



# 《大数据技术及应用》

## 第九章 Spark

# 提纲

**9.1 Spark概述**

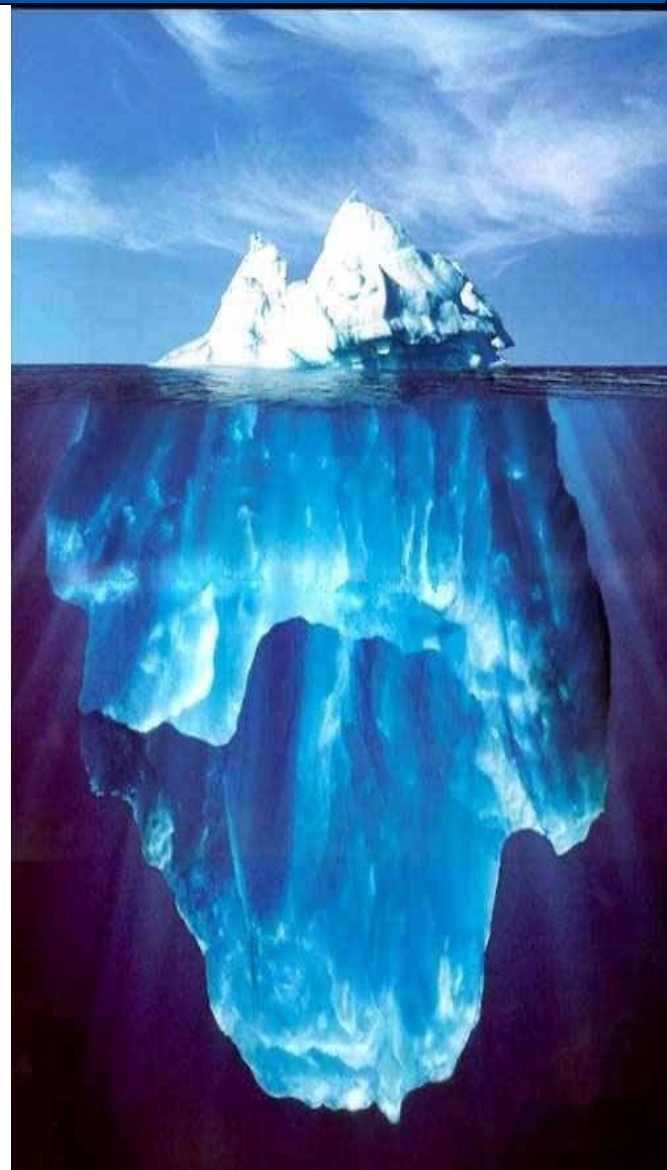
**9.2 Spark生态系统**

**9.3 Spark运行架构**

**9.4 Spark SQL**

**9.5 Spark的部署和应用方式**

**9.6 Spark编程实践**

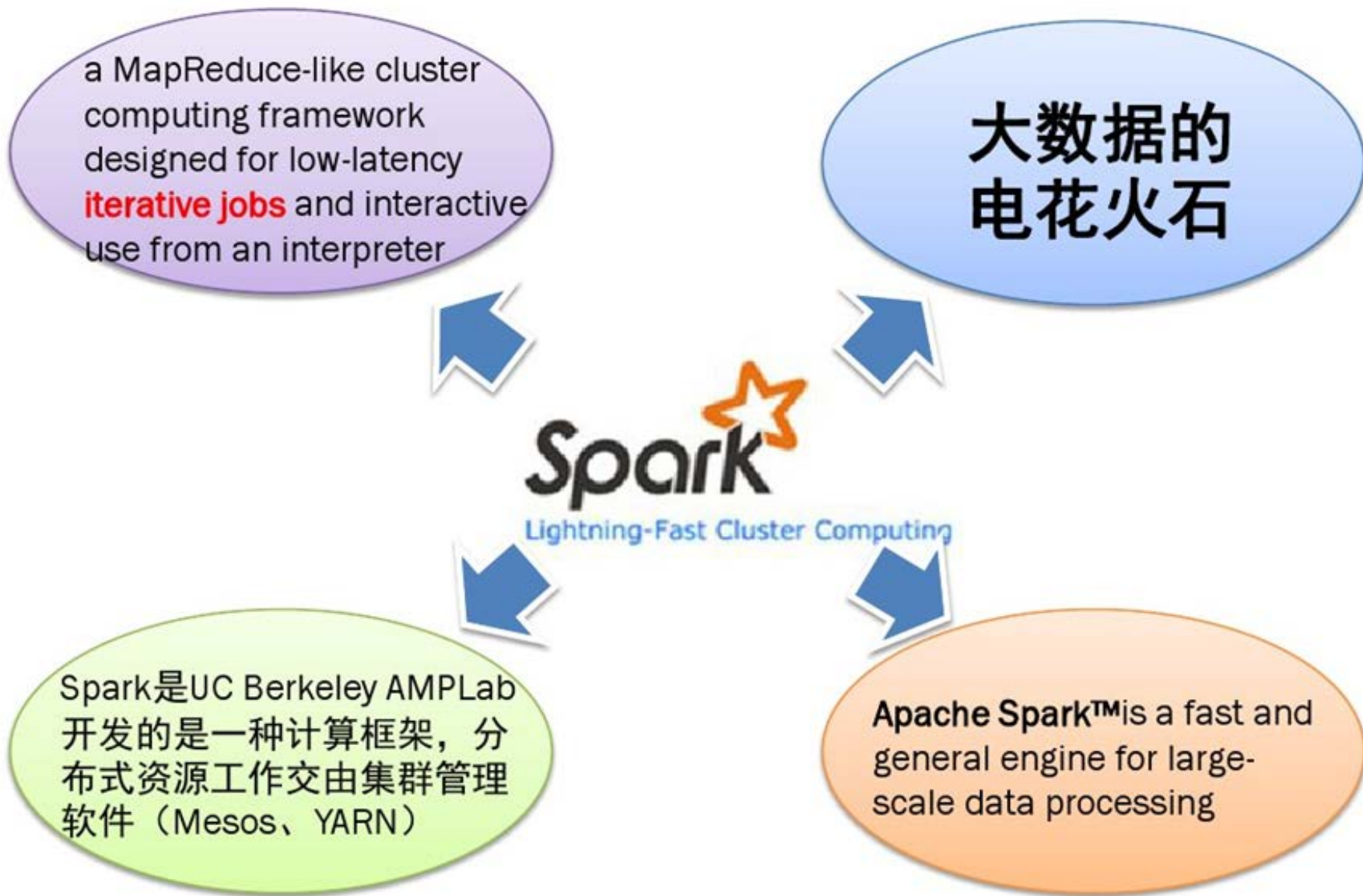


# 9.1 Spark概述

9.1.1 Spark简介

9.1.2 Scala简介

9.1.3 Spark与Hadoop的比较



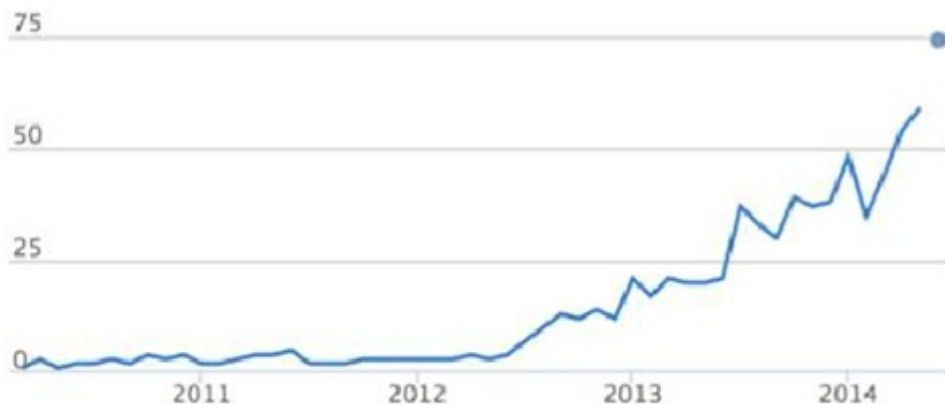
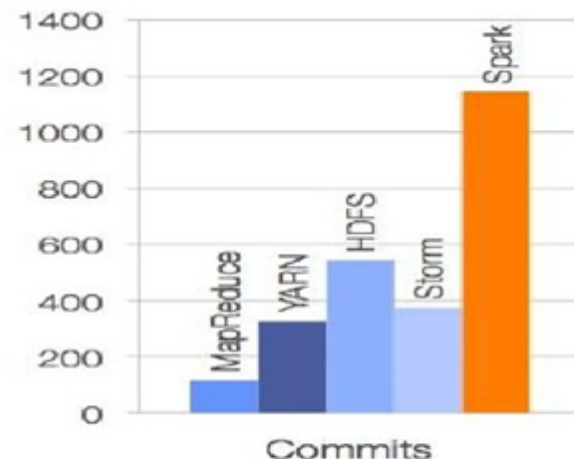
## 9.1.1 Spark简介

- **Spark**最初由美国加州伯克利大学（**UC Berkeley**）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，获得比**Hadoop**快3倍的速度

## 9.1.1 Spark简介

- 2009：Spark诞生于伯克利大学 AMPLab
- 2010：开源
- 2013.6：Apache孵化器项目
- 2014.2：Apache顶级项目
- 目前为止，发布的最新版本为Spark1.4.1

Spark在最近6年内发展迅速，相较于其他大数据平台或框架而言，Spark的代码库最为活跃。



Spark代码贡献者每个月的增长曲线

截止2015年6月

- Spark的Contributor比2014年涨了3倍，达到730人；
- 总代码行数也比2014年涨了2倍多，达到40万行
- Spark应用也越来越广泛，最大的集群来自腾讯——8000个节点，单个Job最大分别是阿里巴巴和Databricks——1PB



### 先进架构

- Spark采用Scala语言编写，底层采用了actor model的akka作为通讯框架，代码十分简洁高效。
- 基于DAG图的执行引擎，减少多次计算之间中间结果写到Hdfs的开销。
- 建立在统一抽象的RDD（分布式内存抽象）之上，使得它可以以基本一致的方式应对不同的大数据处理场景。

