第十章 Spark

第十章考点

- Spark特点
- Spark与Hadoop比较
- 优点
- Spark生态系统（设计理念）
- 核心概念（RDD）
- 进程优点
- RDD运行原理，执行过程，RDD特性，依赖关系

一、Spark概述

1. Spark简介
   Spark是基于内存计算的大数据并行框架
2. **Spark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，可以运行与独立的集群环境中，可以运行与Hadoop中，也可以运行在Amazon EC2等云环境中，并且可以访问HDFS，HBase，Hive等多种数据源
3. Scala简介
   Scala是一门现代多范式变成语言，运行于Java平台，并兼容现有的Java程序
   Scala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，能够提供优雅的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
4. **Spark与Hadoop的比较**
   Hadoop存在以下的一些缺点
   - 表达能力有限
   - 磁盘IO开销大
   - 延迟高
     - 任务之间衔接涉及IO开销
     - 在前一个任务执行完成之前，其他任务无法开始，难以胜任复杂的、多阶段的计算任务

Spark解决了MapReduce的以下问题：

- 使用Hadoop进行迭代计算非常耗费资源
- Spark将数据载入内存之后，之后的迭代结果都可以直接使用内存中的中间结果运算，避免了从磁盘中频繁读取数据

相比Hadoop MapReduce，Spark主要有如下优点：

- Spark的计算模式也属于MapReduce，但是不局限于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更加灵活
- Spark提供了内存计算，可以将中间结果放在内存中，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

二、Spark生态系统

1. 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（例如金融数据分析）
   - 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（例如电商）
   - 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（例如物联网）
     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
2. 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
3. **Spark设计理念以及功能**
   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

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

三、Spark运行架构

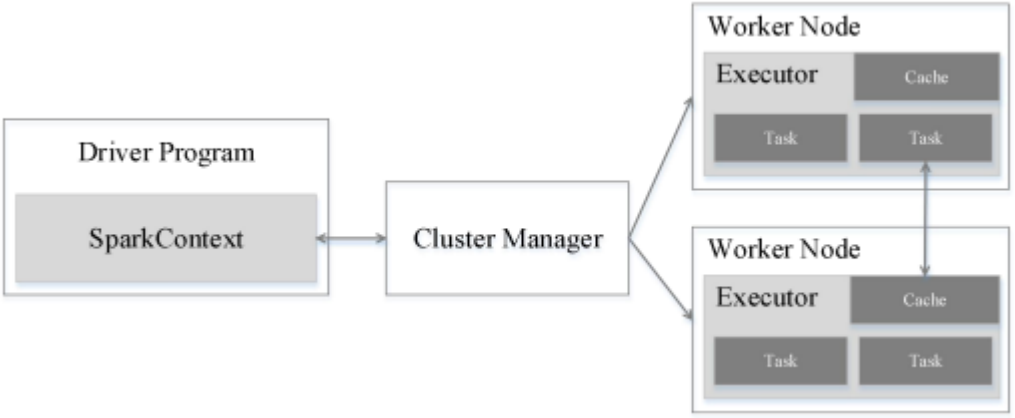
（一）Spark 核心概念字典

1. **RDD (Resilient Distributed Dataset)**
   - **定义**：弹性分布式数据集。它代表一个弹性的、不可变、可分区、里面的元素可并行计算的集合。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。
2. **DAG (Directed Acyclic Graph)**
   - **定义**：有向无环图，反映 RDD 之间的依赖关系。
3. **Executor**
   - **定义**：运行在工作节点（WorkerNode）上的一个进程，负责运行 Task。
4. **Application**
   - **定义**：用户编写的 Spark 应用程序（包含 Driver 和 Executor）。
5. **Job**
   - **定义**：一个 Job 包含多个 RDD 及作用于相应 RDD 上的各种操作。
   - **触发**：通常由 Action（动作）算子触发。
6. **Stage**
   - **定义**：Job 的基本调度单位，一个Job会分为多组Task，每组Task被称为Stage，也称为 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。
7. **Task**
   - **定义**：运行在 Executor 上的最小工作单元。

