终生成的jar包后，再通过**spark-submit**就可以提交到**Spark**中运行了，命令如下：

```
/usr/local/spark/bin/spark-submit --class "SimpleApp" ~/sparkapp/target/scala-2.11/simple-project_2.11-1.0.jar
```

该应用程序的执行结果如下：

Lines with a: 62, Lines with b: 31

## 本章小结

- 本章首先介绍了**Spark**的起源与发展，分析了**Hadoop**存在的缺点与**Spark**的优势。接着介绍了**Spark**的相关概念、生态系统与核心设计。
- **Spark**的核心是统一的抽象**RDD**，在此之上形成了结构一体化、功能多元化的完整的大数据生态系统，支持内存计算，**SQL**即席查询、实时流式计算、机器学习和图计算。
- 本章最后介绍了**Spark**部署与应用，包括**Spark**的安装与**Spark Shell**的使用，并演示了**Spark RDD**的基本操作。**Spark**提供了丰富的**API**，让开发人员可以用简洁的方式来处理复杂的数据计算与分析。