Spark系统

BIG DATA

Spark系统

CONTENTS

Spark产生背景

2009年：由美国加州伯克利大学AMP实验室开发，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

Hadoop：
在2000个节点上排序
100TB数据：72分钟

VS

Spark：
在206个节点上排序
100TB数据：23分钟

Spark用十分之一的计算资源，获得了比Hadoop快3倍的速度！

2014年：Spark在2014年打破了Hadoop保持的基准排序纪录

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）

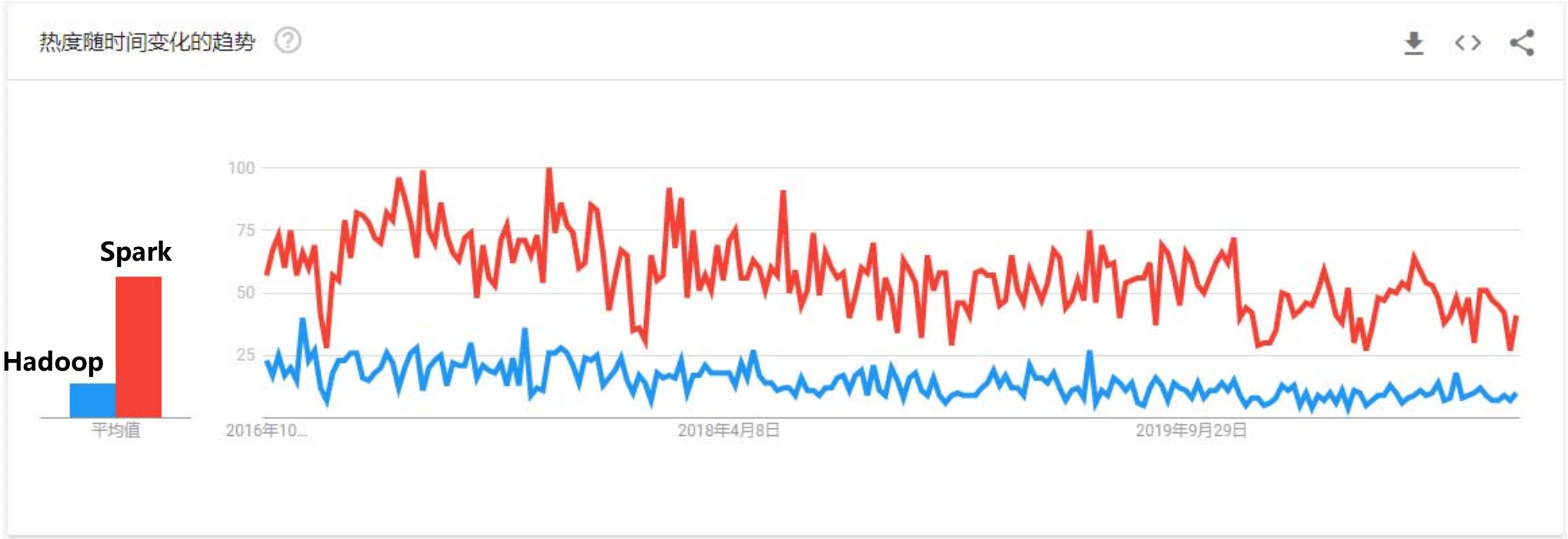
Spark的特点

- 运行速度快
  - 使用DAG（Directed Acyclic Graph）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
- 通用性强
  - 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
  - Spark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与Hadoop发展趋势对比

谷歌趋势：Spark与Hadoop对比（2013~2016年）

Spark与Hadoop发展趋势对比



谷歌趋势：Spark与Hadoop对比（2016~2020年）

Scala编程语言简介

- 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，Spark就是采用Scala实现的。
- Scala具有如下特点：
  - 强大的并发性，支持函数式编程（代码极大简化），可以更好地支持分布式系统
  - 语法简洁，能提供优雅的API
  - 兼容Java，运行速度快，且能融合到Hadoop生态圈中
  - 提供了REPL（Read-Eval-Print Loop，交互式解释器），提高程序开发效率

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教程>>：https://www.runoob.com/scala/scala-tutorial.html

Spark产生背景——Hadoop的缺点？

■ Hadoop虽然提供了一种高可靠的海量数据批处理框架，但其仍存在如下问题：

表达能力有限

- 所有复杂操作都需要抽象为Map和Reduce操作
- 一些计算任务难以进行抽象和表达

磁盘IO开销大

- Hadoop中Map任务的结果需要先存到磁盘，Reduce再从磁盘读取
- 每迭代一轮就需要进行一轮磁盘读写

延迟高

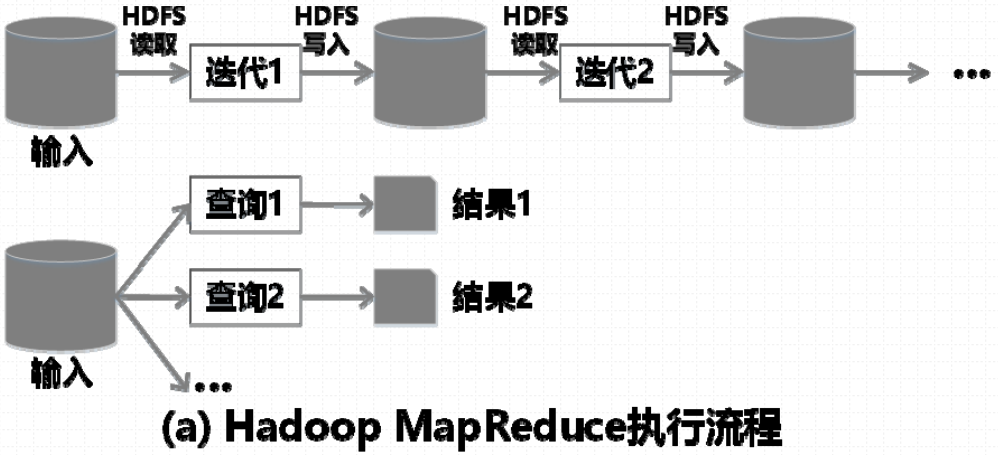
- Map和Reduce任务之间的衔接涉及的IO开销导致延迟
- 在前一个任务执行完成之前，其他任务就无法开始，难以胜任复杂、多阶段的计算任务

Spark与Hadoop的对比

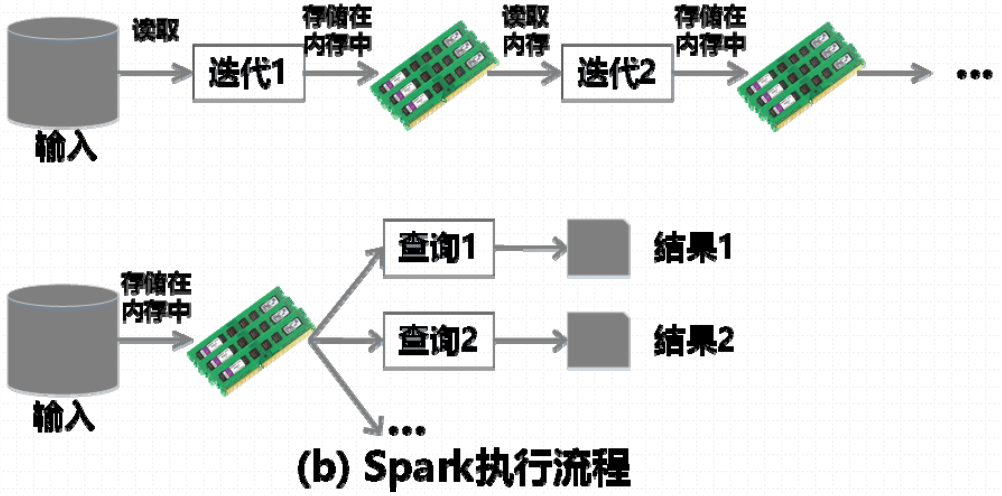
■ Spark在借鉴Hadoop MapReduce优点的同时，很好地解决了MapReduce所面临的问题，相比于Hadoop MapReduce，Spark主要具有如下优点：

更加灵活的编程模型	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
更高效的迭代运算	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
更优越的调度机制	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