（二）Spark 架构设计

1. 物理节点与组件

Spark 运行架构由以下四部分组成：



- **Cluster Manager (集群资源管理器)**：负责资源管理（可自带，也可使用 Mesos 或 YARN）。
- **Worker Node (工作节点)**：运行作业任务的节点。
- **Driver (驱动节点)**：每个应用的任务控制节点。
- **Executor (执行进程)**：每个工作节点上负责具体任务的执行进程。

2. Executor 的优势 (对比 Hadoop MapReduce)

Spark 的 Executor 进程相比 Hadoop MapReduce 有两个显著优点：

1. **多线程模型**：利用多线程执行具体任务，减少了任务启动的开销（MR 采用进程模型）。
2. **BlockManager 存储模块**：将内存和磁盘共同作为存储设备，不需要读写 HDFS，有效减少 IO 开销。

3. 各种概念之间相互关系

- 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、
- 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

4.架构特点

- **专属进程**：每个 Application 都有专属的 Executor 进程，且在运行期间一直驻留。
- **解耦**：Spark 运行过程与资源管理器无关，只要能获取 Executor 进程并保持通信即可。
- **优化机制**：Task 采用了数据本地性和推测执行等优化机制。

（三）核心抽象：RDD (弹性分布式数据集)

1. 设计背景

- **痛点**：MapReduce 将中间结果写入 HDFS，导致大量数据复制、磁盘 IO 和序列化开销，不适合迭代算法和交互式数据挖掘。
- **解决**：RDD 提供抽象数据架构，通过**管道化 (Pipeline)** 避免中间数据存储，不必担心底层分布式特性。

2.RDD 核心概念

- **本质**：只读的分区记录集合。每个 RDD 分成多个分区，分布在集群不同节点上进行并行计算。
- **创建方式**：只能基于稳定的物理存储（如 HDFS）创建，或通过其他 RDD 执行转换操作（map, join 等）创建。
- **受限模型**：不支持细粒度修改（不适合爬虫），只支持粗粒度转换（适合 MR, SQL, Pregel）。
- **存储**：数据可以是 Java 对象，避免不必要的序列化/反序列化开销。

3. 操作类型

- **转换 (Transformation)**：惰性调用，不立即执行，构建血缘关系（如 map, filter）。
- **动作 (Action)**：触发实际计算，输出结果或保存（如 count, saveAsTextFile）。

4. 高效容错机制

- **传统方式**：数据复制或记录日志。
- **RDD 方式**：利用 **Lineage (血缘关系)**。
  - 只记录粗粒度的操作。
  - 丢失分区时，根据血缘关系重新计算。
  - 重算过程在不同节点并行，无需回滚系统。

