

Spark 知识体系吐血总结

本文档来自公众号：**五分钟学大数据**

微信扫码关注



目录

Spark 涉及的知识点如下图所示，本文将逐一讲解：	5
一、Spark 基础.....	6
1. 激动人心的 Spark 发展史.....	6
2. Spark 为什么会流行.....	7
3. Spark VS Hadoop.....	8
3. Spark 特点.....	9
4. Spark 运行模式.....	10
二、Spark Core.....	10
1. RDD 详解.....	10
2. RDD-API.....	13
3. RDD 的持久化/缓存.....	17
4. RDD 容错机制 Checkpoint.....	19
5. RDD 依赖关系.....	20
6. DAG 的生成和划分 Stage.....	21
7. RDD 累加器和广播变量.....	23
三、Spark SQL.....	26
1. 数据分析方式.....	26
2. SparkSQL 前世今生.....	28
3. Hive 和 SparkSQL.....	29
4. 数据分类和 SparkSQL 适用场景.....	30
5. Spark SQL 数据抽象.....	32
6. Spark SQL 应用.....	35
四、Spark Streaming.....	42
1. 整体流程.....	43
2. 数据抽象.....	43
3. DStream 相关操作.....	44
4. Spark Streaming 完成实时需求.....	46
五、Structured Streaming.....	51
1. API.....	52
2. 核心理念.....	53

3. 应用场景.....	53
4. Structured Streaming 实战.....	53
六、Spark 的两种核心 Shuffle.....	59
Spark Shuffle.....	59
一、Hash Shuffle 解析.....	61
二、SortShuffle 解析.....	65
七、Spark 底层执行原理.....	69
Spark 运行流程.....	69
Spark 运行架构特点.....	75
八、Spark 数据倾斜.....	77
1. 预聚合原始数据.....	78
2. 预处理导致倾斜的 key.....	79
3. 提高 reduce 并行度.....	81
4. 使用 map join.....	82
九、Spark 性能优化.....	84
Spark 调优之 RDD 算子调优.....	84
1. RDD 复用.....	84
2. 尽早 filter.....	85
3. 读取大量小文件-用 wholeTextFiles.....	85
4. mapPartition 和 foreachPartition.....	86
5. filter+coalesce/repartition(减少分区).....	88
6. 并行度设置.....	90
7. repartition/coalesce 调节并行度.....	91
8. reduceByKey 本地预聚合.....	92
9. 使用持久化+checkpoint.....	94
10. 使用广播变量.....	95
11. 使用 Kryo 序列化.....	96
Spark 调优之 Shuffle 调优.....	96
1. map 和 reduce 端缓冲区大小.....	96
2. reduce 端重试次数和等待时间间隔.....	97
3. bypass 机制开启阈值.....	98
十、故障排除.....	98

1. 避免 OOM-out of memory.....	98
2. 避免 GC 导致的 shuffle 文件拉取失败.....	99
3. YARN-CLIENT 模式导致的网卡流量激增问题.....	100
4. YARN-CLOUD 模式的 JVM 栈内存溢出无法执行问题.....	100
5. 避免 SparkSQL JVM 栈内存溢出.....	101
十一、Spark 大厂面试真题.....	101

Spark 涉及的知识点如下图所示，本文将逐一讲解：



本文档参考了关于 Spark 的众多资料整理而成，为了整洁的排版及舒适的阅读，对于模糊不清晰的图片及黑白图片进行重新绘制成了高清彩图。

一、Spark 基础

1. 激动人心的 Spark 发展史

大数据、人工智能(Artificial Intelligence)像当年的石油、电力一样，正以前所未有的广度和深度影响所有的行业，现在及未来公司的核心壁垒是数据，核心竞争力来自基于大数据的人工智能的竞争。

Spark 是当今大数据领域最活跃、最热门、最高效的大数据通用计算平台之一。

2009 年诞生于美国加州大学伯克利分校 AMP 实验室；

2010 年通过 BSD 许可协议开源发布；

2013 年捐赠给 Apache 软件基金会并切换开源协议到切换许可协议至 Apache2.0；

2014 年 2 月，Spark 成为 Apache 的顶级项目；

2014 年 11 月，Spark 的母公司 Databricks 团队使用 Spark 刷新数据排序世界记录。

Spark 成功构建起了一体化、多元化的大数据处理体系。在任何规模的数据计算中，Spark 在性能和扩展性上都更具优势。

1. Hadoop 之父 Doug Cutting 指出：Use of MapReduce engine for Big Data projects will decline, replaced by Apache Spark (大数据项目的 MapReduce 引擎的使用将下降，由 Apache Spark 取代)。
2. Hadoop 商业发行版本的市场领导者 Cloudera 、HortonWorks 、MapR 纷纷转投 Spark, 并把 Spark 作为大数据解决方案的首选和核心计算引擎。

