Spark 八股

MLE 算法指北

2025 年 3 月 9 日

1 Spark 基本概念与原理

1.1 Spark 基本架构

Spark 是一个基于 **内存计算**的 **分布式数据处理框架**，核心是 **RDD（Resilient Distributed Dataset）**，提供容错机制和并行计算能力。其架构主要包括 **Driver** 和 **Executor**，并通过 **DAG（有向无环图）**调度任务。

1.2 关键组件

- **Driver**：负责解析代码，生成 DAG，并将 Job 拆分为多个 Stage，管理任务调度。
- **Executor**：运行在 Worker 节点上，执行 Task，计算结果存入存储系统（HDFS、S3、Kafka）。
- **Partition**（分区）：Spark 计算的基本单位，每个 RDD 由多个 Partition 组成。
- **Task（任务）**：运行的最小计算单元，每个 Task 处理一个 Partition 的数据。
- **Job（作业）**：由一个 Action（如 `collect()`、`count()`）触发的计算任务，包含多个 Stage。
- **Stage（阶段）**：由多个 Task 组成，一个 Stage 处理一组连续的窄依赖（Narrow Dependency）。
- **Shuffle（数据洗牌）**：不同 Stage 之间数据重分配的过程，涉及网络 IO，影响性能。
- **Data Skew（数据倾斜）**：指部分 Partition 数据过大，导致任务执行时间不均衡。
- **Schedule（调度）**：Spark 采用 **DAG Scheduler** 和 **Task Scheduler**。

1.3 面试常考问题

1.3.1 1. RDD、DataFrame、Dataset 的区别

特性	RDD	DataFrame	Dataset
数据类型	任意对象	Row（表结构）	编译时类型安全的对象
结构化	无	有（Schema）	有（Schema）
性能	慢（无优化）	快（查询优化、列式存储）	介于 RDD 和 DataFrame 之间
适用场景	低级 API，需手动优化	SQL 查询、ETL、机器学习	适用于需要强类型的应用

表 1: RDD、DataFrame、Dataset 对比

1.3.2 2. 窄依赖（Narrow Dependency）和宽依赖（Wide Dependency）

- **窄依赖（Narrow Dependency）**：子 RDD 的每个分区 ** 只依赖一个或少量父分区 **，无 Shuffle（如 `map()`、`filter()`）。
- **宽依赖（Wide Dependency）**：子 RDD 的一个分区 ** 依赖多个父分区 **，触发 Shuffle（如 `groupByKey()`、`reduceByKey()`）。
- **优化建议**：
  - 尽量使用 ** 窄依赖 **，减少 Shuffle。
  - 使用 `reduceByKey()` 代替 `groupByKey()` 进行局部聚合。

1.3.3 3. 什么是 Shuffle，如何优化 Shuffle？

Shuffle 是指 Spark 在计算过程中需要在不同分区之间 ** 重新分配数据 **，导致网络 IO 和磁盘开销增加。

优化方法：

- **使用 `reduceByKey()` 代替 `groupByKey()`**，减少数据传输量。
- **调整 Shuffle 并行度**：如 'spark.sql.shuffle.partitions = 200'，提高并行计算能力。
- **使用 Broadcast 变量优化 Join**，避免大表 Shuffle。

1.3.4 4. 如何处理数据倾斜？

数据倾斜（Data Skew）指部分分区数据过大，导致任务执行时间不均衡。

优化方法：

- **样本打散（Salting）**：在 Key 上 ** 添加随机前缀 **，均衡数据分布。
- **Broadcast Join**：使用 ** 小表广播 **，避免大数据 Join 产生 Shuffle。
- **提高并行度**：增大 'spark.default.parallelism' 以减少单个 Task 负载。

1.3.5 5. Broadcast 变量的作用是什么？

- **Broadcast 变量**允许在 **Executor 之间共享只读数据 **，减少数据传输。
- **适用于小表 Join**，避免 Shuffle：

```
val smallTable = sc.broadcast(Map(1 -> "A", 2 -> "B"))
val joinedRDD = largeRDD.map(x => (x._1, smallTable.value.getOrElse(x._1, "Unk
```