### 高效

- 提供Cache机制来支持需要反复迭代的计算或者多次数据共享，减少数据读取的IO开销。
- 与Hadoop的MapReduce相比，Spark基于内存的运算比MR要快100倍；而基于硬盘的运算也要快10倍！

### 易用

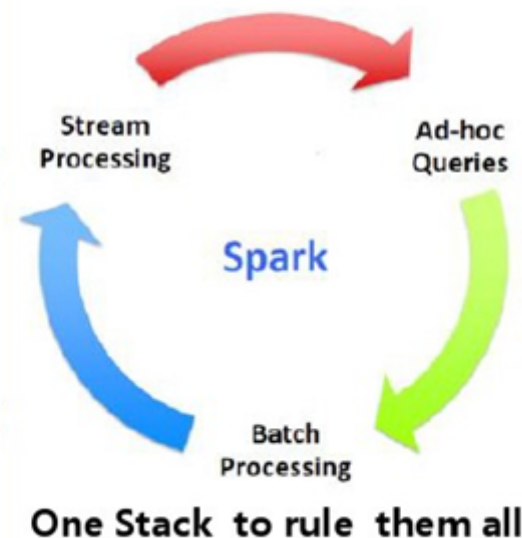
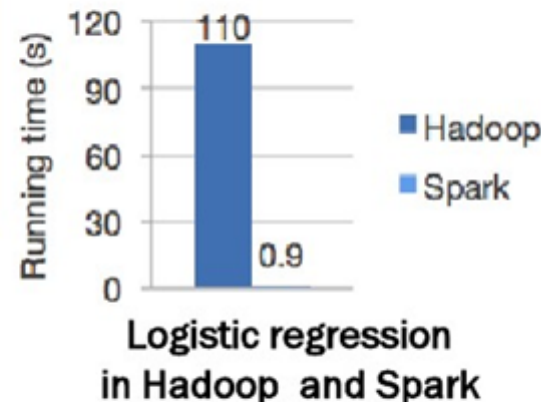
- Spark提供广泛的数据集操作类型（20+种），不像Hadoop只提供了Map和Reduce两种操作。
- Spark支持Java，Python和Scala API，支持交互式的Python和Scala的shell。

### 提供整体解决方案

- 以其RDD模型的强大表现能力，逐渐形成了一套自己的生态圈，提供了full-stack的解决方案。
- 主要包括Spark内存中批处理，Spark SQL交互式查询，Spark Streaming流式计算，GraphX和MLlib提供的常用图计算和机器学习算法。

### 与Hadoop无缝衔接

- Spark可以使用YARN作为它的集群管理器
- 读取HDFS,HBase等一切Hadoop的数据



## 9.1.1 Spark简介

**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

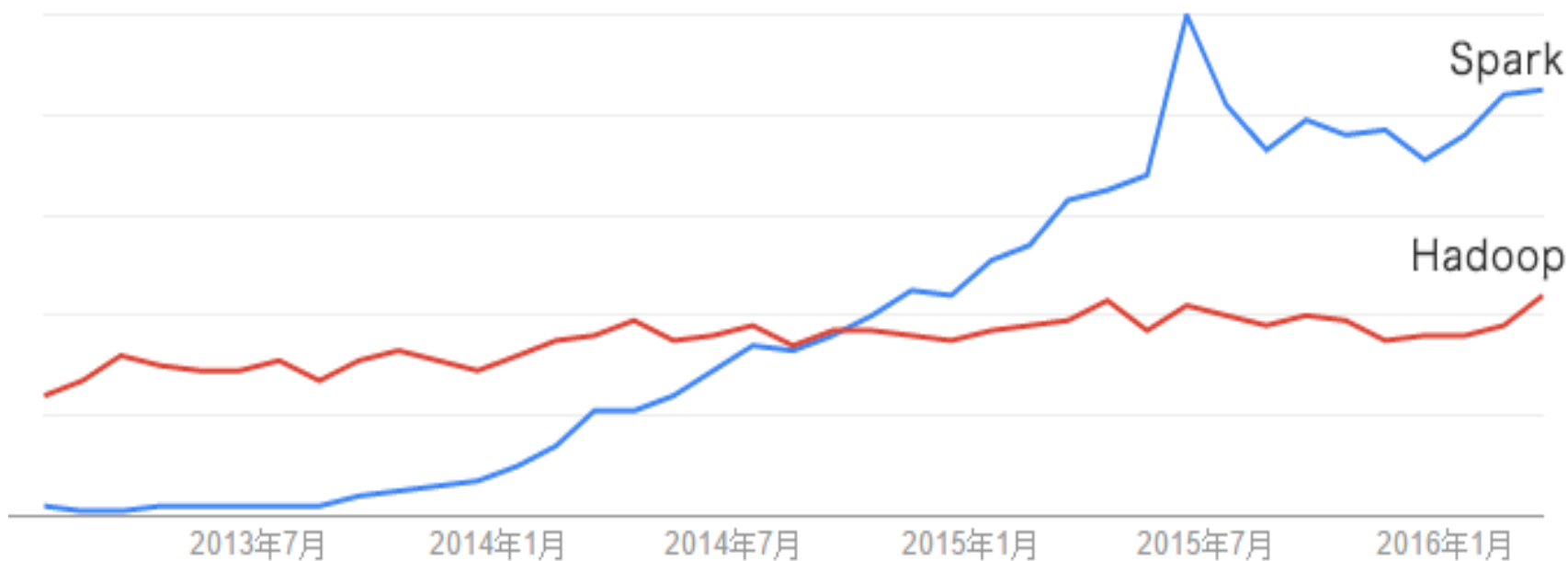
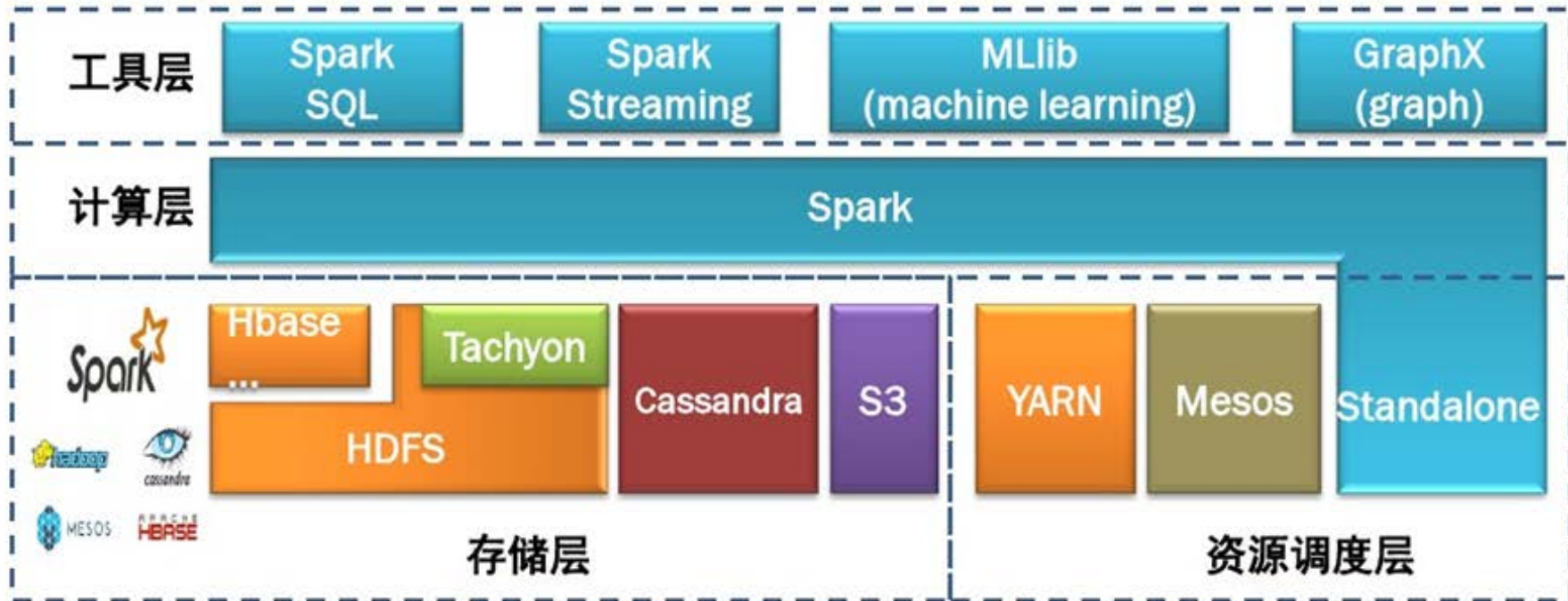


图9-1 谷歌趋势：Spark与Hadoop对比





- Spark提供了多种高级工具：Spark SQL应用于即席查询 ( Ad-hoc query )、Spark Streaming应用于流式计算、MLlib应用于机器学习、GraphX应用于图处理。
- Spark可以基于自带的standalone集群管理器独立运行，也可以部署在Apache Mesos 和 Hadoop YARN 等集群管理器上运行。
- Spark可以访问存储在HDFS、Hbase、Cassandra、Amazon S3、本地文件系统等等上的数据，Spark支持文本文件，序列文件，以及任何Hadoop的InputFormat。

# SPARK的开发语言

- Spark是由Scala语言开发的
- 在开发Spark应用程序的时候多数会选择Scala语言
- 可以直接用Spark的Java API或Python API
- 熟悉Scala之后再看Java代码，有种读汇编的感觉.....
- 入门难度大