Spark与Hadoop的执行流程对比



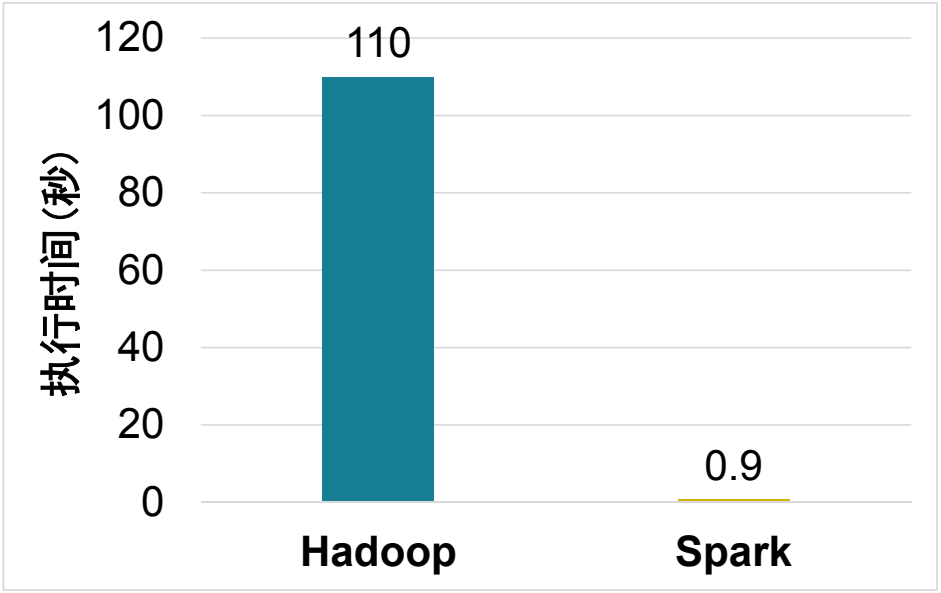
(a) Hadoop MapReduce执行流程



(b) Spark执行流程

Spark与Hadoop的对比

■ Hadoop与Spark执行逻辑回归的时间对比



- 使用Hadoop进行迭代计算，因需要大量磁盘IO，导致效率很低

VS

- 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

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：

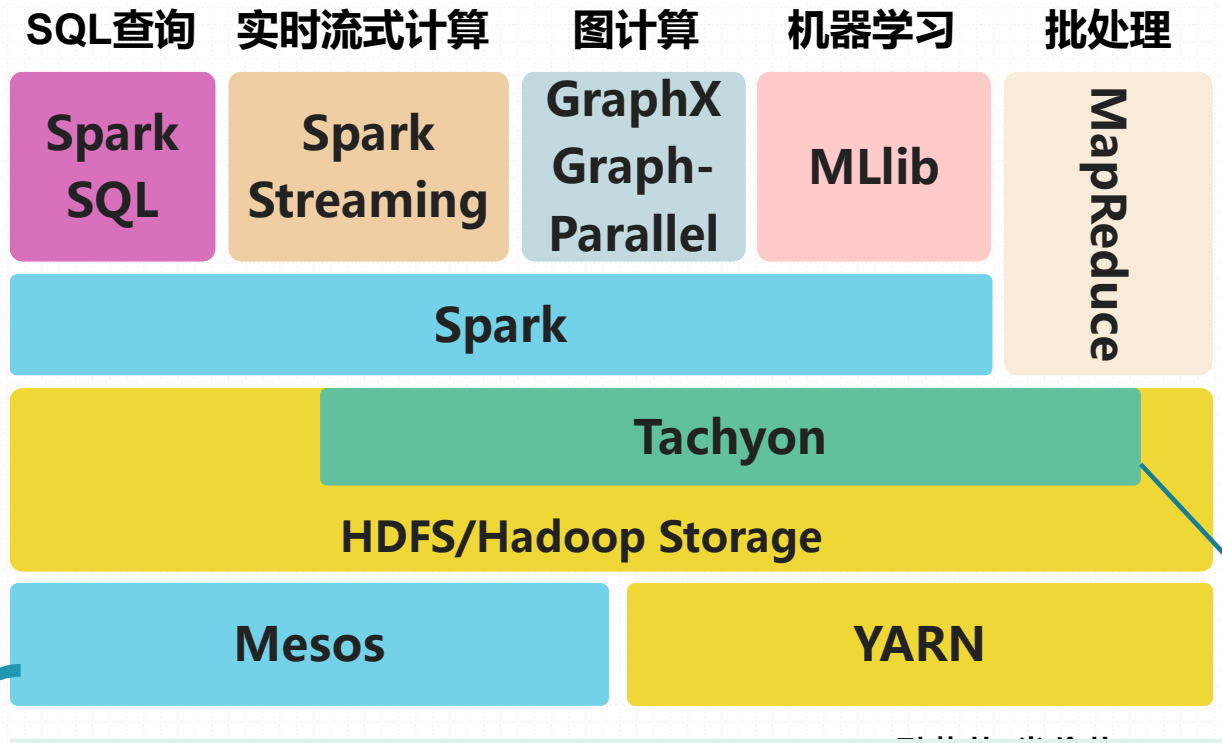
Spark生态系统

- 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，例如：
  - **MapReduce**：复杂的批量数据处理
  - **Cloudera Impala**：基于历史数据的交互式查询
  - **Storm**：基于实时数据流的数据处理

- 这样做带来的问题：
  - 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。

Tachyon：一个高容错的分布式文件系统，允许文件以内存的速度在集群框架中进行可靠的共享

Mesos：加州大学伯克利分校的AMP实验室开发的一款开源集群管理软件

Spark生态系统

应用场景

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

资源管理

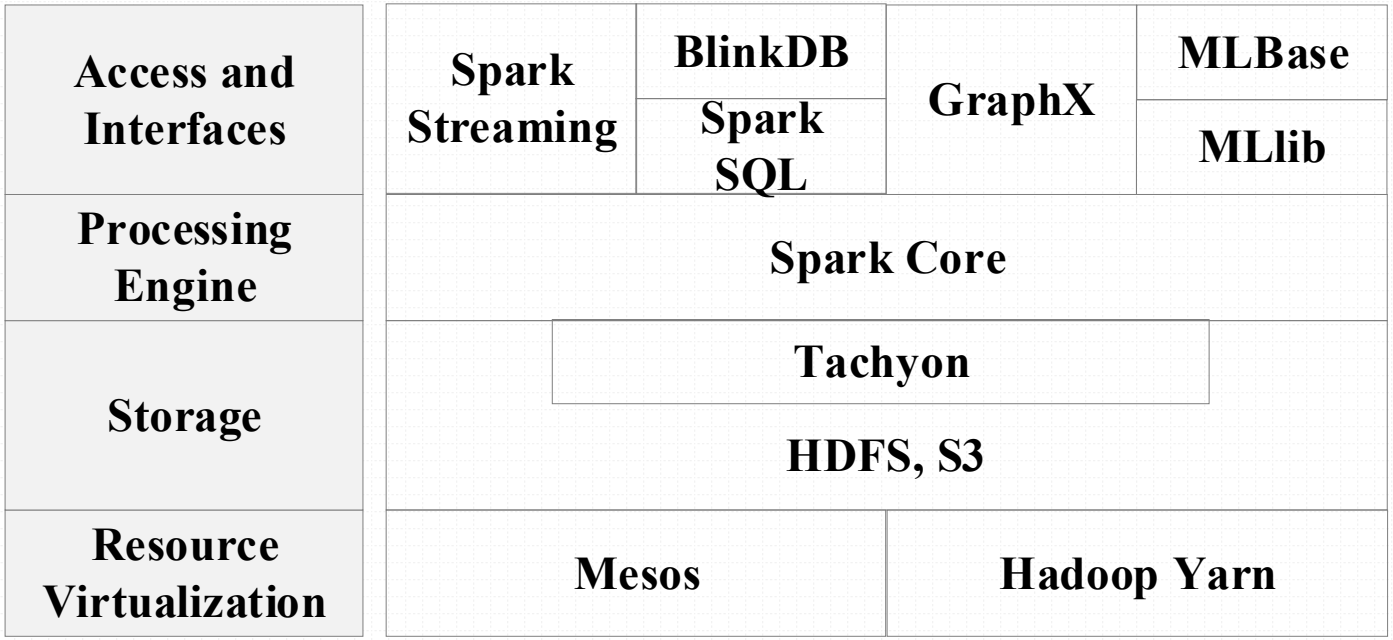
Spark可以部署在资源管理器YARN之上，提供一站式的大数据解决方案

BDAS

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

Spark生态系统

■ BDAS（Berkeley Data Analytics Stack）架构



■ 包含丰富的组件，如：

- Spark Core
- Spark SQL
- Spark Streaming
- MLLib
- GraphX等

Spark生态系统


Spark Streaming
提供流式计算功能
Spark SQL
提供交互式查询分析
Spark MLlib
提供机器学习算法库的组件
APACHE
Spark
Spark Core
提供内存计算
GraphX
提供图计算

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 Core
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	毫秒级、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

Spark系统

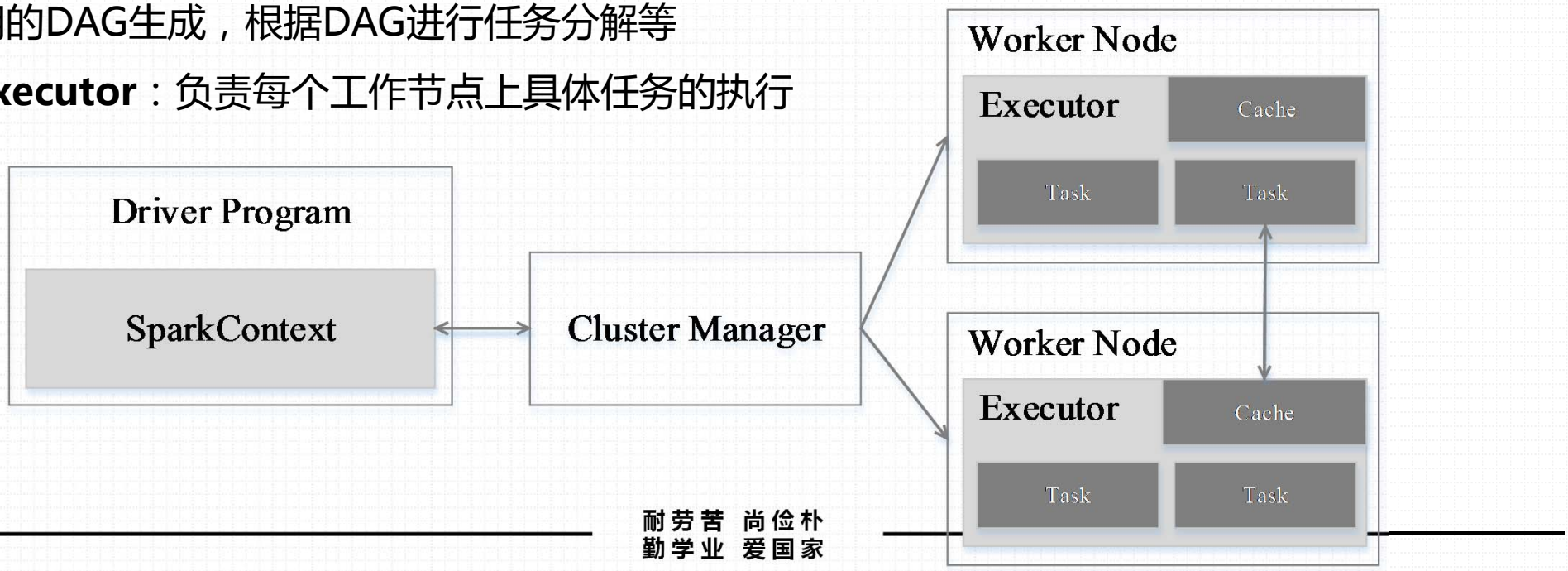
CONTENTS

Spark中的一些核心概念

- **RDD**：是Resillient 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

Spark的架构设计

- **Cluster Manager**: 集群资源管理器，负责对整个Spark集群的资源进行分配和管理调度
- **Worker Node**：工作节点，用于运行Spark作业中的任务
- **Driver Program**：任务控制节点，类似于应用程序的管家，负责整个应用的任务控制，如应用的DAG生成，根据DAG进行任务分解等
- **Executor**：负责每个工作节点上具体任务的执行