1.3.6 6. Executor 和 Driver 的作用？

- **Driver**：
  - 运行 `SparkContext`，负责 **DAG 调度**、任务划分。
  - 发送 Task 给 Executor 并收集结果。
- **Executor**：
  - 负责 ** 执行 Task 计算 ** 并存储结果。
  - 如果有缓存，负责 ** 存储 RDD 数据 **（如 'cache()'）。

1.3.7 7. 为什么 `reduceByKey()` 比 `groupByKey()` 高效？

- `groupByKey()`：会 ** 先 Shuffle 所有数据 **，然后再在 Reduce 端分组，数据传输量大。
- `reduceByKey()`：会 ** 先在 map 端本地聚合 **，然后再进行 Shuffle，减少数据传输量，提高计算效率。

1.3.8 8. 如何优化 Spark 作业的性能？

- ** 减少 Shuffle**：`reduceByKey()` 代替 `groupByKey()`，使用 **Broadcast Join**。
- ** 调整并行度 **：设置 'spark.sql.shuffle.partitions = 200' 以提高计算吞吐量。
- ** 使用持久化（Cache）**：避免重复计算，'persist(StorageLevel.MEMORY$_A N D_D I S K$)'

1.3.9 9. Spark 如何实现 Fault Tolerance（容错）？

- **RDD 的 Lineage（血统）机制 **：
  - RDD 通过 **DAG 记录所有 Transformation 操作 **，丢失数据时可通过 Lineage 重新计算。
- **Checkpoint 机制 **：
  - 'rdd.checkpoint()' 将数据存入 HDFS，防止 DAG 过长导致计算成本过高。

方法	作用	适用场景
`cache()`	仅存储在内存	适用于 **Executor 内存充足 ** 的场景
`persist(StorageLevel.MEMORY_AND_DISK)`	内存不足时，写入磁盘	适用于 ** 数据较大，无法全部放入内存 **

表 2: Cache vs Persist 对比

1.3.10 10. RDD 的持久化（Cache 和 Persist）区别？

最佳实践：

- ** 如果数据多次使用，使用 `cache()` 提升性能 **。
- ** 如果数据量大，使用 `persist(StorageLevel.MEMORY_AND_DISK)`，避免 OOM**。
- ** 使用 `unpersist()` 及时释放内存 **，避免不必要的内存占用。

2 如何优化 spark 性能

2.1 样本打散（Salting）优化：SortByKey vs RepartitionAndSortWithinPartitions

在 Spark 计算过程中，**数据倾斜（Data Skew）**是常见的性能问题，尤其是在涉及 **Shuffle 操作**（如 `groupByKey`、`join`、`reduceByKey`）时，某些 Key 可能比其他 Key 具有 **过多数据**，导致任务执行时间不均衡，进而降低作业性能。

一种常见的优化方法是 ** 样本打散（Salting）**，即 **人为在 Key 上添加随机前缀或后缀**，使数据均匀分布，从而减少计算瓶颈。

在数据倾斜优化后，我们通常需要对数据进行 ** 排序 **，其中常用的方法有：

- **sortByKey()**：全局排序
- **repartitionAndSortWithinPartitions()**：分区内排序

2.1.1 sortByKey()

`sortByKey()` 是 **全局排序**操作，执行流程如下：

1. 先执行 **Shuffle**，将相同 Key 的数据移动到相同的分区。
2. 在每个分区内 ** 单独排序 ** 数据（默认升序）。
3. 排序后，每个分区的数据仍然是**局部有序**的。

特点：

- 触发**全局 Shuffle**，计算开销较大，适用于**小数据集**。
- 适合需要严格排序的场景，但由于需要全局排序，可能会加重 **数据倾斜**问题。

2.1.2 repartitionAndSortWithinPartitions()