2014 年的 Benchmark 测试中，Spark 秒杀 Hadoop，在使用十分之一计算资源的情况下，相同数据的排序上，Spark 比 MapReduce 快 3 倍！在没有官方 PB 排序对比的情况下，首次将 Spark 推到了 IPB 数据(十万亿条记录)的排序，在使用 190 个节点的情况下，工作负载在 4 小时内完成，同样远超雅虎之前使用 3800 台主机耗时 16 个小时的记录。

在 FullStack 理想的指引下，**Spark 中的 Spark SQL 、SparkStreaming 、MLLib 、GraphX 、R 五大子框架和库之间可以无缝地共享数据和操作**，这不仅打造了 Spark 在当今大数据计算领域其他计算框架都无可匹敌的优势，而且使得 Spark 正在加速成为大数据处理中心首选通用计算平台。

2. Spark 为什么会流行

- 原因 1: 优秀的数据模型和丰富计算抽象

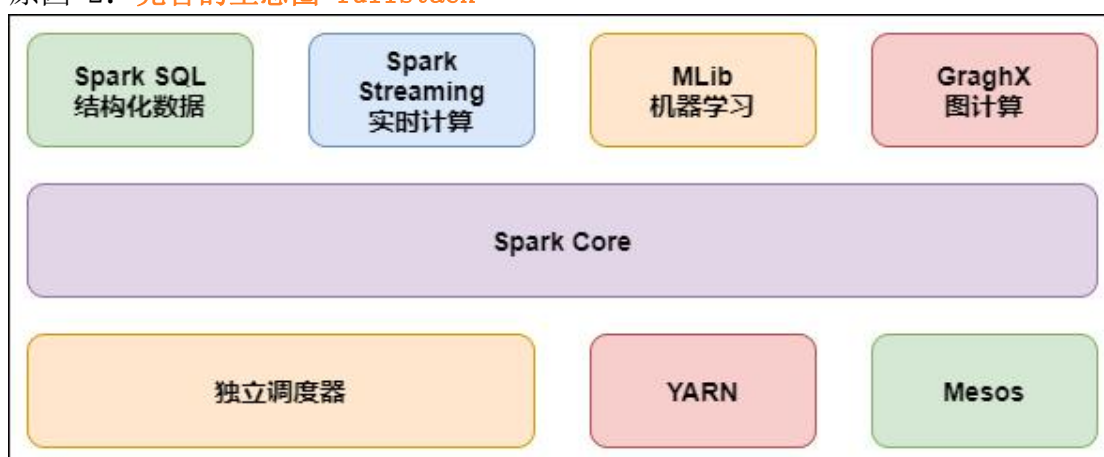
Spark 产生之前，已经有 MapReduce 这类非常成熟的计算系统存在了，并提供了高层次的 API (map/reduce)，把计算运行在集群中并提供容错能力，从而实现分布式计算。

虽然 MapReduce 提供了对数据访问和计算的抽象，但是对于数据的复用就是简单的将中间数据写到一个稳定的文件系统中(例如 HDFS)，所以会产生数据的复制备份，磁盘的 I/O 以及数据的序列化，所以在遇到需要在多个计算之间复用中间结果的操作时效率就会非常的低。而这类操作是非常常见的，例如迭代式计算，交互式数据挖掘，图计算等。

认识到这个问题后，学术界的 AMPLab 提出了一个新的模型，叫做 RDD。RDD 是一个可以容错且并行的数据结构(其实可以理解成分布式的集合，操作起来和操作本地集合一样简单)，它可以让用户显式的将中间结果数据集保存在内存中，并且通过控制数据集的分区来达到数据存放处理最优化。同时 RDD 也提供了丰富的 API (map、reduce、filter、foreach、reducerByKey...)来操作数据集。后来 RDD 被 AMPLab 在一个叫做 Spark 的框架中提供并开源。

简而言之，Spark 借鉴了 MapReduce 思想发展而来，保留了其分布式并行计算的优点并改进了其明显的缺陷。让中间数据存储在内存在中提高了运行速度、并提供丰富的操作数据的 API 提高了开发速度。

- 原因 2: 完善的生态圈-fullstack



目前，Spark 已经发展成为一个包含多个子项目的集合，其中包含 SparkSQL、Spark Streaming、GraghX、MLlib 等子项目。

Spark Core: 实现了 Spark 的基本功能, 包含 RDD、任务调度、内存管理、错误恢复、与存储系统交互等模块。

Spark SQL: Spark 用来操作结构化数据的程序包。通过 Spark SQL, 我们可以使用 SQL 操作数据。

Spark Streaming: Spark 提供的对实时数据进行流式计算的组件。提供了用来操作数据流的 API。

Spark MLlib: 提供常见的机器学习(ML)功能的程序库。包括分类、回归、聚类、协同过滤等, 还提供了模型评估、数据导入等额外的支持功能。

GraphX(图计算): Spark 中用于图计算的 API, 性能良好, 拥有丰富的功能和运算符, 能在海量数据上自如地运行复杂的图算法。

集群管理器: Spark 设计为可以高效地在一个计算节点到数千个计算节点之间伸缩计算。

Structured Streaming: 处理结构化流, 统一了离线和实时的 API。

3. Spark VS Hadoop

	Hadoop	Spark
类型	分布式基础平台, 包含计算, 存储, 调度	分布式计算工具
场景	大规模数据集上的批处理	迭代计算, 交互式计算, 流计算
价格	对机器要求低, 便宜	对内存有要求, 相对较贵
编程范式	Map+Reduce, API 较为底层, 算法适用性差	RDD 组成 DAG 有向无环图, API 较为顶层, 方便使用
数据存储结构	MapReduce 中间计算结果存在 HDFS 磁盘上, 延迟大	RDD 中间运算结果存在内存中, 延迟小
运行方式	Task 以进程方式维护, 任务启动慢	Task 以线程方式维护, 任务启动快