## 9.1.2 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

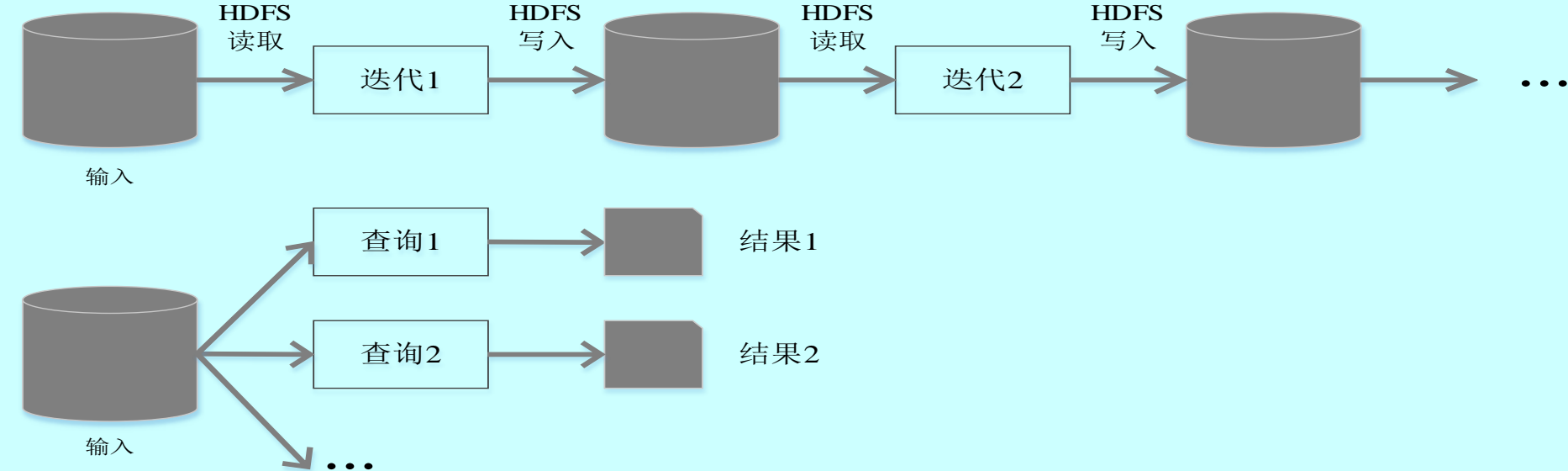
**Scala**是**Spark**的主要编程语言，但**Spark**还支持**Java**、**Python**、**R**作为编程语言；**Scala**的优势是提供了**REPL**（**Read-Eval-Print Loop**，交互式解释器），提高程序开发效率

## 9.1.3 Spark与Hadoop的对比

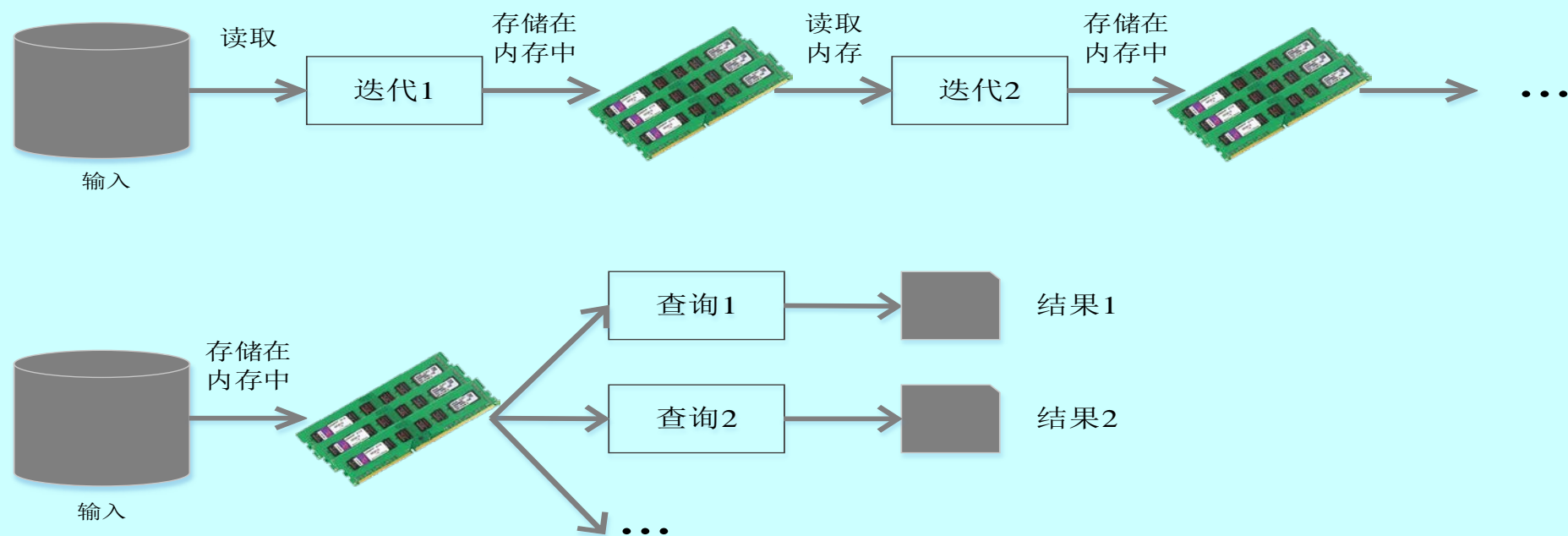
**Spark**在借鉴**Hadoop MapReduce**优点的同时，很好地解决了**MapReduce**所面临的问题

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

## 9.1.3 Spark与Hadoop的对比

- 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

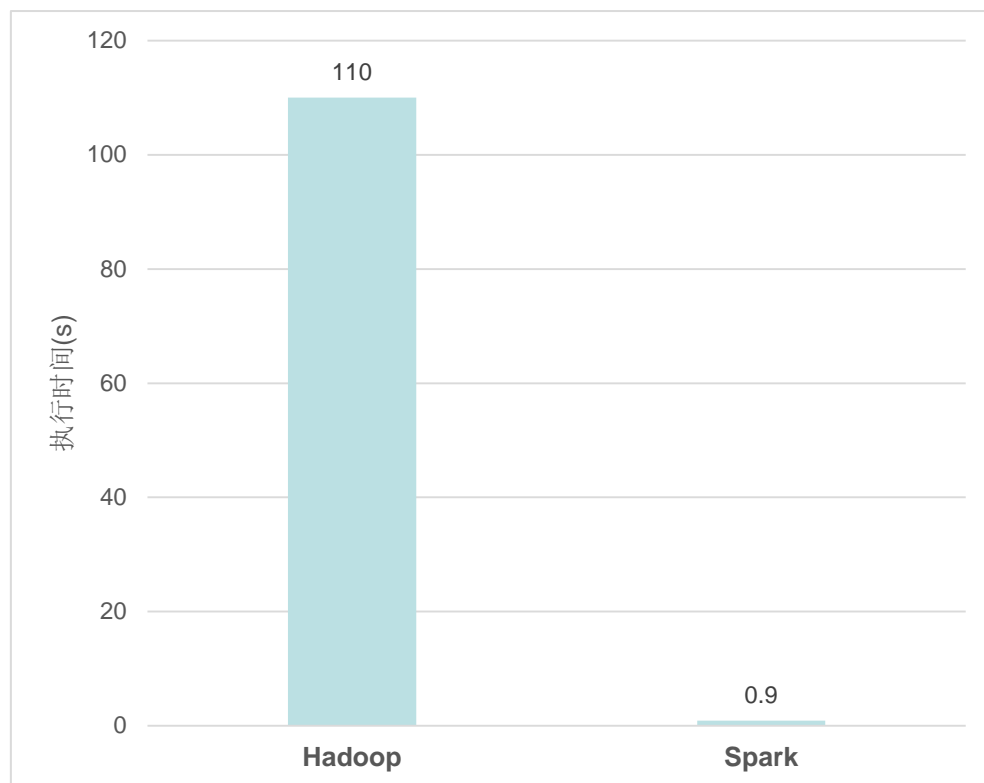


图9-3 Hadoop与Spark执行逻辑回归的时间对比



## 9.2 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：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：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/出数据无法无缝共享，需要数据格式转换
- 不同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