Spark的架构设计

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

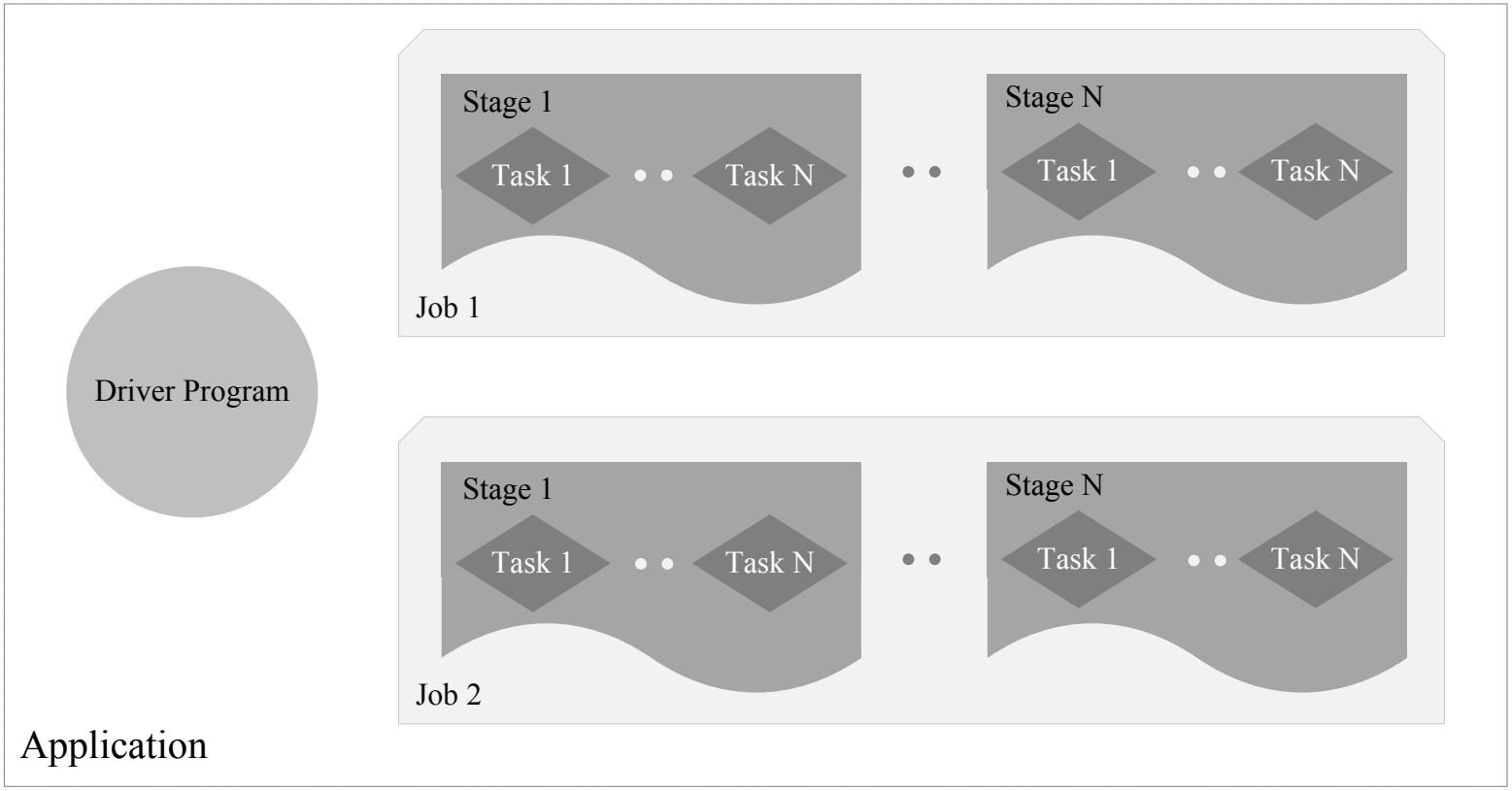
更快的启动速度

- Spark利用多线程来执行具体的任务，在一个Executor进程中会同时启动多个线程来执行相关Task，与之相比，Hadoop采用进程为单位执行任务，启动开销较大。

更小的IO开销

- 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，当内存空间足够时，会优先使用内存进行存储，只有内存不够时才会溢写到磁盘上，这种方式能够有效减小有效减少IO开销

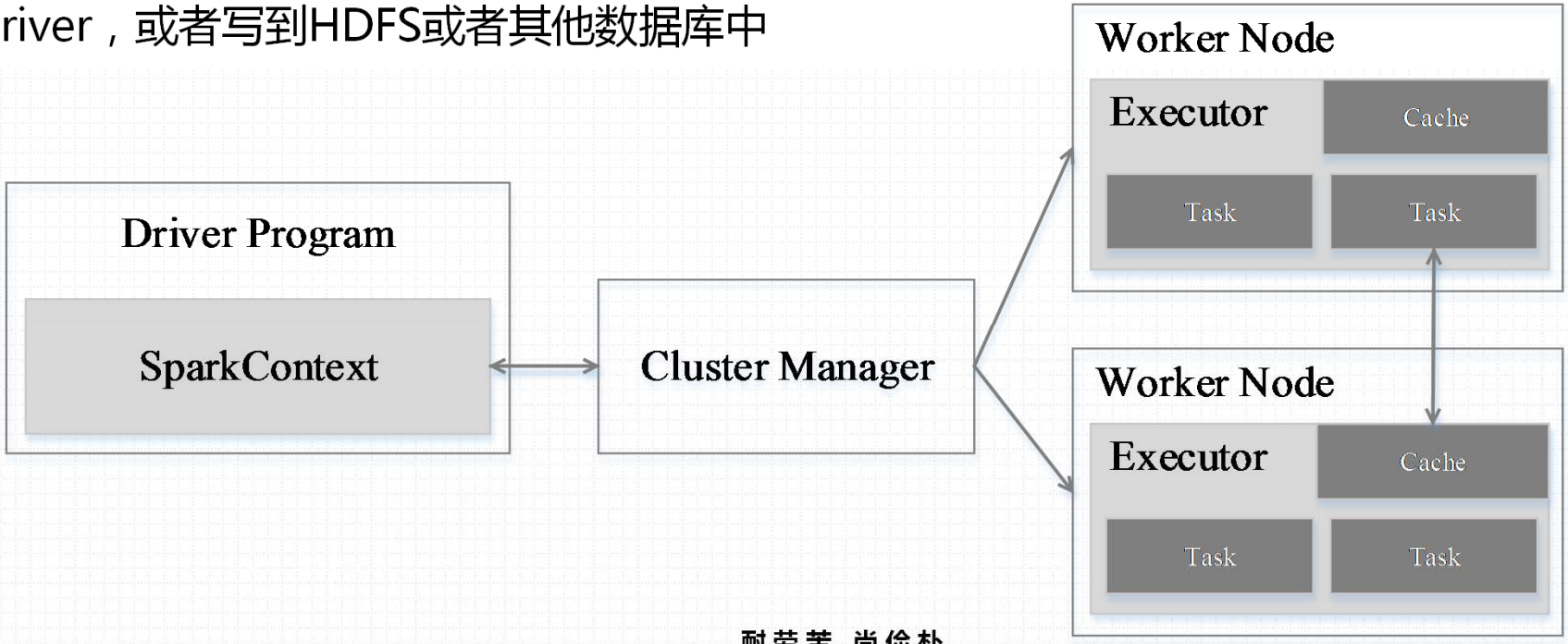
Spark中各种概念的相互关系



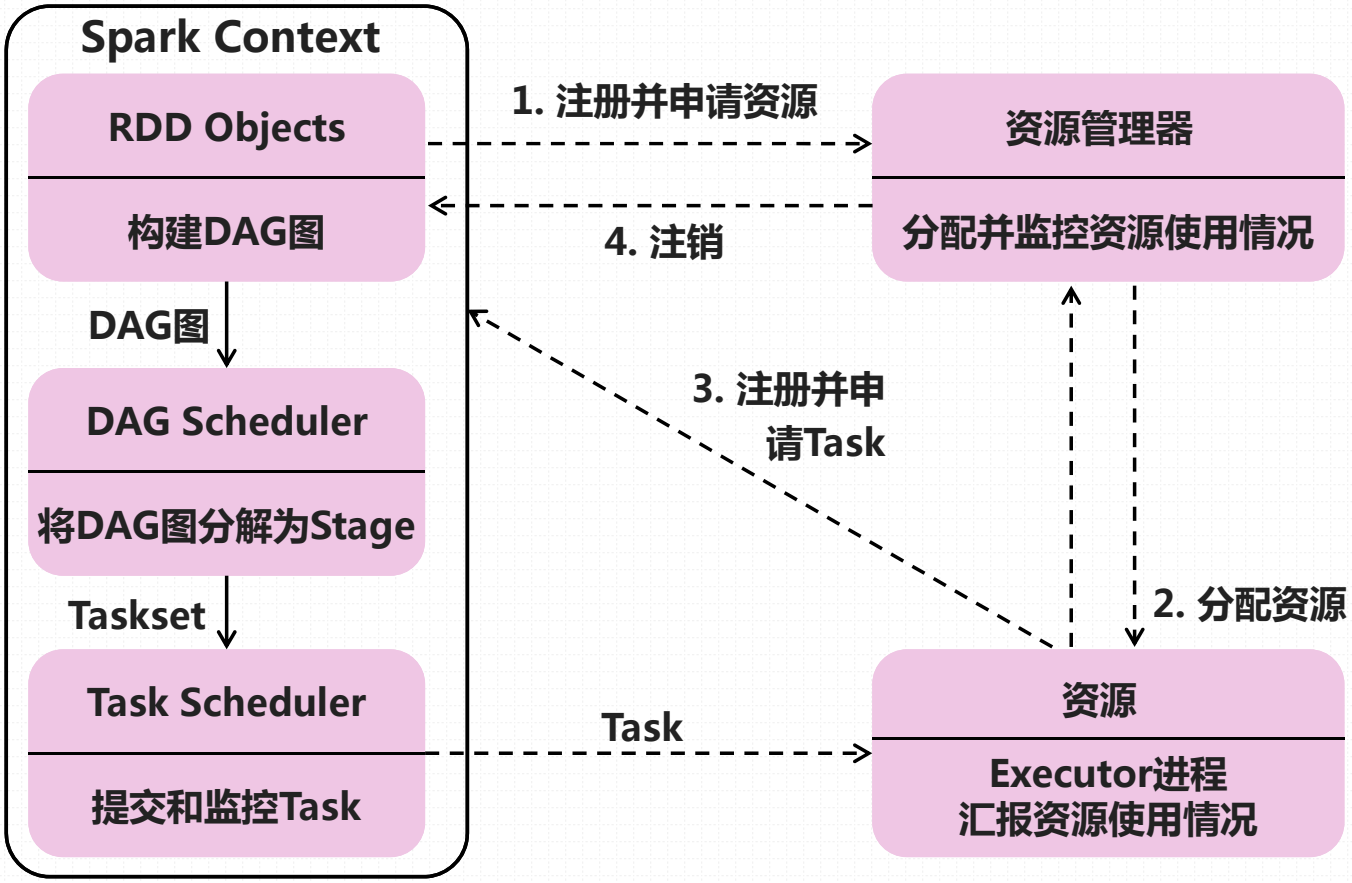
- 一个Application由一个Driver和若干个Job构成
- 一个Job由多个Stage构成
- 一个Stage由多个没有Shuffle关系的Task组成