`repartitionAndSortWithinPartitions()` 适用于**大规模数据排序**，其执行流程如下：

1. 先 **重新分区**数据，使相同的 Key 分布到同一个分区。
2. 在每个分区内 **单独排序**，但不会对整个数据集进行全局排序。
3. **减少 Shuffle 操作**，因为数据会在 Map 端提前排序，避免不必要的网络 IO。

特点：

- 只在 **每个分区内排序**，相比 `sortByKey()` 计算效率更高。
- 避免全局 Shuffle，更适用于 **大规模数据集**。
- 适用于**优化 Join、减少 Shuffle 负担、数据倾斜优化**等场景。

2.1.3 面试考点

- **sortByKey vs repartitionAndSortWithinPartitions 的区别？**
- **为什么 repartitionAndSortWithinPartitions 更适合大数据排序？**
- **如何通过 Salting 解决数据倾斜问题？**
- **如果一个 Key 的数据过多，会造成什么问题，如何优化？**
- **在 Join 操作中，如何减少 Shuffle 的影响？**

2.2 Cache 的运用原则与注意事项

在 Spark 中，`cache()` 和 `persist()` 用于 **缓存 RDD 或 DataFrame**，以 **避免重复计算**，从而提升作业性能。然而，不合理的缓存可能会 **占用过多内存、导致 OOM（Out of Memory）**，甚至影响整个 Job 的稳定性。因此，合理使用 Cache 是 Spark 性能优化的重要环节。

2.2.1 Cache 的运用原则

建议使用 `cache()` 或 `persist()` 的场景：

- **数据会被多次使用**：如果 RDD 或 DataFrame 在不同计算流程中多次使用，可以使用 `cache()` 避免重复计算。
- **RDD / DataFrame 计算代价较高**：如 `groupByKey`、`join`、`aggregation` 等高代价操作时，可使用 Cache 提高性能。
- **存储系统 IO 代价高**：如数据存储在远程存储（如 S3），每次计算都需要读取数据，`cache()` 可减少 IO 负担。
- **Shuffle 开销较大**：在一些宽依赖（Wide Dependency）操作（如 `reduceByKey()`）之后，如果数据需要被多次操作，可以 `cache()`，减少重复的 Shuffle 计算。

2.2.2 Cache 的注意事项

1. 内存占用问题 `cache()` 默认将数据存入 **Executor 的内存**，如果内存不足，缓存数据可能会被清理，甚至导致 OOM（内存溢出）。
解决方案：

- 调整 Executor 内存（`spark.executor.memory`）。
- 使用 `persist(StorageLevel.MEMORY_AND_DISK)`，让数据部分存储到磁盘。

方法	作用	适用场景
`cache()`	仅存储在内存中，不指定存储级别	适用于内存充足、数据量较小
`persist(StorageLevel.MEMORY_AND_DISK)`	内存不足时，数据写入磁盘	适用于数据较大，无法全部存入
`persist(StorageLevel.DISK_ONLY)`	仅存储在磁盘，不占用内存	适用于内存有限但需要避免重复

表 3: Cache vs Persist 的对比

2. Cache vs Persist

3. 及时释放 Cache 缓存数据后，若不再需要，**应使用 `unpersist()` 释放内存**，避免 Executor 过度占用内存：

4. Cache 适用于哪些算子？ 适用于：

- Transformation（转换操作）：`map()`、`flatMap()`、`filter()`、`groupByKey()`、`reduceByKey()` 等。
- 需要被 **多次使用**的 RDD 或 DataFrame。
- 计算代价较高的 **聚合操作**（如 `join()`）。

不适用于：

- 只被 **使用一次**的数据，`cache()` 没有意义，反而会占用内存。
- 小数据集：如果数据量小，缓存不会带来太大性能提升，反而可能浪费内存。

2.2.3 面试常考问题

- **Cache 和 Persist 的区别？**
- **为什么要使用 Cache，什么情况下不建议使用？**
- **Spark Cache 什么时候会被清理？**
- **Cache 会影响 Job 执行时间吗？**
- **如果一个 RDD `cache()` 了，但作业失败重启后，Cache 还存在吗？**