## 9.2 Spark生态系统

- **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

## 9.2 Spark生态系统

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

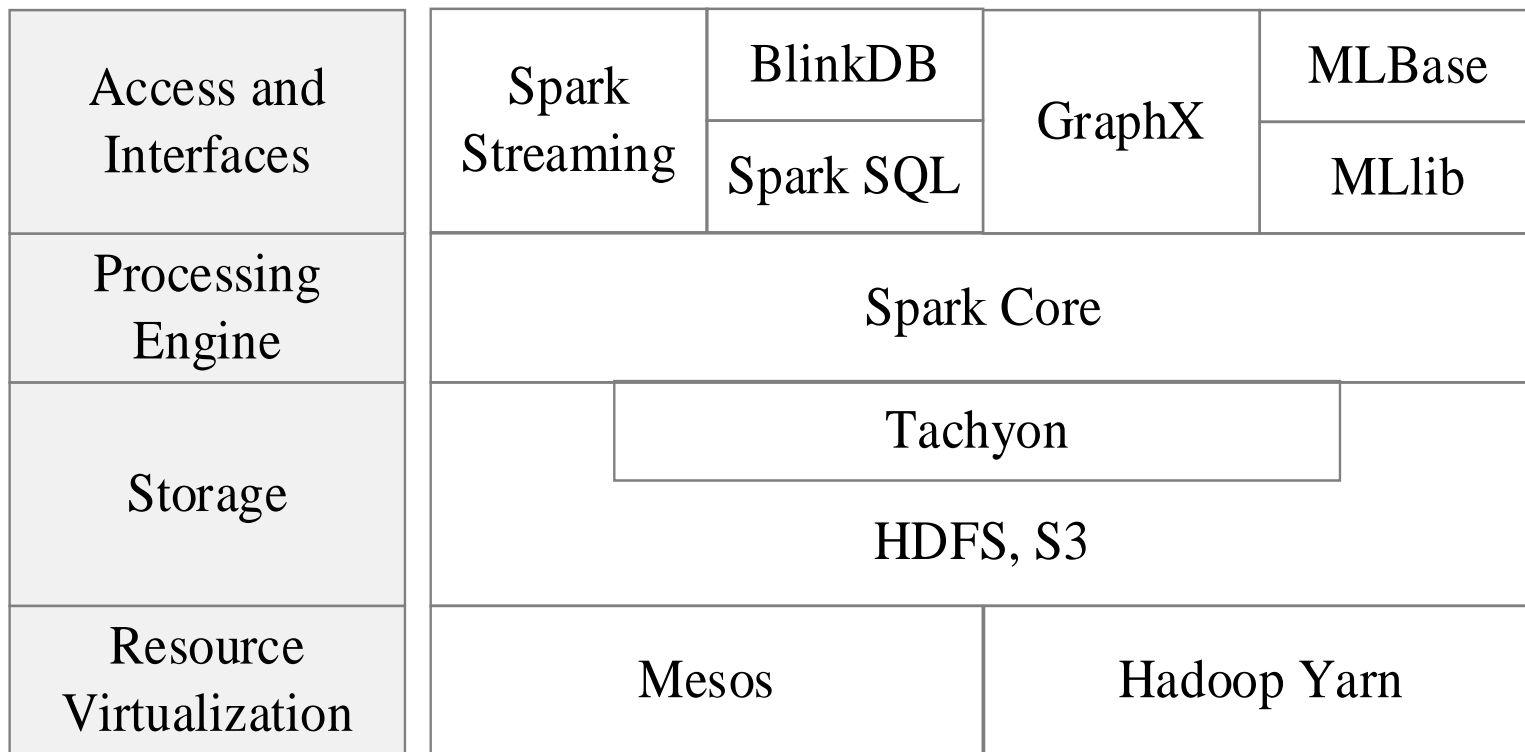


图9-4 BDAS架构

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

## 9.2 Spark生态系统

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

## 9.3 Spark运行架构

### 9.3.1 基本概念

### 9.3.2 架构设计

### 9.3.3 Spark运行基本流程

### 9.3.4 Spark运行原理

## 9.3.1 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



## 9.3.2 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

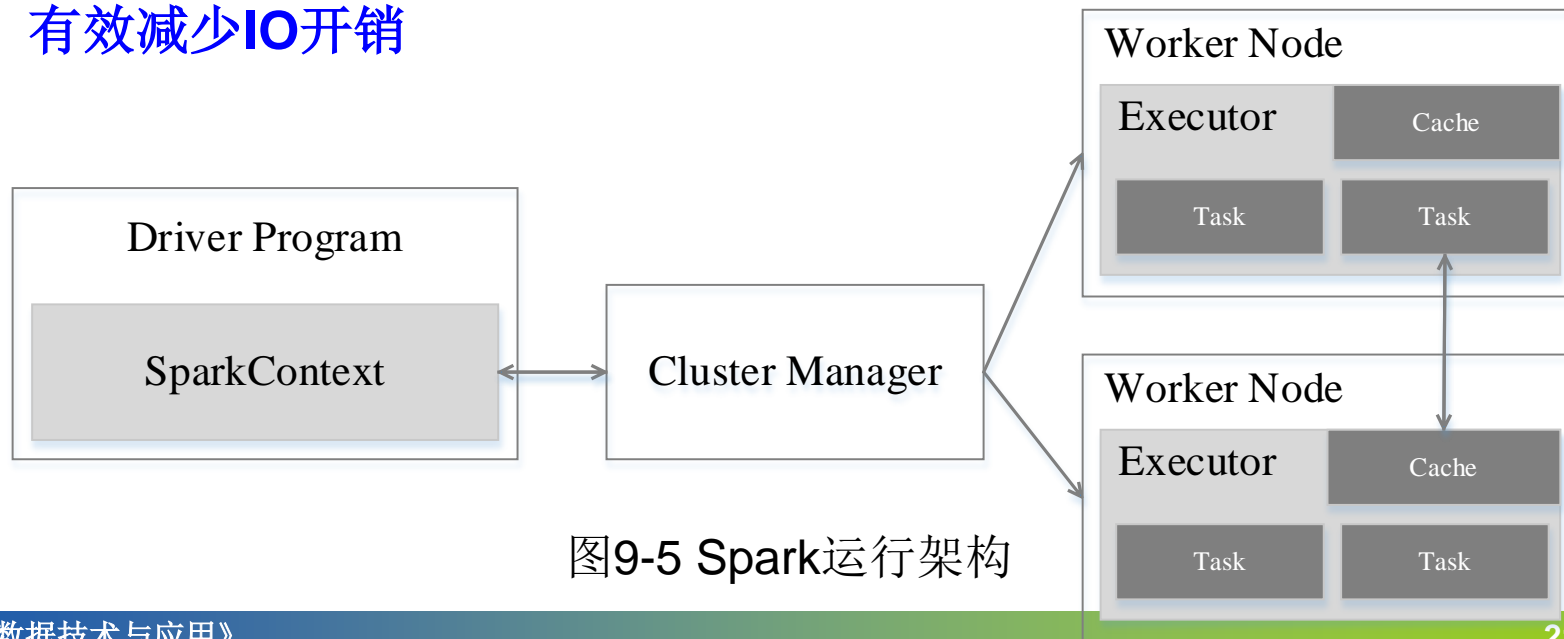
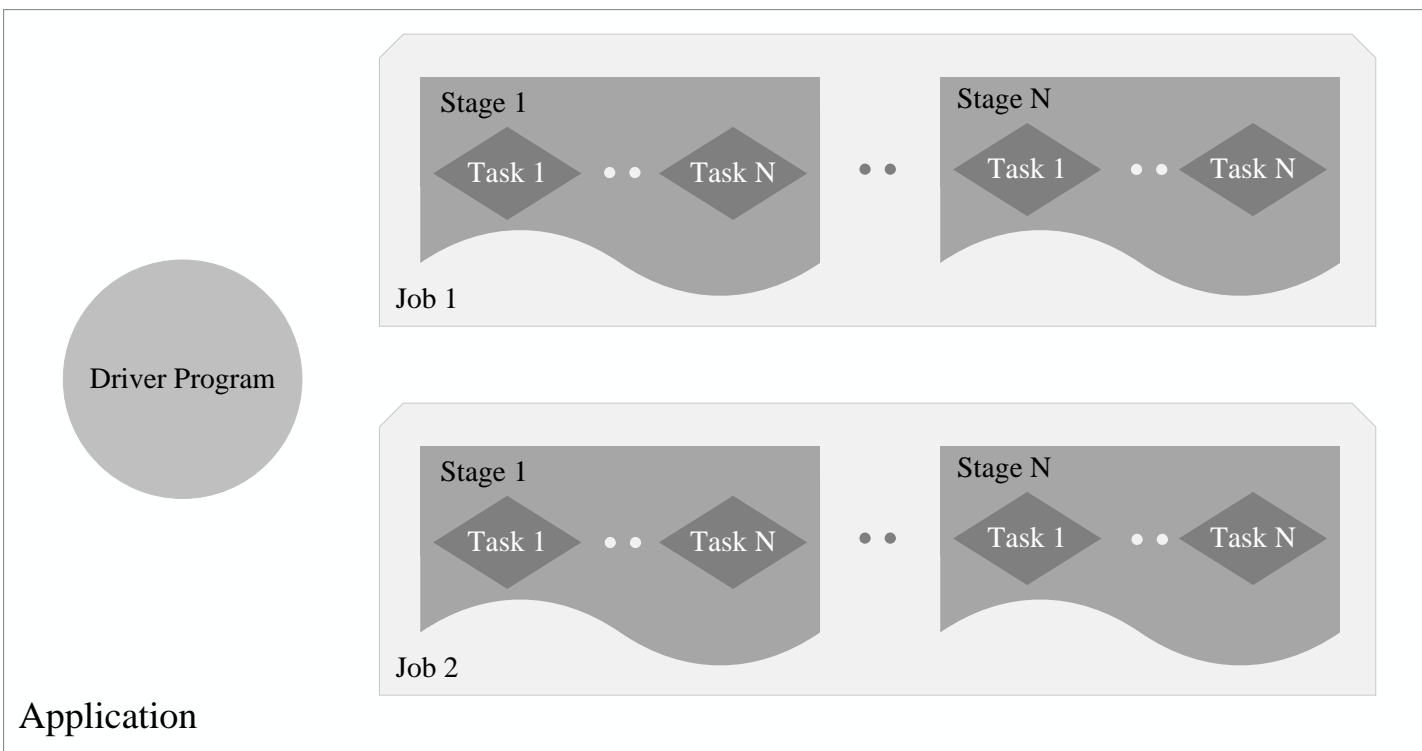