Spark中各种概念的相互关系

- 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



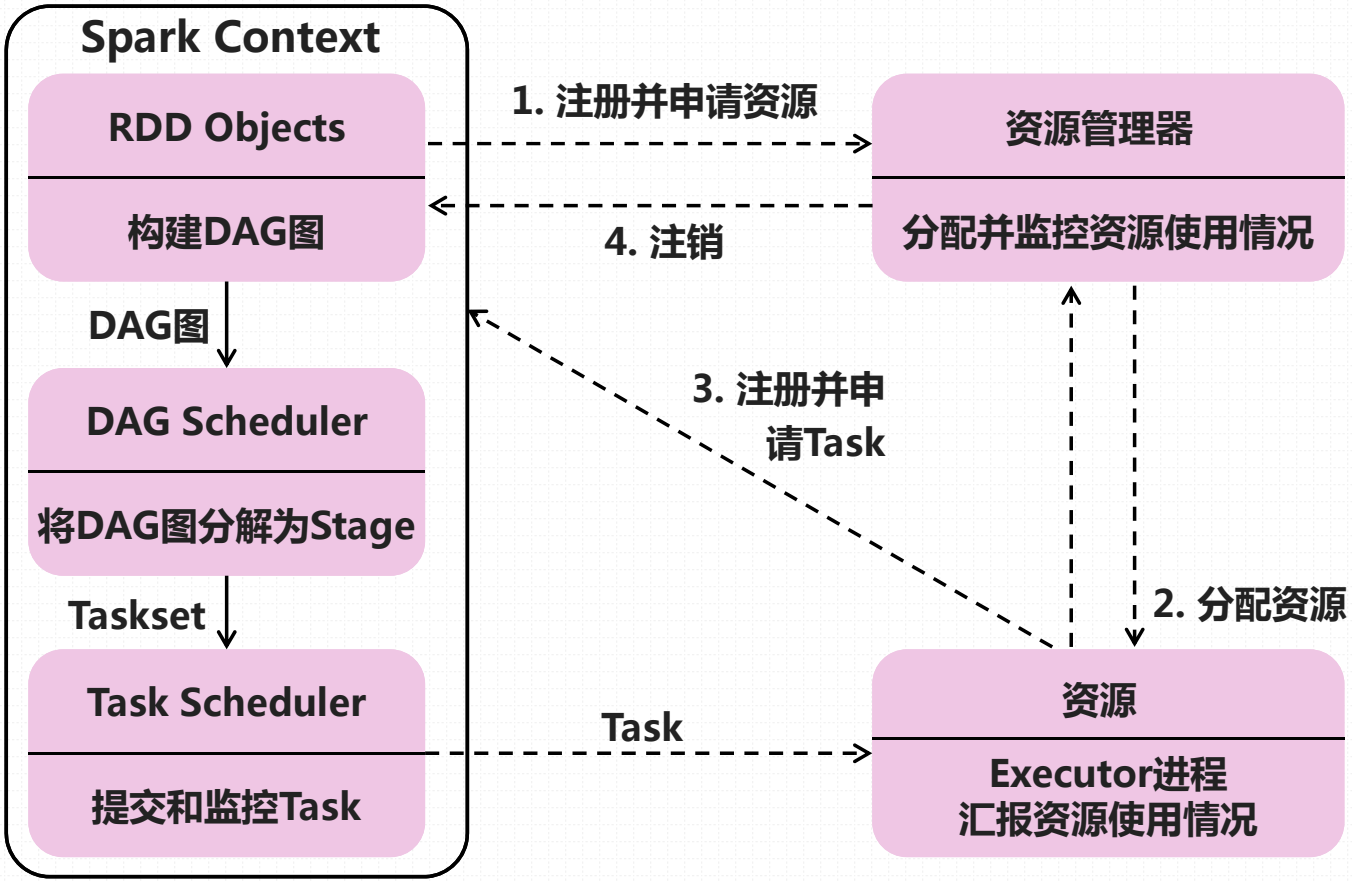
Spark运行基本流程



第一步

- 首先为应用构建起基本的运行环境，即由Driver创建一个SparkContext，作为用户应用程序的接口，进行资源的申请、任务的分配和监控

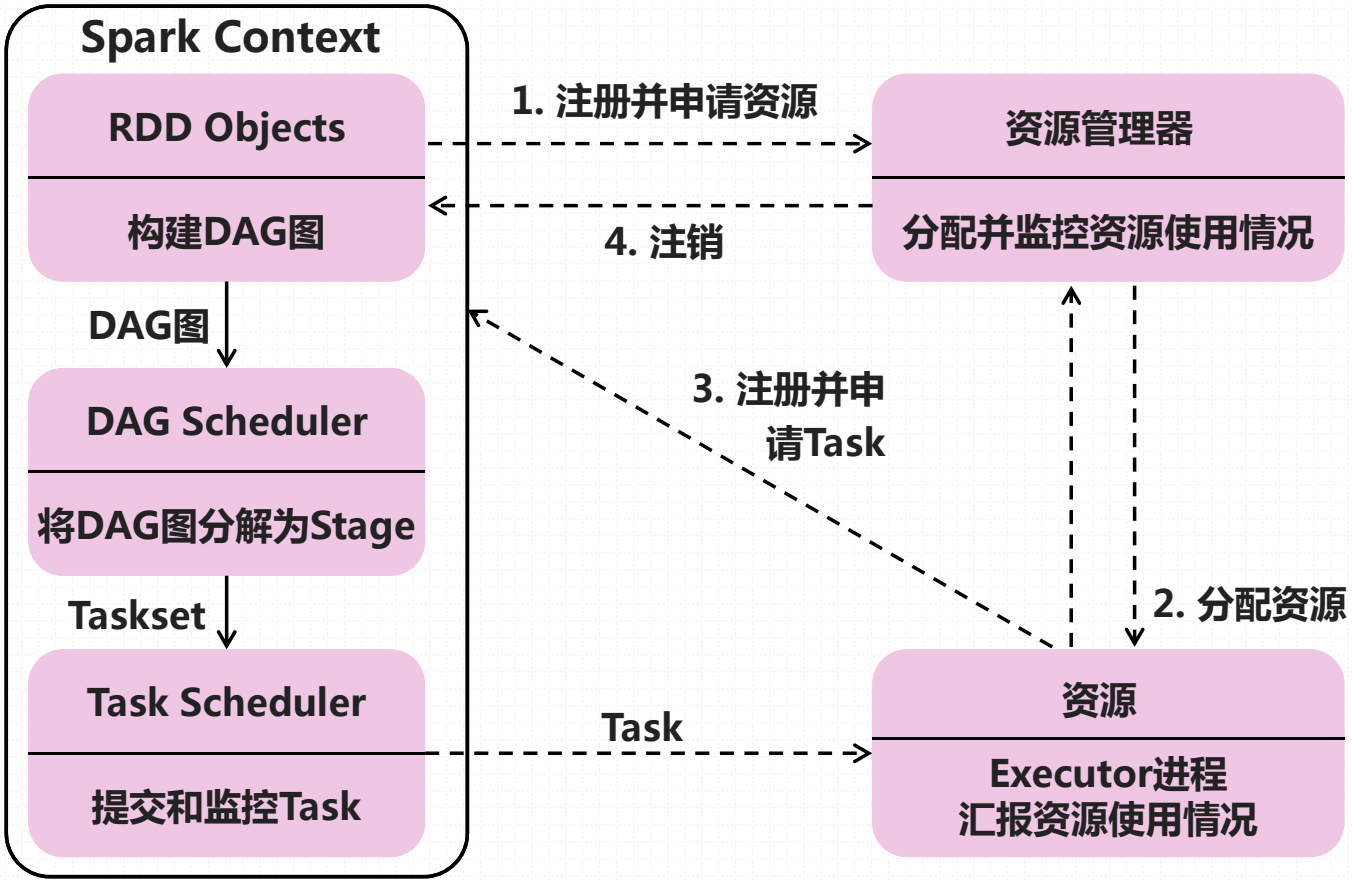
Spark运行基本流程



第二步

- 资源管理器为Executor分配资源，并启动Executor进程

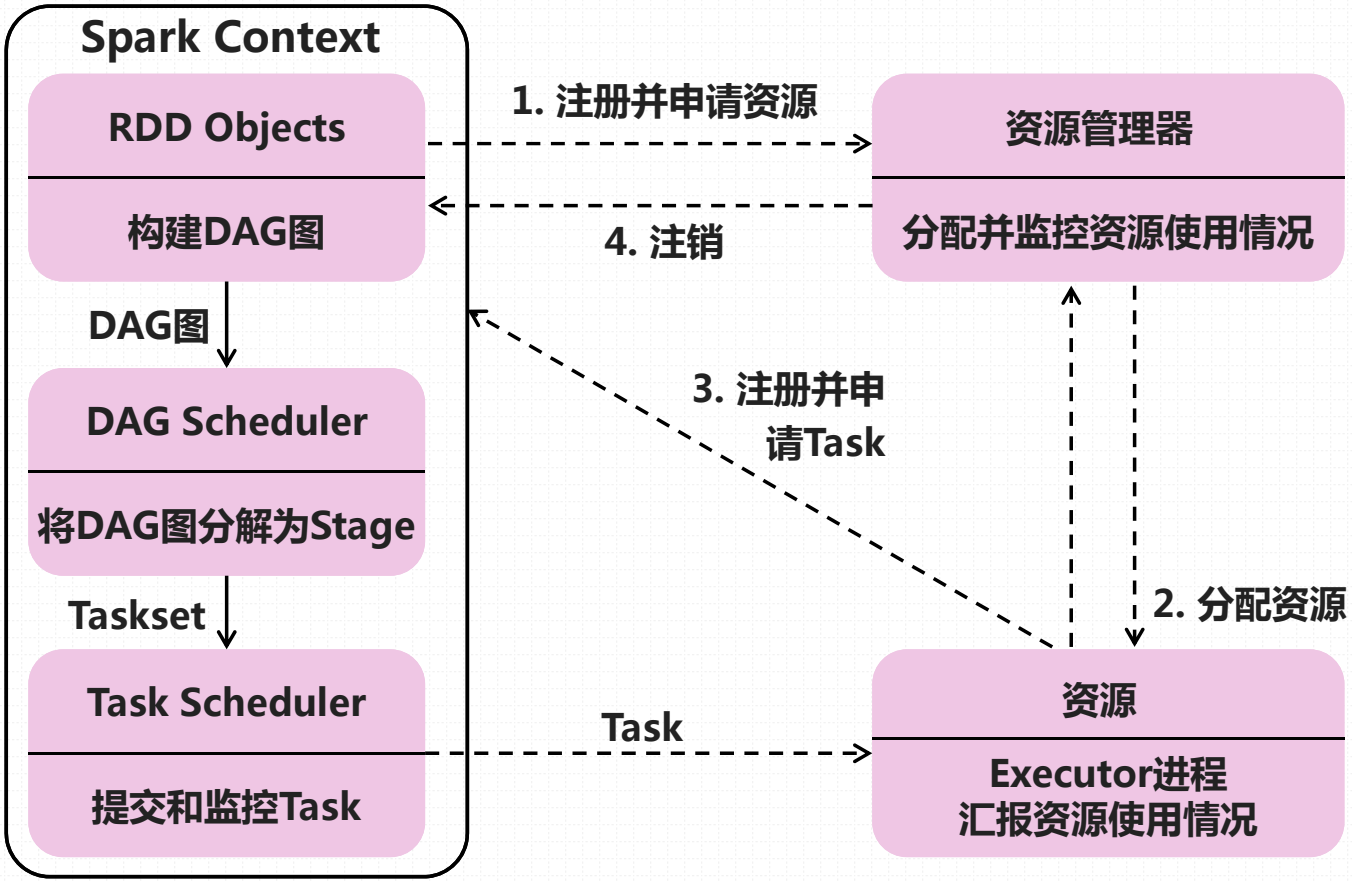
Spark运行基本流程



第三步

- 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

Spark运行基本流程



第四步

- 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

重庆大学 CHONGQING UNIVERSITY

Spark运行架构特点

01

- 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

02

- 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

03

- Task采用了数据本地性和推测执行等优化机制

Spark系统

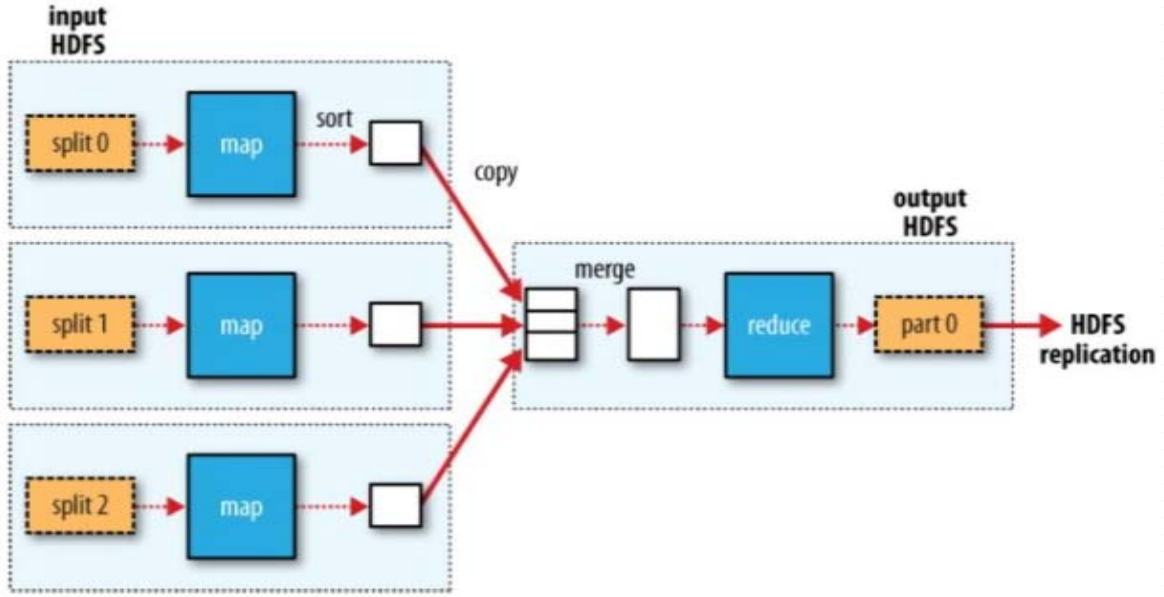
CONTENTS

RDD的设计背景

- 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的共同之处是：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(Resillient Distributed Dataset，弹性分布式数据集)