2.3 Broadcast 变量的使用与 Task 闭包

在 Spark 中，**Broadcast 变量**允许在集群的所有节点上 ** 高效共享只读变量 **，避免每个 Task 都拷贝同样的数据，从而减少 ** 网络传输开销 ** 和 ** 内存占用 **。此外，**Task 闭包（Task Closure）**也是 Spark 中的重要概念，涉及到 ** 变量的作用域和序列化 **，不合理的闭包可能会导致 ** 性能问题甚至错误 **。

2.3.1 Broadcast 变量的使用

1. 什么是 Broadcast 变量？ Broadcast（广播）变量允许将 ** 大数据集（如查找表、小型配置数据）** 仅 ** 在 Driver 端广播一次 **，并让所有 Executor 共享该变量，而不是每个 Task 都复制一份数据。这样可以 ** 减少网络传输开销 **，提高任务执行效率。

2. 适用场景

- **小表与大表 Join**：如果一个表（如维度表）较小，而另一个表非常大，可以使用 **Broadcast 变量 ** 来优化 Join。
- **Task 需要共享只读数据**：如果 Task 需要频繁读取同一份数据（如配置参数），使用 Broadcast 变量可以避免 ** 每个 Task 复制一份 ** 数据。
- **避免数据重复传输**：当数据较大且需要被所有 Task 访问时，Broadcast 变量可以 ** 减少网络 IO**。

执行流程：

1. 在 Driver 端创建一个 ** 广播变量 **（`sc.broadcast()`）。
2. Broadcast 变量被 ** 发送到所有 Executor**，而不是每个 Task 各自创建一份。
3. 在 Task 端使用 `broadcastVar.value` 访问广播数据。

3. Broadcast 变量的注意事项

- **不可修改**：Broadcast 变量是 ** 只读 ** 的，不能被修改。
- **适用于小型数据集**：如果 Broadcast 变量过大（如 GB 级别数据），可能会占用大量内存，影响计算性能。
- **需要手动销毁**：可以使用 `broadcastVar.unpersist()` 释放内存：

```
broadcastVar.unpersist()
```

2.3.2 Task 闭包（Task Closure）

1. 什么是 Task 闭包？ Spark 的 Task 通过 ** 闭包（Closure）** 将 Driver 端的变量 ** 序列化并发送到 Executor**，然后在 Task 中执行计算。
如果闭包涉及到 ** 非广播的共享变量 **，可能会导致每个 Task 复制 ** 大量的对象 **，影响性能。

2. Task 闭包示例

```
// 在 Driver 端定义变量
val factor = 10

// 在 Task 中使用 factor
val resultRDD = rdd.map(x => x * factor)
```

在上述代码中：

- `factor` 变量是在 Driver 端定义的。
- Spark 会 ** 将 `factor` 变量序列化 **，并随 Task 一起发送到 Executor。
- 这样，每个 Task 都会得到一个 ** 独立的 `factor` 副本 **。

3. Task 闭包的常见问题

- **变量作用域问题**:

```
var counter = 0
rdd.foreach(x => counter += 1)  // 错误: Driver 端的变量不会被 Executor更新!
```

由于 `counter` 是在 Driver 端定义的，但 'foreach' 运行在 Executor 上，因此 **Executor 的修改不会同步到 Driver**，导致逻辑错误。

- **非序列化变量问题**:

```
class NonSerializableClass { var data = 100 }
val obj = new NonSerializableClass()
val resultRDD = rdd.map(x => x * obj.data) // 可能会报 "NotSerializableExcepti
```