图9-5 Spark运行架构

## 9.3.2 架构设计

- 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



## 9.3.3 Spark运行基本流程

(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

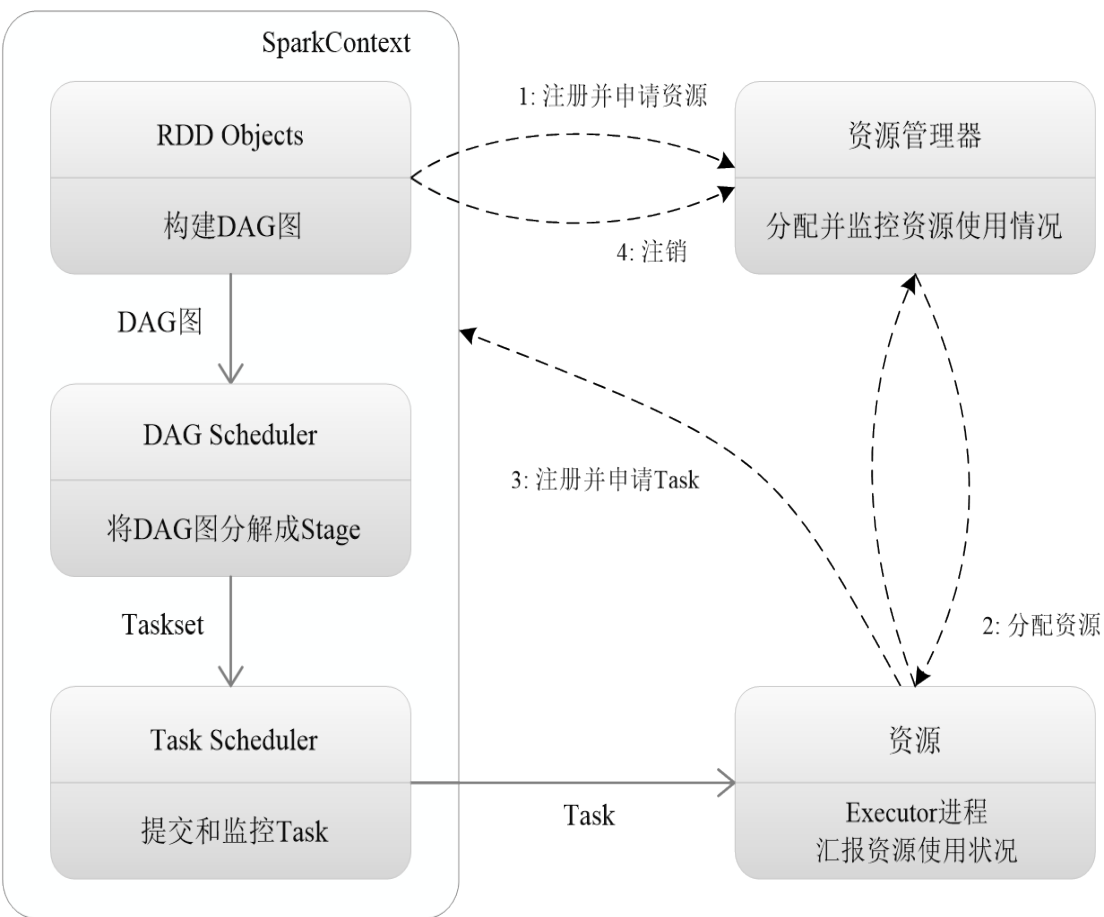


图9-7 Spark运行基本流程图

(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

## 9.3.3 Spark运行基本流程

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

## 9.3.4 RDD运行原理

**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

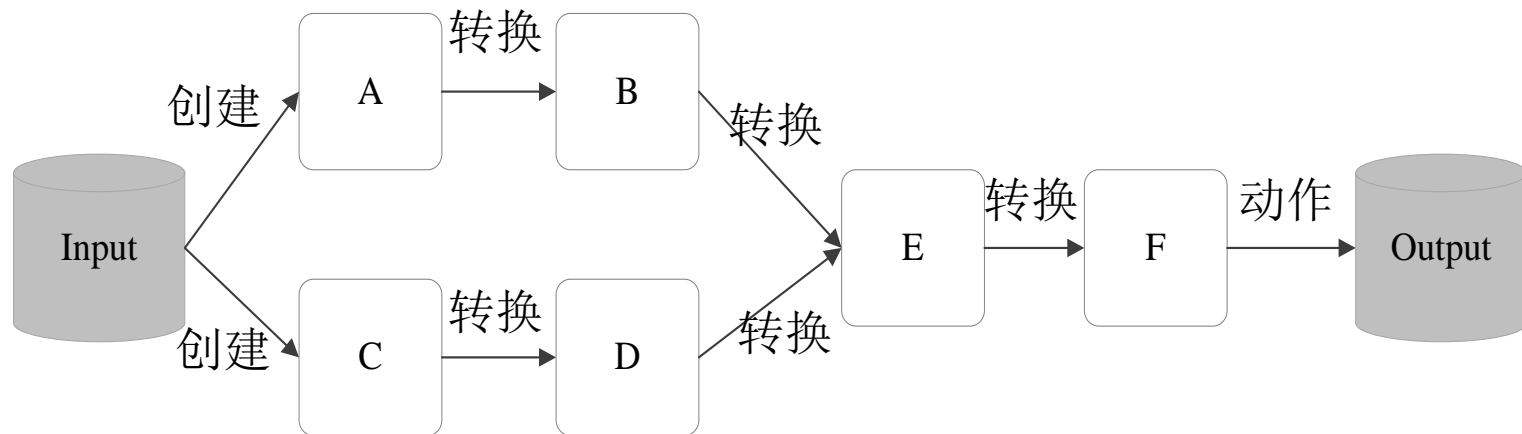
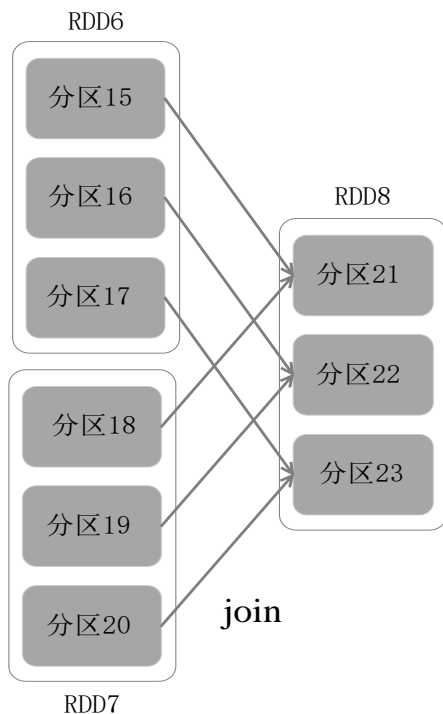
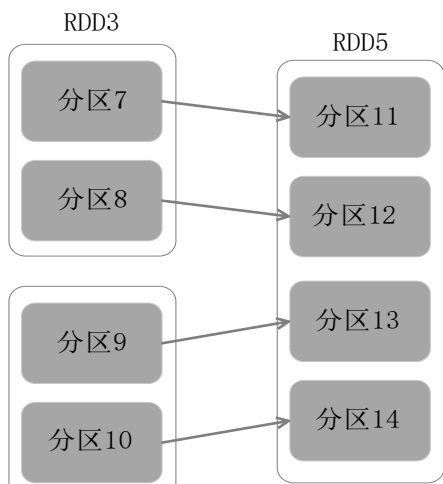
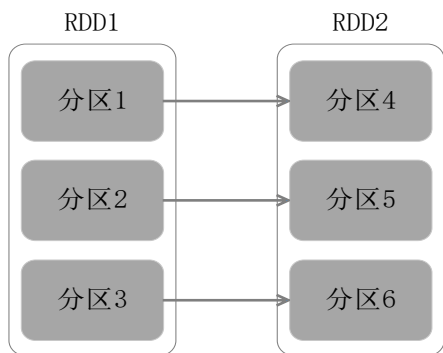


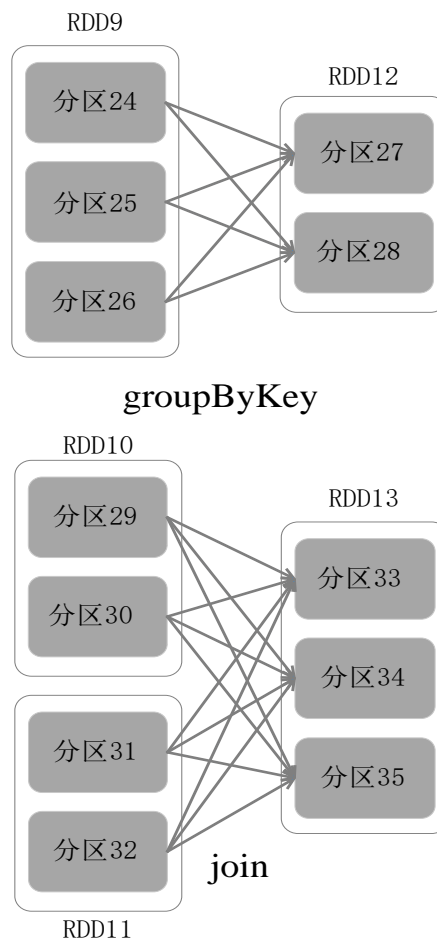
图9-8 RDD执行过程的一个实例

## 9.3.4 RDD运行原理

### RDD之间的依赖关系



(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9-9 窄依赖与宽依赖的区别



## 9.3.4 RDD运行原理

### Stage的划分

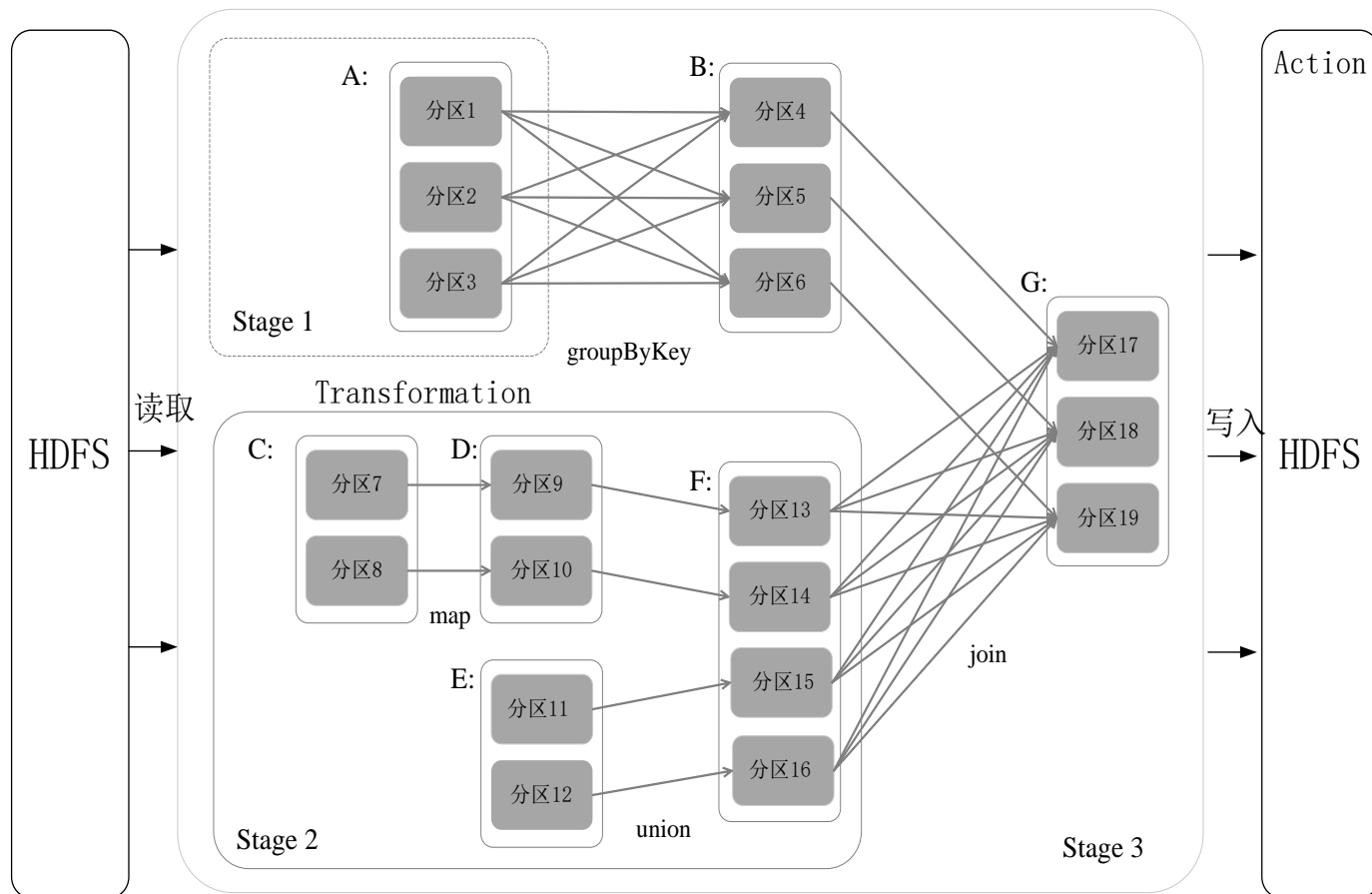
**Spark**通过分析各个**RDD**的依赖关系生成了**DAG**，再通过分析各个**RDD**中的分区之间的依赖关系来决定如何划分**Stage**，具体划分方法是：