- **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**
- **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**



RDD的概念

- 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的概念

- 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的概念

■ RDD典型的执行过程如下：

RDD读取外部数据源进行创建

↓

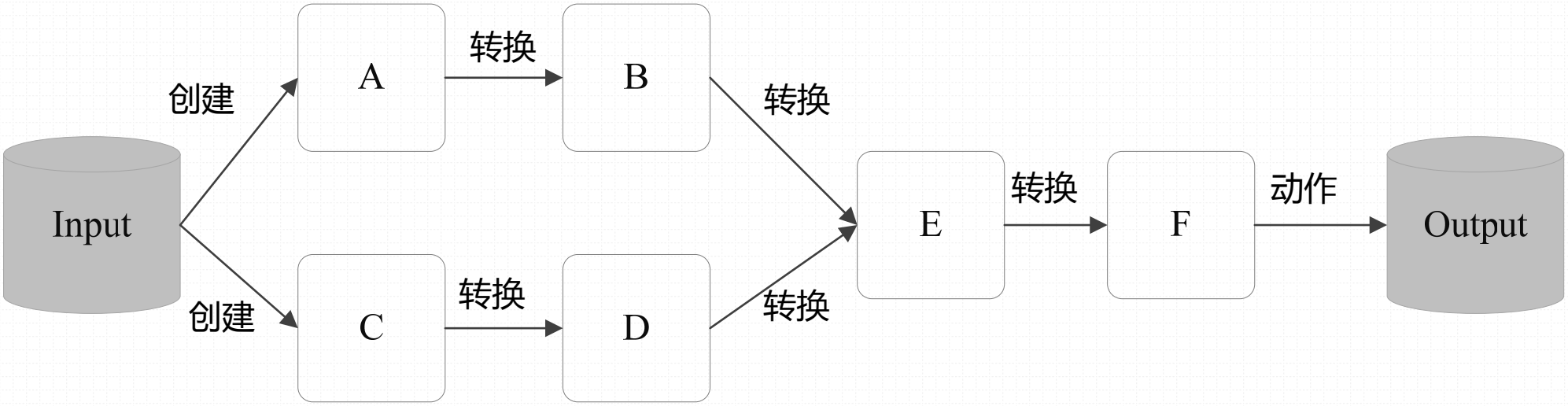
RDD经过一系列的转换操作，每一次都会产生不同的RDD，供给下一个转换操作使用

↓

最后一个RDD经过“动作”操作进行转换，并输出到外部数据源

RDD的概念

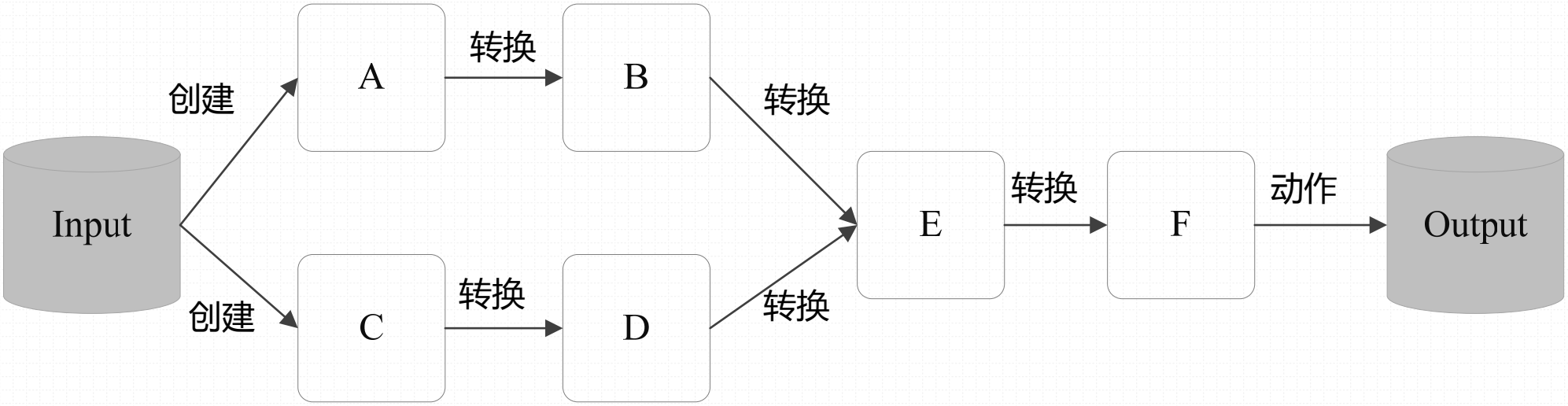
- RDD执行过程的一个实例如下：



- **转换操作（Transformation）**：并不会发生真正的计算，只是记录转换的轨迹
- **动作操作（Action）**：会触发从头到尾的真正计算，并得到结果

RDD的概念

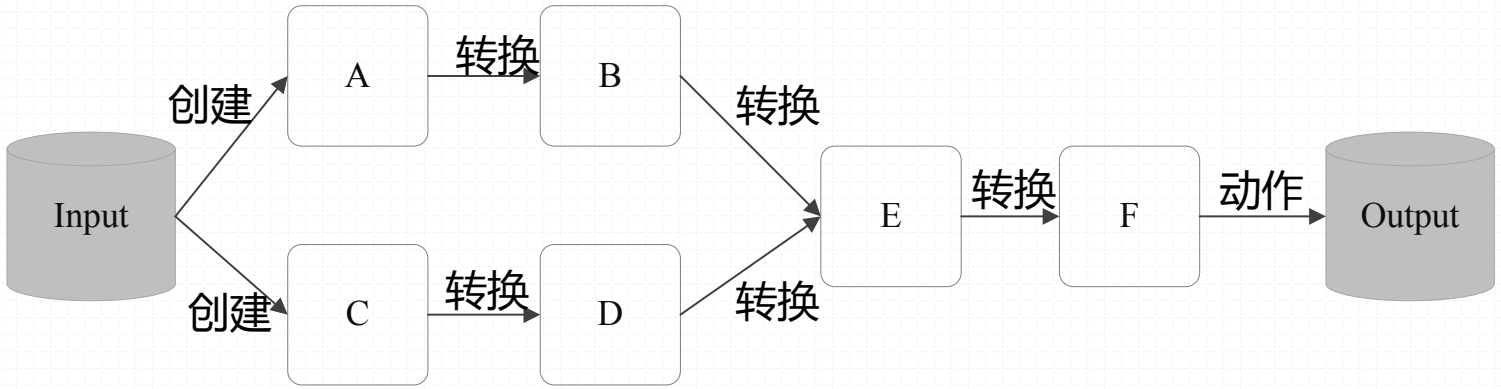
■ RDD执行过程的一个实例如下：



- 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

RDD的特性

- Spark采用RDD以后能够实现高效计算的原因
- **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