如果 `obj` 不能被序列化，Spark 可能会抛出 `NotSerializableException`。

2.3.3 如何优化 Task 闭包？

1. 使用 Broadcast 变量优化闭包 对于 ** 只读共享数据 **，应使用 **Broadcast 变量 **，避免每个 Task 拷贝一份:

```
val factor = sc.broadcast(10)
val resultRDD = rdd.map(x => x * factor.value)
```

这样，每个 Task ** 只需要访问一份广播变量 **，减少内存开销。

2. 避免 Driver 变量在 Executor 端被修改 如果需要在 Executor 端更新变量，应使用 **Accumulator**:

```
val counter = sc.longAccumulator("Counter")
rdd.foreach(x => counter.add(1))
println(counter.value) // 在 Driver 端打印结果
```

Accumulator 特点:

- Executor 可以 ** 更新 ** Accumulator。
- Driver 可以 ** 读取 ** Accumulator 结果。
- 但 ** 不能用于返回计算结果 **（只适用于累加统计）。

2.3.4 面试常考问题

1. 什么是 Broadcast 变量？它的作用是什么？
2. 什么时候适合使用 Broadcast 变量？什么时候不适合？
3. Task 闭包是什么？它的常见问题有哪些？
4. 如何避免 Task 闭包带来的变量作用域问题？
5. 为什么普通变量不能在 Executor 端修改后同步到 Driver？
6. Spark 中如何使用 Accumulator 实现全局计数？

2.4 Map vs MapPartitions

在 Spark 中，`map()` 和 `mapPartitions()` 都是 Transformation（转换）算子，它们的作用是对 RDD 进行转换处理，但两者在 ** 执行方式 **、** 性能 ** 和 ** 适用场景 ** 上存在显著区别。

2.4.1 Map 算子

1. 定义 `map()` 是 ** 行级（Row-wise）** 操作，即 ** 对 RDD 的每个元素 ** 都执行指定的函数，并返回一个新的 RDD。

2. 特点

- **对每个元素单独执行转换**，适用于行级别转换，如数值计算、数据格式转换等。
- **每个元素都触发一次函数调用**，可能会导致函数调用开销较高。
- **适用于独立计算任务**，不需要跨 Partition 操作数据。

3. 适用场景

- ** 简单的数据转换 **，如数值计算、字符串处理。
- ** 单独处理每条数据 **，无须考虑 Partition 结构。

2.4.2 MapPartitions 算子

1. 定义 `mapPartitions()` 是 ** 基于 Partition（分区级别）** 操作，即 ** 一次处理一个 Partition 的所有数据 **，并返回一个新的 RDD。

2. 特点

- **一次处理整个 Partition 的数据**，减少函数调用开销。
- 适用于 ** 需要批量操作的场景 **，如数据库写入、大数据计算等。
- 由于一次处理整个 Partition，** 容易导致内存溢出（OOM）**，如果 Partition 过大，可能会消耗大量内存。

3. 适用场景

- ** 批量数据处理 **，如数据库批量写入（JDBC）。
- ** 减少函数调用开销 **，适用于计算量较大的任务。

2.4.3 Map vs MapPartitions 对比

算子	执行方式	适用场景
`map()`	每个元素执行一次函数	适用于独立数据转换，适合小计算量操作
`mapPartitions()`	每个 Partition 处理一次	适用于批量处理，减少函数调用次数

表 4: Map vs MapPartitions 对比

结论:

- `map()` 会 ** 对每条数据创建和关闭一次数据库连接 **，导致 ** 连接开销过大 **，影响性能。
- `mapPartitions()` ** 每个 Partition 只创建一次数据库连接 **，避免频繁创建连接，提高吞吐量。
- **如果操作是 IO 密集型（如数据库写入），推荐使用 `mapPartitions()` 以减少连接开销**。
- **如果操作是 CPU 密集型（如数学计算），可以使用 `map()`，避免单个 Partition 过大导致 OOM**。