- 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

## 9.3.4 RDD运行原理

### Stage的划分

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9-10根据RDD分区的依赖关系划分Stage

## 9.3.4 RDD运行原理

### Stage的划分

**Stage**的类型包括两种：**ShuffleMapStage**和**ResultStage**，具体如下：

(1) **ShuffleMapStage**：不是最终的**Stage**，在它之后还有其他**Stage**，所以，它的输出一定需要经过**Shuffle**过程，并作为后续**Stage**的输入；这种**Stage**是以**Shuffle**为输出边界，其输入边界可以从外部获取数据，也可以是另一个**ShuffleMapStage**的输出，其输出可以是另一个**Stage**的开始；在一个**Job**里可能有该类型的**Stage**，也可能没有该类型**Stage**；

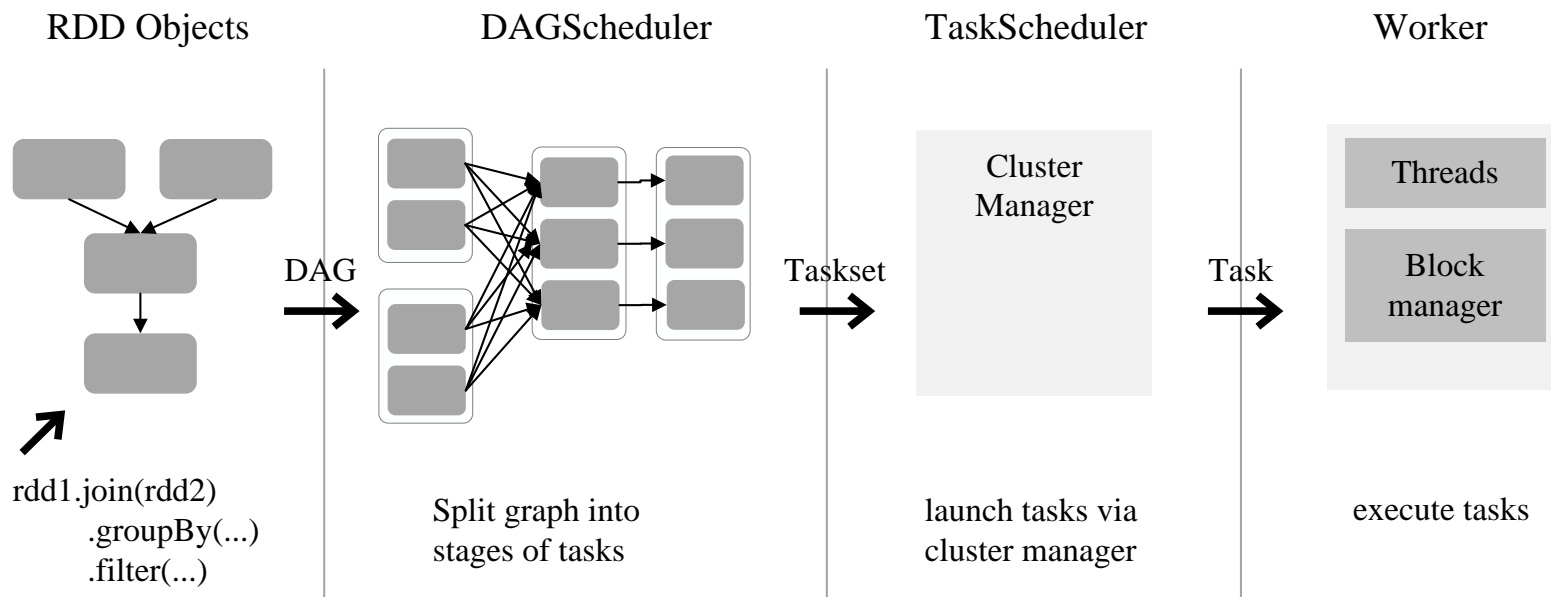
(2) **ResultStage**：最终的**Stage**，没有输出，而是直接产生结果或存储。这种**Stage**是直接输出结果，其输入边界可以从外部获取数据，也可以是另一个**ShuffleMapStage**的输出。在一个**Job**里必定有该类型**Stage**。