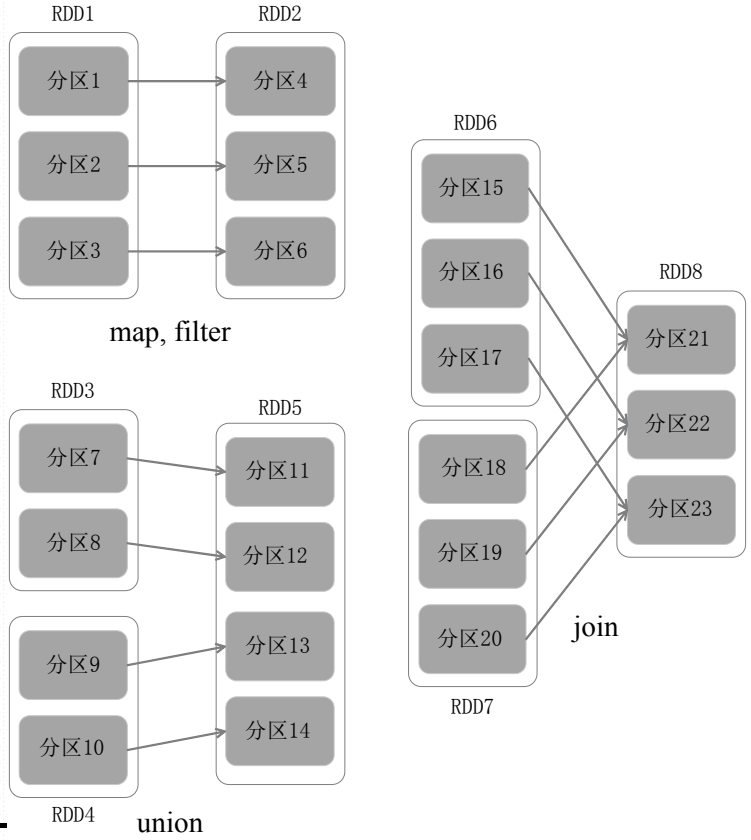
RDD的特性

- Spark采用RDD以后能够实现高效计算的原因
  - 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
  - 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

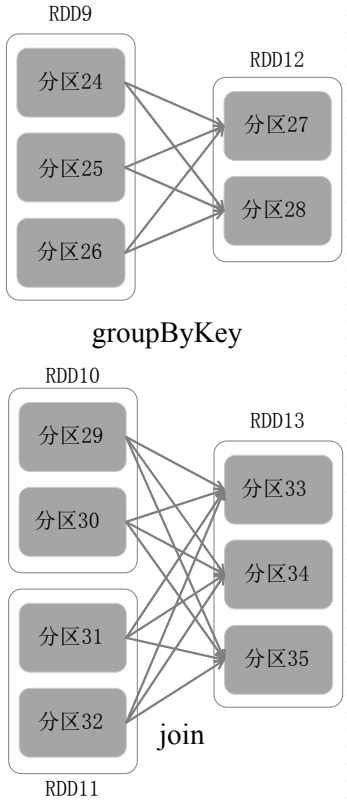
基于上述原因，Spark并行计算框架的运行效率要明显高于Hadoop

RDD的依赖关系和运行过程

■ RDD中的依赖关系包括**窄依赖**和**宽依赖**



(a)窄依赖



(b)宽依赖

- **窄依赖(Narrow Dependency)**：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

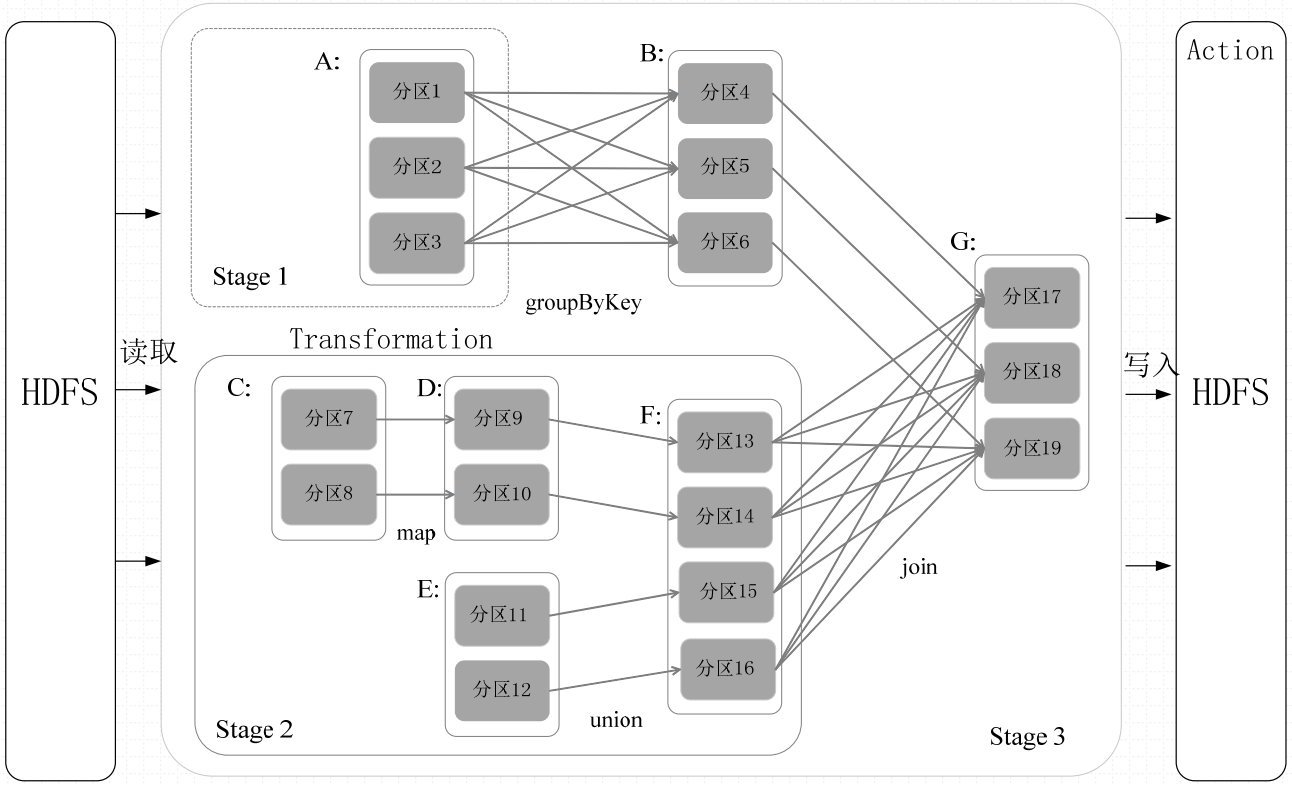
- **宽(Wide Dependency)**：表现为存在一个父RDD的一个分区对应一个子RDD的多个分区

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，若遇到宽依赖，则直接断开
  - 若遇到窄依赖，则把当前的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，从而使得数据可以直接在内存中进行交换，避免了磁盘IO开销

Stage的划分

■ 根据RDD分区的依赖关系进行Stage划分实例



■ 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

■ 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

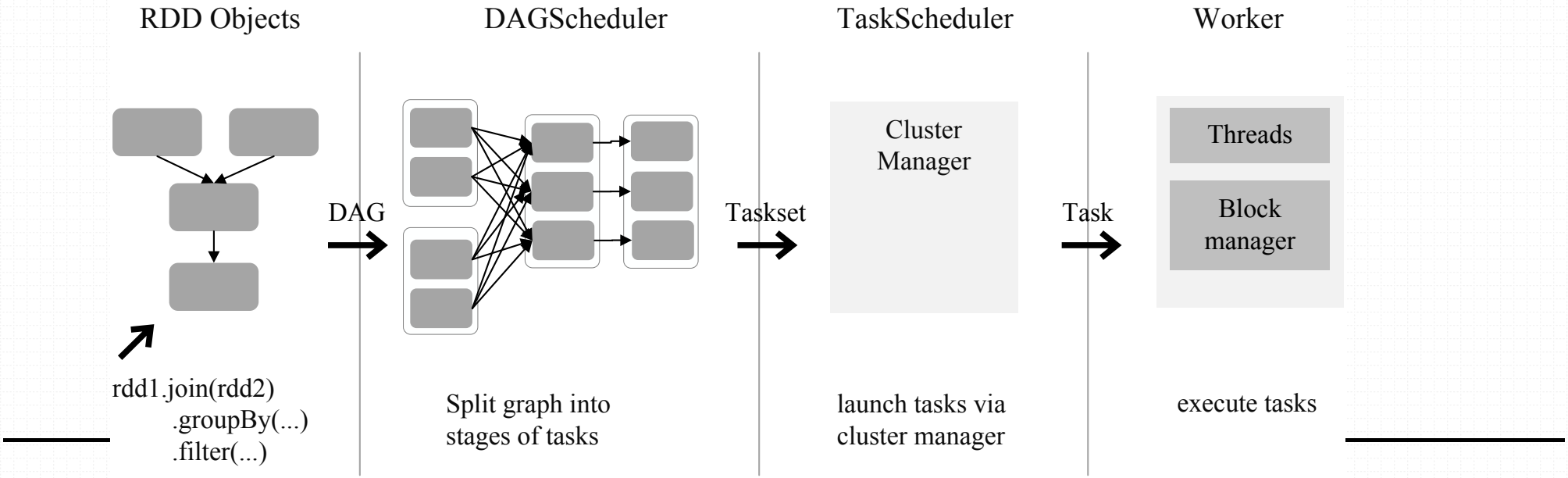
Stage的类型

- **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，其输入边界可以是从外部获取数据，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
- **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，其输入边界可以是从外部获取数据，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

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

创建RDD对象 → SparkContext负责计算RDD之间的依赖关系，构建DAG → 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。