2.4.4 面试常考问题

1. **`map()` 和 `mapPartitions()` 的区别？**
2. **为什么 `mapPartitions()` 比 `map()` 更高效？**
3. **`mapPartitions()` 可能会导致什么问题？如何避免？**
4. **什么时候应该用 `mapPartitions()`，什么时候应该用 `map()`？**
5. **如何使用 `mapPartitions()` 进行数据库批量写入？**

2.5 Reduce 算子优化

在 Spark 中，**Reduce 相关算子**（如 `reduce()`、`reduceByKey()`）用于聚合数据，是常见的计算操作。然而，错误的使用方式可能会导致 ** 性能下降 **，甚至 ** 数据倾斜 **。优化 Reduce 算子对于提高 Spark 作业的计算效率至关重要。

2.5.1 常见的 Reduce 相关算子

1. `reduce()` `reduce()` 直接对 RDD 的所有元素执行聚合操作，返回一个最终的单一结果值。**特点**：

- 适用于 ** 全局聚合 **，但不适用于 Key-Value 形式的数据。
- 由于 ** 所有数据需要经过 Shuffle**，当数据量较大时，可能会带来 ** 性能问题 **。

2. `reduceByKey()` `reduceByKey()` 适用于 Key-Value 形式的数据，会在 ** 本地（map 端）进行部分聚合 **，然后在 Reduce 端进行最终计算。

特点：

- 先在 **map 端局部聚合 **，减少 Shuffle 过程中传输的数据量，提高性能。
- 适用于 **Key-Value 聚合 **，如统计日志 PV、UV 等场景。

—

3. `groupByKey()` vs `reduceByKey()` `groupByKey()` 和 `reduceByKey()` 都可以用于 Key-Value 聚合，但前者会 ** 将所有相同 Key 的数据传输到同一分区 **，而后者会 ** 在 map 端进行本地聚合 **。

对比分析：

算子	执行方式	适用场景
`groupByKey()`	直接将相同 Key 的所有数据传输到同一分区	适用于少量 Key 的数据
`reduceByKey()`	先在 map 端局部聚合，再进行 Shuffle 传输	适用于大规模 Key-Value 计算，减少 Shuffle 开销

表 5: groupByKey vs reduceByKey 对比

结论：

- ** 避免使用 `groupByKey()` 进行聚合 **，因为它会导致 ** 数据倾斜 ** 和 **Shuffle 过大 **。
- ** 推荐使用 `reduceByKey()` 进行局部聚合 **，减少数据传输，提高计算效率。

—

2.5.2 Reduce 算子优化策略

1. 使用 `reduceByKey()` 代替 `groupByKey()` 如前所述，`reduceByKey()` 先在 **map 端进行局部聚合 **，减少了 ** 网络传输和 Shuffle 开销 **，比 `groupByKey()` 更高效。

2. 调整 Shuffle 并行度 Spark 通过 ‘spark.default.parallelism‘ 和 ‘spark.sql.shuffle.partitions‘ 控制并行度，适当增加 **Shuffle 分区数 ** 可以 ** 避免单个 Task 负载过重 **，降低数据倾斜风险。

3. 处理数据倾斜：添加 Salting 如果 Key 过热，导致某些 Partition 负载过重，可以通过 ** 样本打散 (Salting) ** 进行优化。

4. 结合 `combineByKey()` 进行优化 `combineByKey()` 是更灵活的聚合算子，适用于 ** 不同 Key 需要不同的初始值 ** 的场景。

优势：

- ** 支持自定义初始化逻辑 **，比 `reduceByKey()` 更灵活。
- ** 适用于聚合复杂数据结构 **，如计算平均值（`(sum, count)`）。

—

2.5.3 面试常考问题

1. **`reduce()` 和 `reduceByKey()` 的区别？**
2. **为什么 `reduceByKey()` 比 `groupByKey()` 高效？**
3. **如何优化 Reduce 算子，避免数据倾斜？**
4. **什么情况下适合使用 `combineByKey()`？**
5. **如何调整 Shuffle 并行度来优化 Reduce 计算？**