因此，一个**Job**含有一个或多个**Stage**，其中至少含有一个**ResultStage**。

## 9.3.4 RDD运行原理

**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。



## 9.4 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

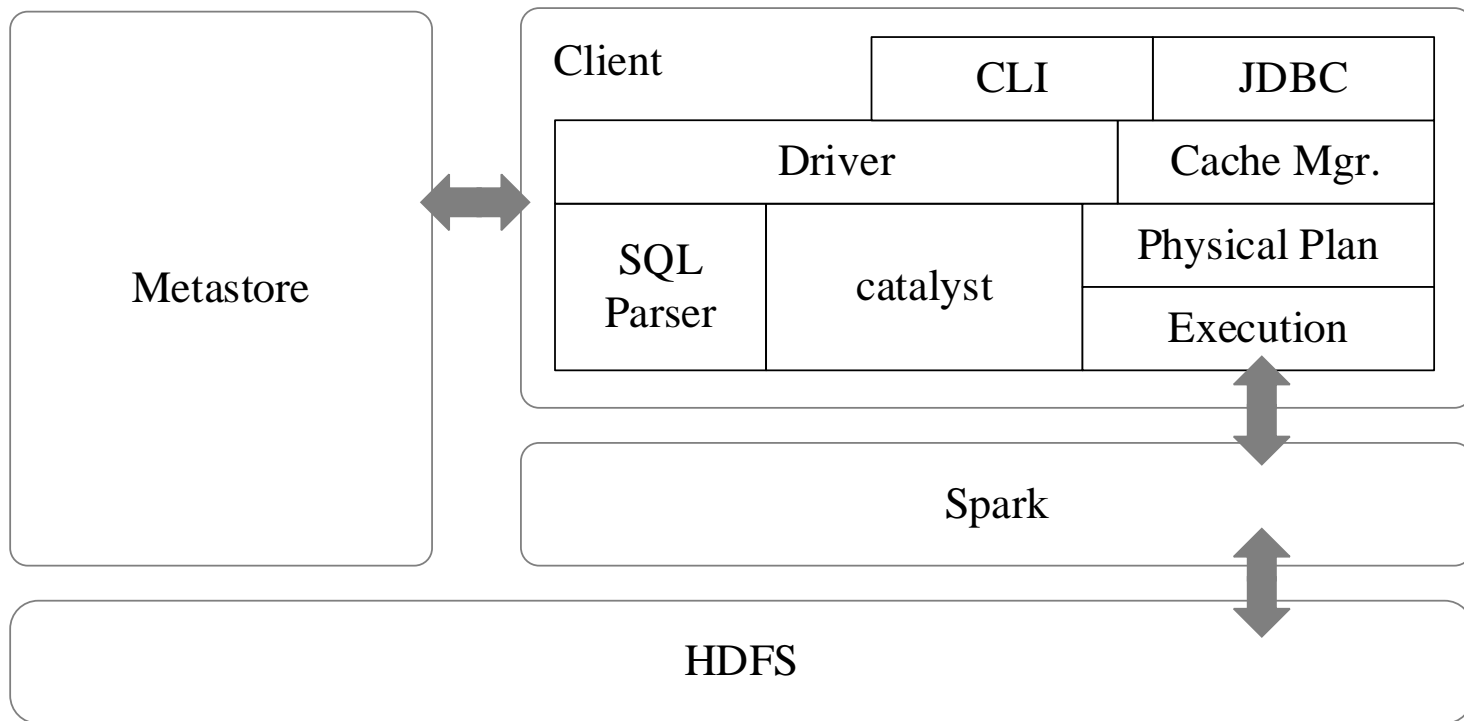


图9-12 Spark SQL架构

## 9.4 Spark SQL设计

- 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

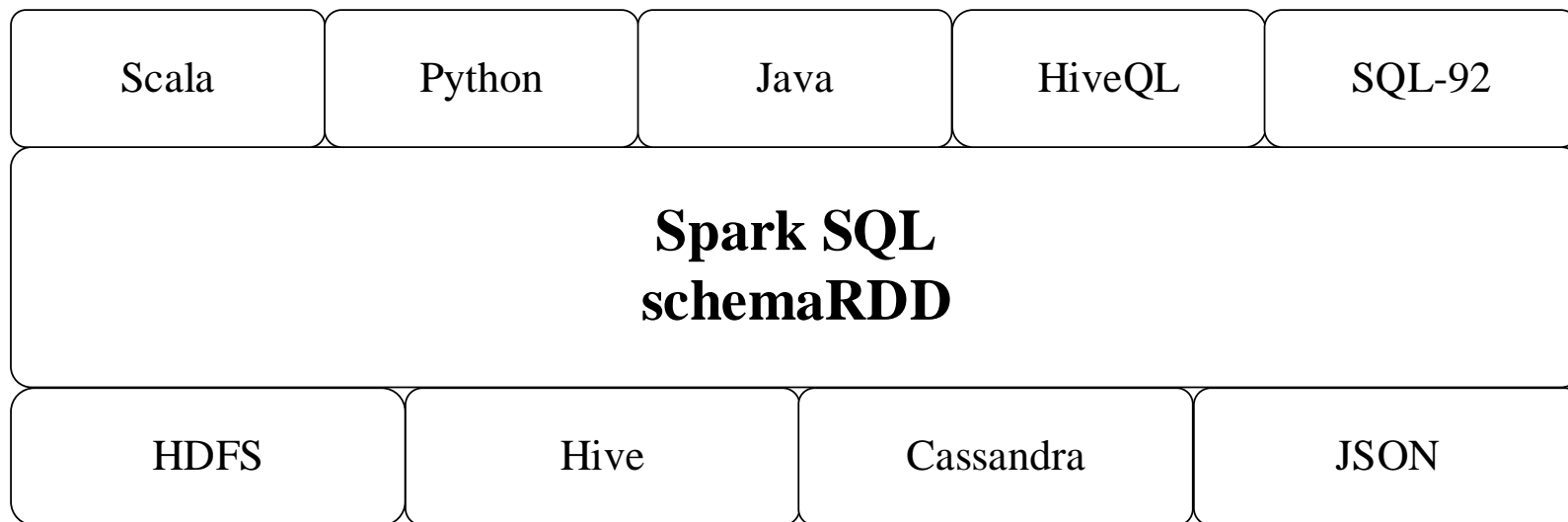


图9-13 Spark SQL支持的数据格式和编程语言



## 9.5 Spark的部署和应用方式

### 9.5.1 Spark三种部署方式

### 9.5.2 从Hadoop+Storm架构转向Spark架构

### 9.5.3 Hadoop和Spark的统一部署

## 9.5.1 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

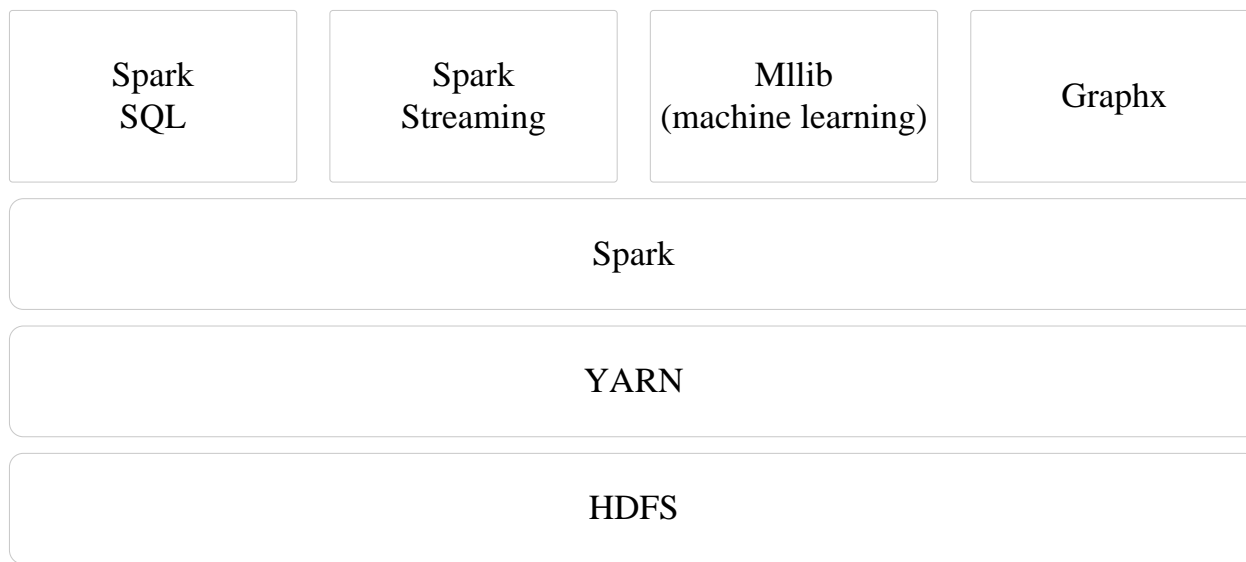


图9-17 Spark on Yarn架构

## 9.5.2 从Hadoop+Storm架构转向Spark架构

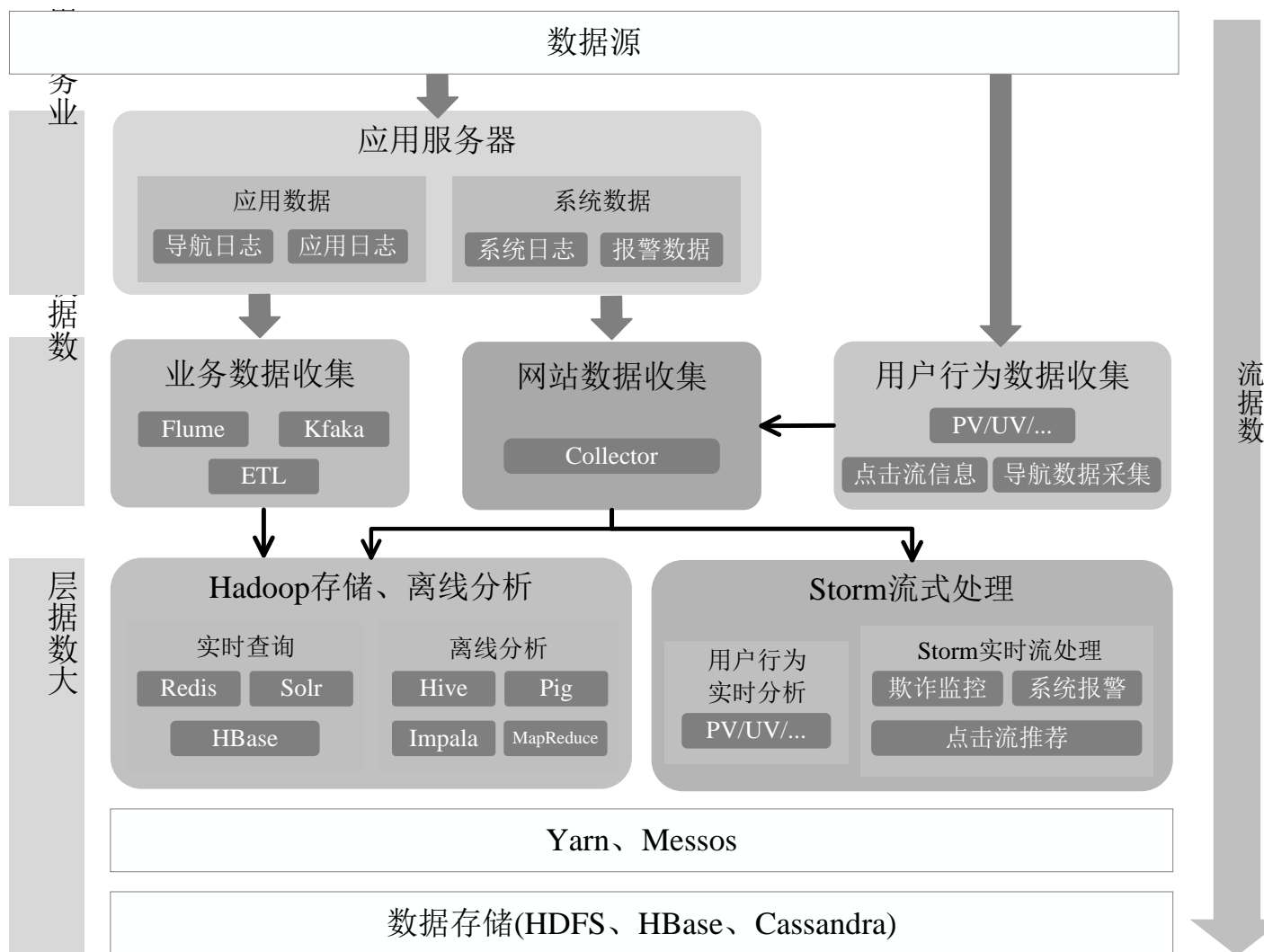


图9-18 采用Hadoop+Storm部署方式的一个案例

这种架构部署较为繁琐

## 9.5.2 从Hadoop+Storm架构转向Spark架构

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**）

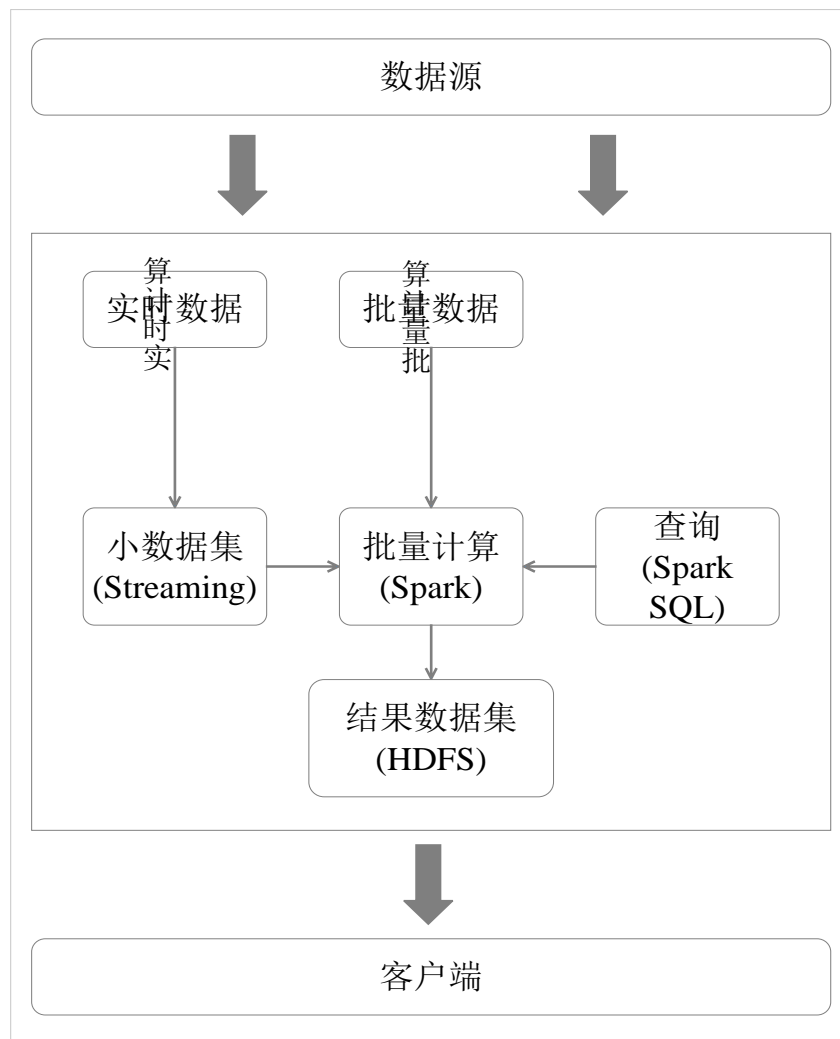


图9-19 用Spark架构满足批处理和流处理需求

## 9.5.3 Hadoop和Spark的统一部署

- 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

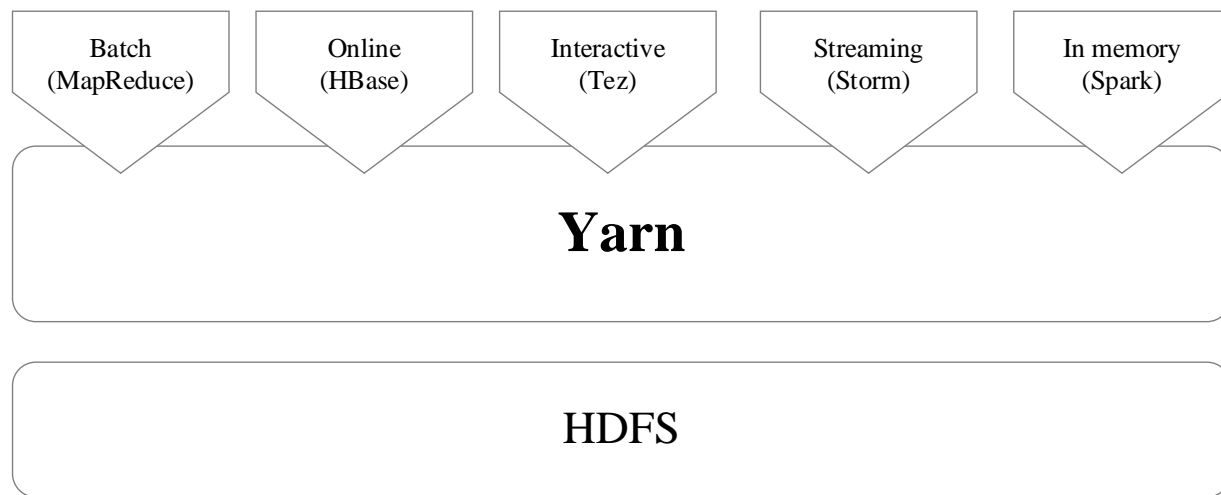


图9-20 Hadoop和Spark的统一部署

## 9.6 Spark编程实践

### 9.6.1 Spark安装

### 9.6.2 启动Spark Shell

### 9.6.3 Spark RDD基本操作

### 9.6.4 Spark应用程序

## 9.6.1 Spark安装

安装**Spark**之前需要安装**Java**环境和**Hadoop**环境。

•下载地址：<http://spark.apache.org>

进入下载页面后，点击主页右侧的“**Download Spark**”按钮进入下载页面，下载页面中提供了几个下载选项，主要是**Spark release**及**Package type**的选择，如下图所示。第1项**Spark release**一般默认选择最新的发行版本，如截止至**2010年3月份**的最新版本为**1.6.0**。第2项**package type**则选择“**Pre-build with user-provided Hadoop [can use with most Hadoop distributions]**”，可适用于多数**Hadoop**版本。选择好之后，再点击第4项给出的链接就可以下载**Spark**了。

### Download Spark

The latest release of Spark is Spark 1.6.0, released on January 4, 2016 ([release notes](#)) ([git tag](#))

1. Choose a Spark release:
2. Choose a package type:
3. Choose a download type:
4. Download Spark: [spark-1.6.0-bin-without-hadoop.tgz](#)
5. Verify this release using the [1.6.0 signatures and checksums](#).

*Note: Scala 2.11 users should download the Spark source package and build with Scala 2.11 support.*

## 9.6.1 Spark安装

- 解压安装包**spark-1.6.0-bin-without-hadoop.tgz**至路径 **/usr/local**:

```
$ sudo tar -zxvf ~/下载/spark-1.6.0-bin-without-hadoop.tgz -C /usr/local/
```

```
$ cd /usr/local
```

```
$ sudo mv ./spark-1.6.0-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**。



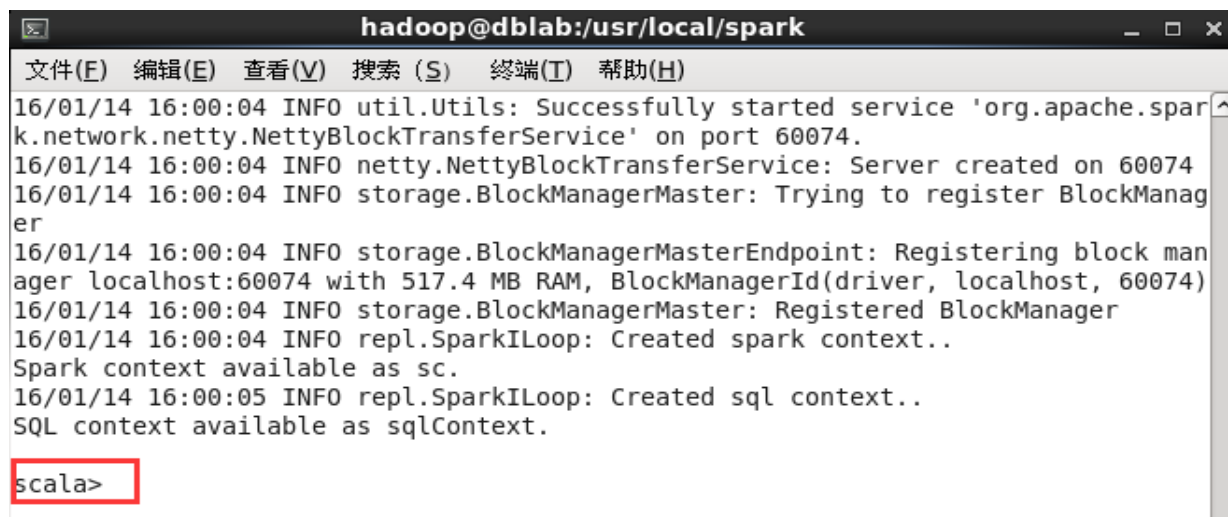
## 9.6.2启动Spark Shell

- **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**

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。



```
hadoop@dblab:/usr/local/spark
文件(E) 编辑(E) 查看(V) 搜索(S) 终端(T) 帮助(H)
16/01/14 16:00:04 INFO util.Utils: Successfully started service 'org.apache.spark.network.netty.NettyBlockTransferService' on port 60074.
16/01/14 16:00:04 INFO netty.NettyBlockTransferService: Server created on 60074
16/01/14 16:00:04 INFO storage.BlockManagerMaster: Trying to register BlockManager
16/01/14 16:00:04 INFO storage.BlockManagerMasterEndpoint: Registering block manager localhost:60074 with 517.4 MB RAM, BlockManagerId(driver, localhost, 60074)