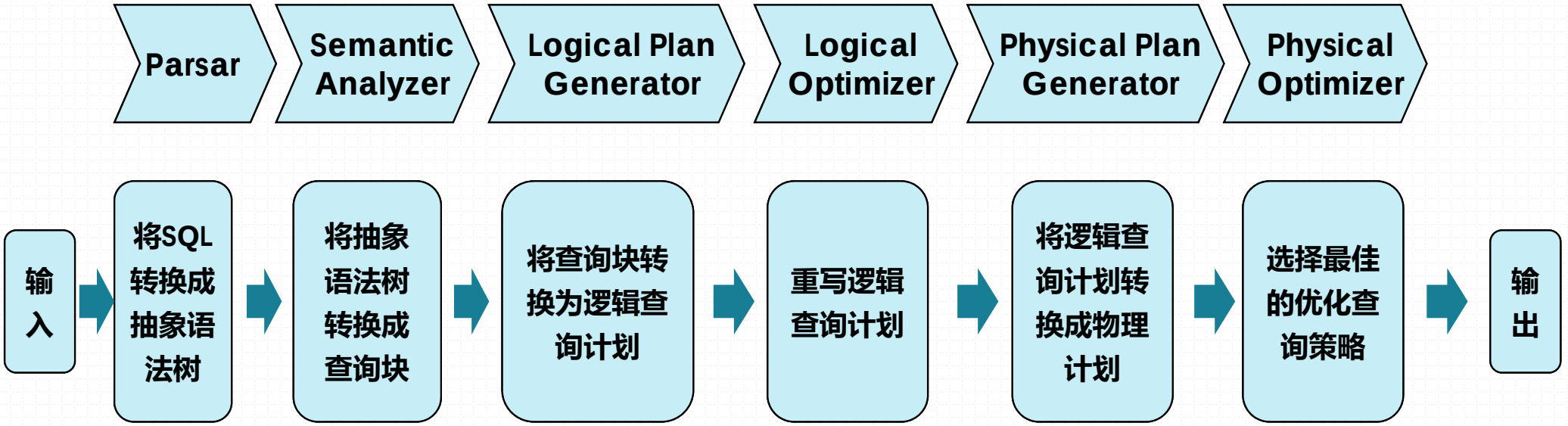
Spark系统

CONTENTS

从Shark说起

- **Shark的基本原理**
  - 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

Shark中SQL查询的MapReduce作业转化过程

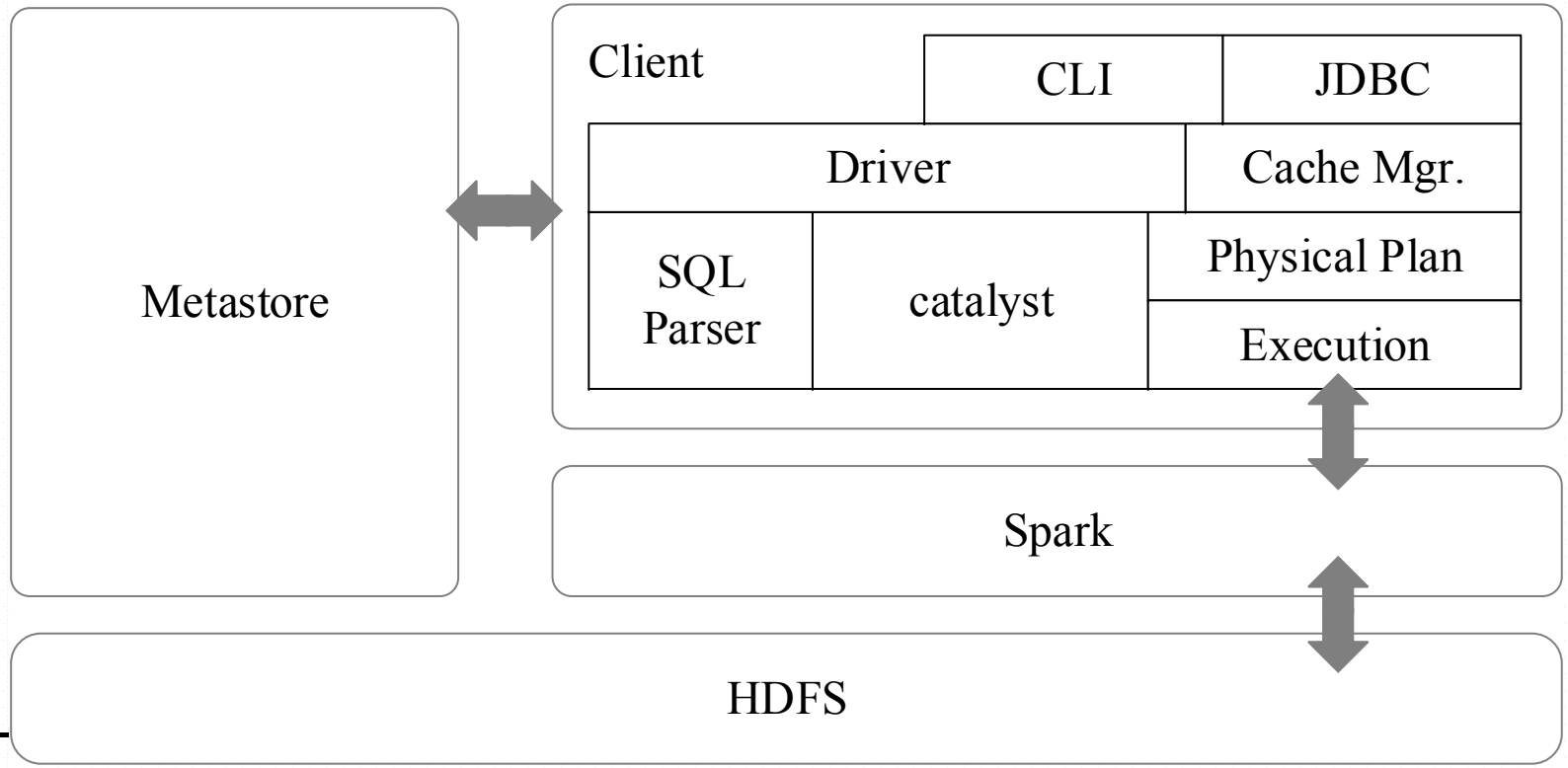


问题一：执行计划优化完全依赖于Hive，不方便添加新的优化策略

问题二：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

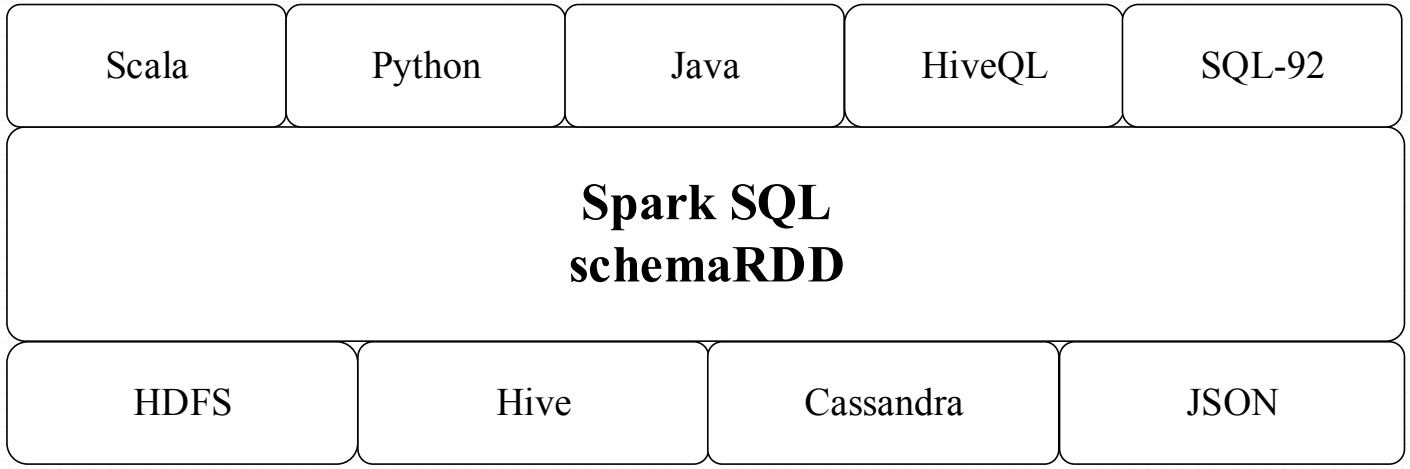
Shark SQL架构

- Spark SQL在Hive兼容层面仅依赖HiveQL解析、Hive元数据，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



Shark 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（备注：SchemaRDD在后来的Spark SQL版本中演化为DataFrame）
- 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



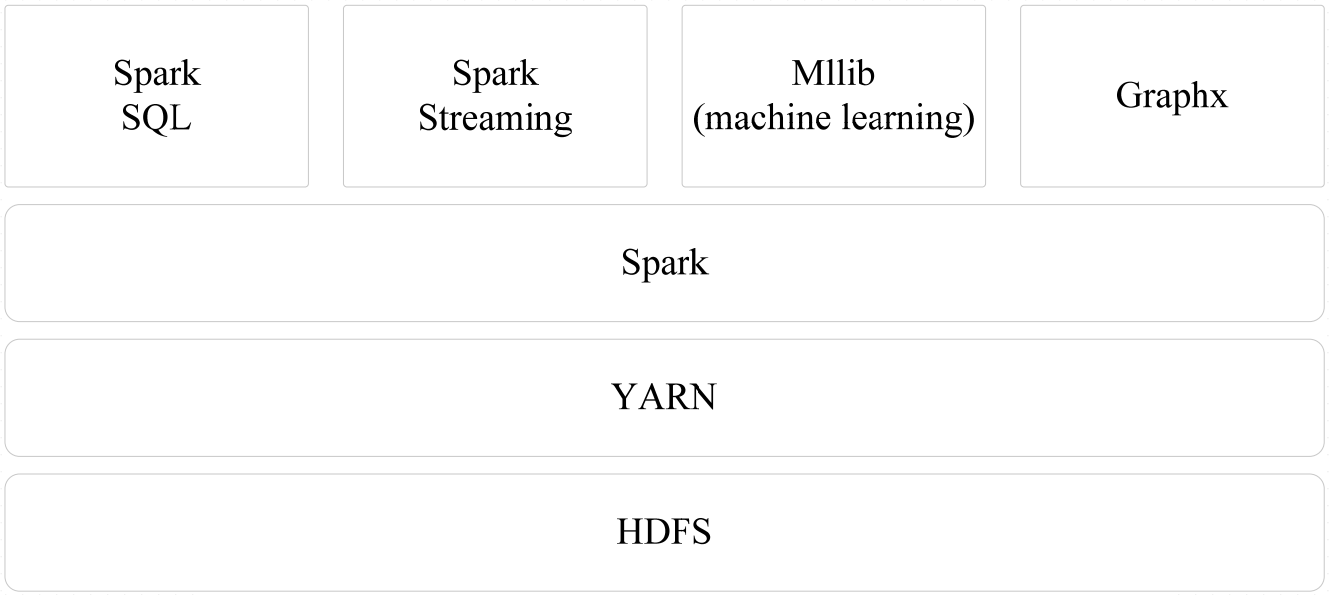
Spark SQL支持的数据格式和编程语言

Spark系统

CONTENTS

Spark的三种部署方式

- **方式一：Standalone**
  - 类似于MapReduce1.0，slot为资源分配单位
- **方式二：Spark on Mesos**
  - 由Mesos进行资源调度，Mesos与Spark有血缘关系，支持较高
- **方式三：Spark on YARN**
  - 由YARN进行资源调度