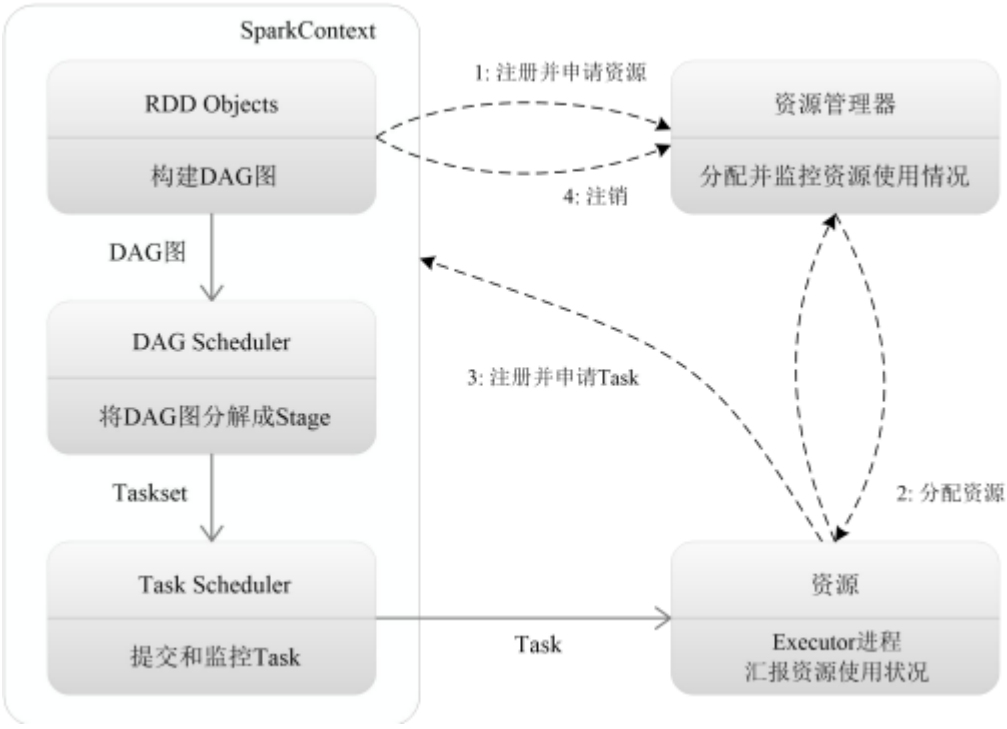
5. 依赖关系与 Stage 划分 (关键)

RDD 的依赖关系决定了 Stage 的划分：

- **依赖类型**：
  - **窄依赖 (Narrow Dependency)**：父 RDD 的一个分区对应子 RDD 的一个分区（1对1 或 N对1）。**有利于流水线优化**。
  - **宽依赖 (Wide Dependency)**：父 RDD 的一个分区对应子 RDD 的多个分区（1对N），存在 **Shuffle** 过程。**无法流水线优化**。
- **Stage 划分算法**：
  1. SparkContext 分析 RDD 依赖生成 DAG。
  2. 在 DAG 中进行**反向解析**。
  3. **遇到宽依赖就断开**，划分为不同的 Stage。
  4. **遇到窄依赖就加入当前 Stage**，尽量将窄依赖划分在同一个 Stage 中以实现“流水线”计算。

> **示例**：Stage 内部（如 map 到 union）全是窄依赖，形成流水线；Stage 之间由 Shuffle（宽依赖）分隔。

（四）Spark 运行基本流程



一个 Application 由一个 Driver 和若干 Job 构成，Job 由多个 Stage 构成，Stage 由多个 Task 组成。具体流程如下：

1. **环境构建**：
   - Driver 启动，创建 **SparkContext**。
   - SparkContext 负责资源申请、任务分配和监控。
2. **资源启动**：
   - 资源管理器为 Application 分配资源，启动 **Executor** 进程。
   - Driver 向 Executor 发送应用程序代码和文件。
3. **DAG 构建与解析**：
   - SparkContext 根据 RDD 依赖关系构建 **DAG 图**。
   - DAG 提交给 **DAGScheduler**。
   - DAGScheduler 将 DAG 解析成多个 **Stage (TaskSet)**。
4. **任务调度**：
   - DAGScheduler 将 TaskSet 提交给底层调度器 **TaskScheduler**。
   - TaskScheduler 将 **Task** 发放给 Executor 运行。
5. **任务执行**：
   - Executor 以多线程方式运行 Task。
   - 执行结果反馈给 TaskScheduler，再反馈给 DAGScheduler。
6. **结束**：
   - 运行完毕后写入数据（HDFS或其他数据库），释放资源。

总结 RDD 在架构中的流转：

1. 创建 RDD 对象。
2. **SparkContext** 构建 DAG。
3. **DAGScheduler** 拆分 Stage。

4. **TaskScheduler** 分发 Task。
5. **Executor** 执行 Task。

四、Spark SQL

（一）前身：Shark (Hive on Spark)