16/01/14 16:00:04 INFO storage.BlockManagerMaster: Registered BlockManager
16/01/14 16:00:04 INFO repl.SparkILoop: Created spark context..
Spark context available as sc.
16/01/14 16:00:05 INFO repl.SparkILoop: Created sql context..
SQL context available as sqlContext.
scala>
```

## 9.6.3 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，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

## 9.6.3 Spark RDD基本操作

**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。

### 常用的几个Action API介绍

Action API	说明
<code>count()</code>	返回数据集中的元素个数
<code>collect()</code>	以数组的形式返回数据集中的所有元素
<code>first()</code>	返回数据集中的第一个元素
<code>take(n)</code>	以数组的形式返回数据集中的前n个元素
<code>reduce(func)</code>	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
<code>foreach(func)</code>	将数据集中的每个元素传递到函数func中运行

### 常用的几个Transformation API介绍

Transformation API	说明
<code>filter(func)</code>	筛选出满足函数func的元素，并返回一个新的数据集
<code>map(func)</code>	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
<code>flatMap(func)</code>	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
<code>groupByKey()</code>	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
<code>reduceByKey(func)</code>	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

## 9.6.3 Spark RDD基本操作

- 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。

## 9.6.3 Spark RDD基本操作

**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。

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

## 9.6.4 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：

1. 下载**sbt-launch.jar**（下载地址 <http://pan.baidu.com/s/1eRyFddw>）
2. 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
cp ~/下载/sbt-launch.jar /usr/local/sbt
sudo chown -R hadoop /usr/local/sbt #此处的hadoop为系统当前用户名
```

3. 在安装目录中创建一个**Shell**脚本文件（文件路径：**/usr/local/sbt/sbt**）用于启动**sbt**，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 $0`/sbt-launch.jar "$@"
```

4. 保存后，还需要为该**Shell**脚本文件增加可执行权限，命令如下：

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

## 9.6.4 Spark应用程序

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），文件中的代码内容如下：

```
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))
  }
}
```

## 9.6.4 Spark应用程序

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 := "2.10.5"
libraryDependencies += "org.apache.spark" %% "spark-core" % "1.6.0"
```

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
```

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



```
hadoop@dblab:~/sparkapp
文件(E) 编辑(E) 查看(V) 搜索(S) 终端(T) 帮助(H)
[hadoop@dblab sparkapp]$ /usr/local/sbt/sbt package
[info] Set current project to Simple Project (in build file:/home/hadoop/sparkapp/)
[info] Compiling 1 Scala source to /home/hadoop/sparkapp/target/scala-2.10/classes...
[info] Packaging /home/hadoop/sparkapp/target/scala-2.10/simple-project_2.10-1.0.jar ...
[info] Done packaging. 打包成功
[success] Total time: 4 s, completed 2016-1-15 19:37:11
[hadoop@dblab sparkapp]$
```



## 9.6.4 Spark应用程序

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-2.10/simple-project_2.10-1.0.jar
```

该应用程序的执行结果如下：

**Lines with a: 58, Lines with b: 26**