Spark on YARN架构图

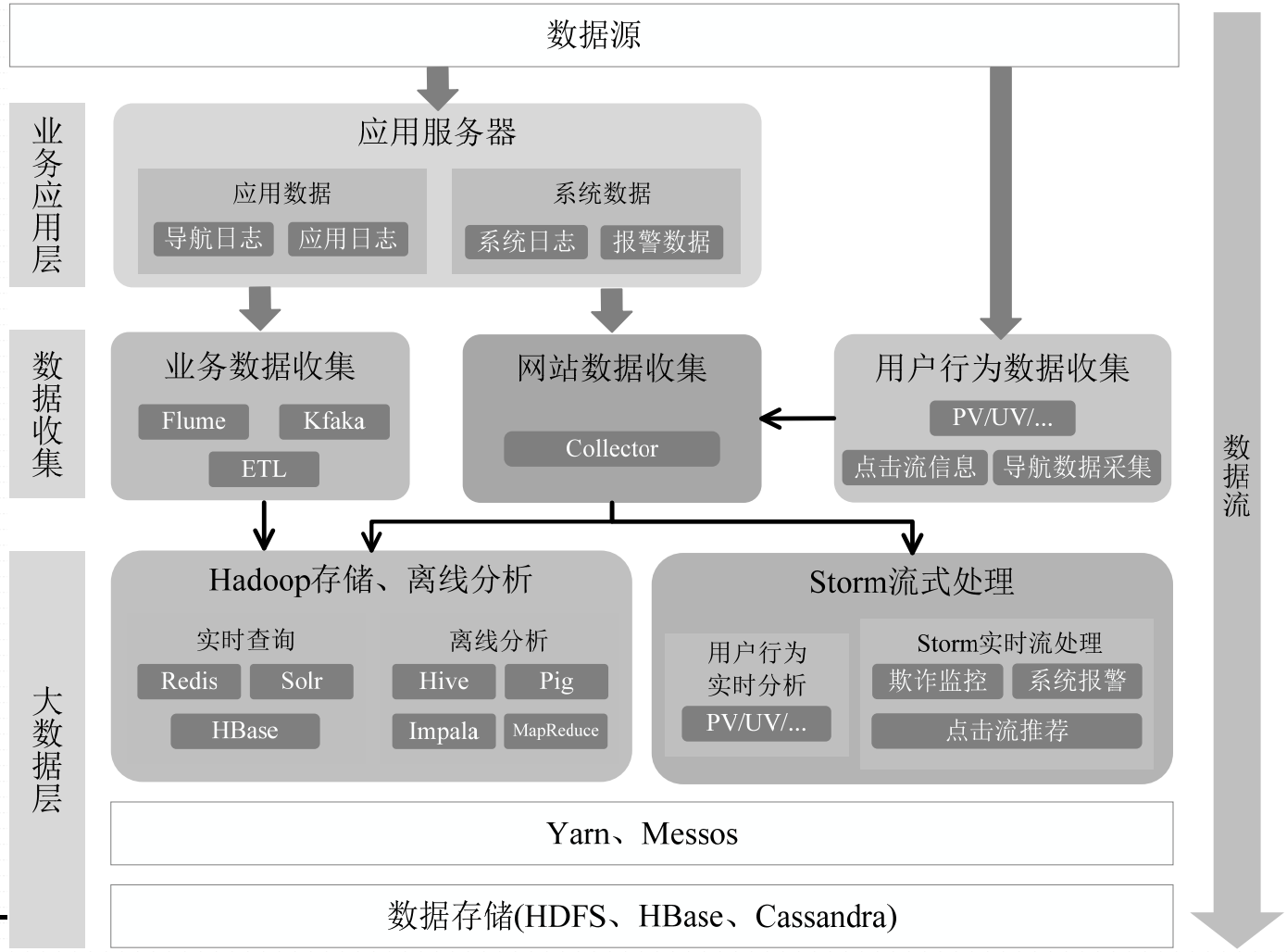
5 Spark的部署与应用

企业的典型部署方式

- 在Spark流行以前，企业普遍采用Hadoop+Storm方式进行部署

传统 Hadoop + Storm 部署方式

☹ 这种架构部署较为繁琐！

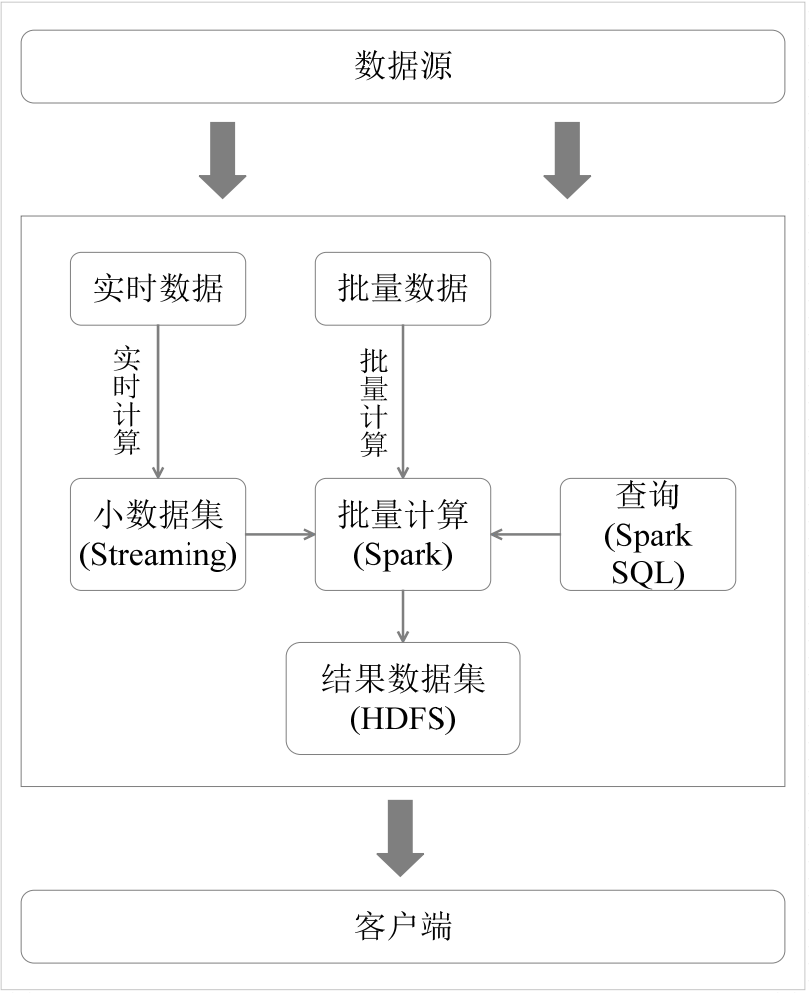


从Hadoop+Storm架构转向Spark架构

■ Spark架构优点

- 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

说明：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）

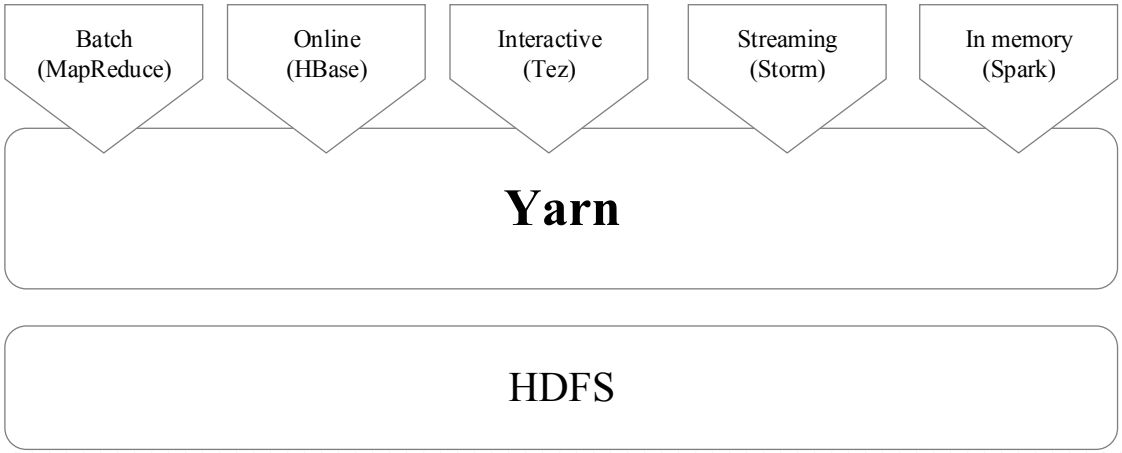


Spark架构的数据处理流程

Hadoop和Spark的统一部署

- **为什么需要统一部署？**
  - 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，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
- **将不同计算框架统一运行在YARN中，其好处如下：**
  - 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

Hadoop和Spark统一部署于YARN之上



重庆大学 CHONGQING UNIVERSITY

Spark的应用

- 若Spark运行于HDFS之上，则在启动Spark之前需要先启动Hadoop
- Spark Shell交互方式
  - 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

cd到Spark根目录，输入：
./bin/spark-shell
即可进入Spark Shell模式

```
hadoop@dblab:/usr/local/spark
文件(F)  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.spar
k.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 BlockManag
er
16/01/14 16:00:04 INFO storage.BlockManagerMasterEndpoint: Registering block man
ager 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>
```

Spark RDD基本操作

- **Spark的主要操作对象就是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，或从其他RDD转化而来
- **SparkContext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。
- **示例**
  - 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
```

重庆大学 CHONGQING UNIVERSITY

Spark RDD基本操作

- **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
- **常用的Action API和Transformation API如下表所示**

动作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

转换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

Spark RDD基本操作

■ Spark Shell API使用示例

- 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"))
```

第一条命令会返回一个新的RDD，第二天命令输出结果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，命令如下：

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

链式操作可避免存储不必要的中间数据，运行效率更高

Spark RDD基本操作

■ Spark Shell API使用示例

- 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）

Thank You!