1. 基本原理

- **定义**：Shark 本质上是 **Hive on Spark**。
- **设计目标**：实现与 Hive 的兼容。
- **实现方式**：
  - **重用 Hive 逻辑**：在 HiveQL 解析、逻辑执行计划翻译、执行计划优化等方面，完全重用了 Hive 的原生逻辑。
  - **替换物理引擎**：仅将物理执行计划从 Hadoop **MapReduce** 作业替换成了 **Spark** 作业。
  - **流程**：`HiveQL -> Hive解析 -> Spark RDD操作`。

2. 存在的两大缺陷

Shark 的架构设计（重度依赖 Hive）导致了以下无法克服的问题：

1. **优化受限**：
   - 执行计划优化完全依赖于 Hive，导致无法方便地添加 Spark 专有的优化策略。
2. **线程安全问题（维护成本高）**：
   - **Spark** 是**线程级**并行。
   - **MapReduce** 是**进程级**并行。
   - **结果**：为了解决 Spark 在兼容 Hive 实现上的线程安全问题，Shark 不得不维护一套独立的、打了补丁的 Hive 源码分支。

（二）Spark SQL 架构

1. 架构改进（对比 Shark）

Spark SQL 摒弃了对 Hive 的重度依赖，实现了更彻底的接管：

- **仅依赖两点**：Spark SQL 在 Hive 兼容层面，仅依赖 **HiveQL 解析** 和 **Hive 元数据**。
- **接管时机**：从 HQL 被解析成 **抽象语法树（AST）** 起，后续工作就全部由 Spark SQL 接管。

2. 核心组件：Catalyst

- **定义**：函数式关系查询优化框架。

- **职责**：负责 Spark SQL 的 **执行计划生成** 和 **优化**。

（三）Spark SQL 核心概念与特性

1. 数据抽象：SchemaRDD

- **定义**：带有 **Schema**（结构/元数据）信息的 RDD。
  - *注：SchemaRDD 是 DataFrame 的前身，概念上等同于“结构化数据”。*
- **特点**：
  - 与普通 RDD 相似，但具有额外的结构信息（数据字段的类型信息）。
  - 允许用户在 Spark SQL 中直接执行 SQL 语句。

2. 数据源支持

Spark SQL 具有极佳的兼容性，数据可以来自：

- 内存中的 RDD
- Hive
- HDFS
- Cassandra
- JSON 格式数据
- 其他外部数据源

3. 多语言支持

目前支持三种编程语言进行开发：

- Scala
- Java
- Python

总结：Shark vs Spark SQL

比较维度	Shark	Spark SQL
本质	Hive on Spark (修改版 Hive)	Spark 的一个独立模块
依赖 Hive 程度	**重度依赖** (解析、优化、逻辑计划全靠 Hive)	**轻度依赖** (仅用 HiveQL 解析和元数据)
优化器	Hive 优化器	**Catalyst** (Spark 专用优化器)
中间形式	普通 RDD	SchemaRDD (结构化 RDD)
主要痛点	线程安全问题、优化扩展难	(解决了 Shark 的痛点)

五、Spark的部署和应用方式

（一）Spark 三种部署模式

Spark 支持多种集群管理器，主要分为以下三种：

1. **Standalone (自带模式)**
   - Spark 原生自带的资源管理模式。
   - **特点**：类似于 MapReduce 1.0。
   - **资源单位**：以 **Slot** 为资源分配单位。
2. **Spark on Mesos**
   - **特点**：由于 Spark 和 Mesos 有血缘关系（同出名门），因此 Spark 对 Mesos 有更好的支持。
3. **Spark on YARN**
   - **特点**：运行在 Hadoop 的 YARN 资源管理器上，是企业中最通用的部署方式。

（二）Spark 架构的运维与开发优势

采用 Spark 架构带来以下便利：

1. **易于部署**：支持一键式安装和配置。
2. **精细监控**：提供 **线程级别** 的任务监控和告警（相比 MR 的进程级更细致）。
3. **降低门槛**：
   - 降低了硬件集群和软件维护的难度。
   - 降低了任务监控和应用开发的难度。
4. **资源整合**：便于构建统一的硬件与计算平台资源池。

（三）技术选型注意：实时计算的限制

- **Spark Streaming 的局限**：
  - 无法实现真正的 **毫秒级** 流计算（本质是微批处理）。
- **选型建议**：
  - 对于需要 **毫秒级实时响应** 的企业应用，Spark Streaming 可能不适用。
  - 建议采用 **Storm** 或 Flink（现代替代方案）等纯流式计算框架。

（四）Hadoop 与 Spark 的统一部署 (基于 YARN)

1. 为什么要统一部署?

目前 Spark 无法完全取代 Hadoop 生态，两者需要共存：

- **功能互补**：Hadoop 生态中部分组件功能（如 Storm 的毫秒级响应）Spark 暂时无法完全替代。
- **迁移成本**：将现有的 Hadoop 组件应用完全转移到 Spark 上需要一定的重构成本。

2. 统一运行在 YARN 上的三大好处

将不同的计算框架（Spark, MapReduce, Storm 等）统一运行在 **YARN** 上，有以下优势：

1. **资源按需伸缩**：计算资源可以根据任务需求动态调整。
2. **集群利用率高**：不同类型的应用（不用负载应用）混搭运行，避免闲置。
3. **共享底层存储**：所有框架共享 HDFS，避免了数据在不同集群间迁移、复制的开销。

六、Spark编程实践

（一）RDD 编程入口：SparkContext

1. **核心对象**
   - **SparkContext** 是 Spark 程序的唯一入口。
   - **职责**：负责创建 RDD、启动任务、连接集群等。
2. **获取方式**
   - **代码中**：必须显示创建一个 SparkContext 对象。
   - **Spark Shell 中**：启动时会自动创建该对象，可以通过变量 `sc` 直接访问。

（二）RDD 的两大操作类型

Spark 的 RDD 操作分为两类，逻辑非常清晰，请务必区分：

操作类型	关键字	定义/作用	返回值	示例
转换	**Transformation**	基于现有数据集创建一个**新的 RDD**	**新 RDD** (不会立即计算)	`filter`, `map`
动作	**Action**	在数据集上进行运算，触发实际执行	**具体计算结果** (非 RDD)	`count`, `collect`

> **注意**：Transformation 是惰性的（Lazy），只有遇到 Action 时才会真正执行计算。

（三）代码实战示例

以下示例演示了从“读取文件”到“筛选数据”的全过程。
假设数据源为 Spark 安装目录下的 `README.md` 文件。

1. 创建 RDD (读取数据)

使用 `sc.textFile` 从外部存储系统读取数据。

```
// file:// 表示读取本地文件系统，而非 HDFS
```

```
val textFile = sc.textFile("file:///usr/local/spark/README.md")
```

2. 执行动作 (Action)

统计文本的总行数。

```
textFile.count()
// 输出结果: Long = 95 (表示共有 95 行)
```

3. 执行转换 (Transformation)

使用 `filter` 算子筛选出包含 "Spark" 字符串的行。

```
// line => line.contains("Spark") 是一个匿名函数，表示筛选逻辑
// 这行代码执行后，返回一个新的 RDD，名为 linesWithSpark
val linesWithSpark = textFile.filter(line => line.contains("Spark"))
```

4. 链式调用

对筛选后的新 RDD 再次执行 Action 操作。

```
linesWithSpark.count()
// 输出结果: Long = 17 (表示包含 "Spark" 的行有 17 行)
```

💡 流程总结图解

1. `sc.textFile` (加载) -> 得到 RDD1
2. `filter` (转换 RDD1) -> 得到 RDD2
3. `count` (对 RDD2 动作) -> 得到 结果 (17)