💡 注意:

尽管 Spark 相对于 Hadoop 而言具有较大优势, 但 Spark 并不能完全替代 Hadoop, Spark 主要用于替代 Hadoop 中的 MapReduce 计算模型。存储依然可以使用 HDFS, 但是中间结果可以存放在内存中; 调度可以使用 Spark 内置的, 也可以使用更成熟的调度系统 YARN 等。

实际上, Spark 已经很好地融入了 Hadoop 生态圈, 并成为其中的重要一员, 它可

以借助于 YARN 实现资源调度管理，借助于 HDFS 实现分布式存储。

此外，Hadoop 可以使用廉价的、异构的机器来做分布式存储与计算，但是，Spark 对硬件的要求稍高一些，对内存与 CPU 有一定的要求。

3. Spark 特点

- 快

与 Hadoop 的 MapReduce 相比，Spark 基于内存的运算要快 100 倍以上，基于硬盘的运算也要快 10 倍以上。Spark 实现了高效的 DAG 执行引擎，可以通过基于内存来高效处理数据流。

- 易用

Spark 支持 Java、Python、R 和 Scala 的 API，还支持超过 80 种高级算法，使用户可以快速构建不同的应用。而且 Spark 支持交互式的 Python 和 Scala 的 shell，可以非常方便地在这些 shell 中使用 Spark 集群来验证解决问题的方法。

- 通用

Spark 提供了统一的解决方案。Spark 可以用于批处理、交互式查询(Spark SQL)、实时流处理(Spark Streaming)、机器学习(Spark MLlib)和图计算(GraphX)。这些不同类型的处理都可以在同一个应用中无缝使用。Spark 统一的解决方案非常具有吸引力，毕竟任何公司都想用统一的平台去处理遇到的问题，减少开发和维护的人力成本和部署平台的物力成本。

- 兼容性

Spark 可以非常方便地与其他开源产品进行融合。比如，Spark 可以使用 Hadoop 的 YARN 和 Apache Mesos 作为它的资源管理和调度器，并且可以处理所有 Hadoop 支持的数据，包括 HDFS、HBase 和 Cassandra 等。这对于已经部署 Hadoop 集群的用户特别重要，因为不需要做任何数据迁移就可以使用 Spark 的强大处理能力。Spark 也可以不依赖于第三方的资源管理和调度器，它实现了 Standalone 作为其内置的资源管理和调度框架，这样进一步降低了 Spark 的使

用门槛，使得所有人都可以非常容易地部署和使用 Spark。此外，Spark 还提供了在 EC2 上部署 Standalone 的 Spark 集群的工具。

4. Spark 运行模式

1. local 本地模式(单机)--学习测试使用
分为 local 单线程和 local-cluster 多线程。
2. standalone 独立集群模式--学习测试使用
典型的 Master/slave 模式。
3. standalone-HA 高可用模式--生产环境使用
基于 standalone 模式，使用 zk 搭建高可用，避免 Master 是有单点故障的。
4. on yarn 集群模式--生产环境使用
运行在 yarn 集群之上，由 yarn 负责资源管理，Spark 负责任务调度和计算。
好处：计算资源按需伸缩，集群利用率高，共享底层存储，避免数据跨集群迁移。
5. on mesos 集群模式--国内使用较少
运行在 mesos 资源管理器框架之上，由 mesos 负责资源管理，Spark 负责任务调度和计算。
6. on cloud 集群模式--中小公司未来会更多的使用云服务
比如 AWS 的 EC2，使用这个模式能很方便的访问 Amazon 的 S3。

二、Spark Core

1. RDD 详解

1) 为什么要有 RDD?

在许多迭代式算法(比如机器学习、图算法等)和交互式数据挖掘中，不同计算阶段之间会重用中间结果，即一个阶段的输出结果会作为下一个阶段的输入。但是，之前的 MapReduce 框架采用非循环式的数据流模型，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。且这些框架只能支持一些特定的计算模式(map/reduce)，并没有提供一种通用的数据抽象。

AMP 实验室发表的一篇关于 RDD 的论文:《Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing》就是为了解决这些问题的。

RDD 提供了一个抽象的数据模型,让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操作(函数),不同 RDD 之间的转换操作之间还可以形成依赖关系,进而实现管道化,从而避免了中间结果的存储,大大降低了数据复制、磁盘 IO 和序列化开销,并且还提供了更多的 API (map/reduce/filter/groupBy...)。

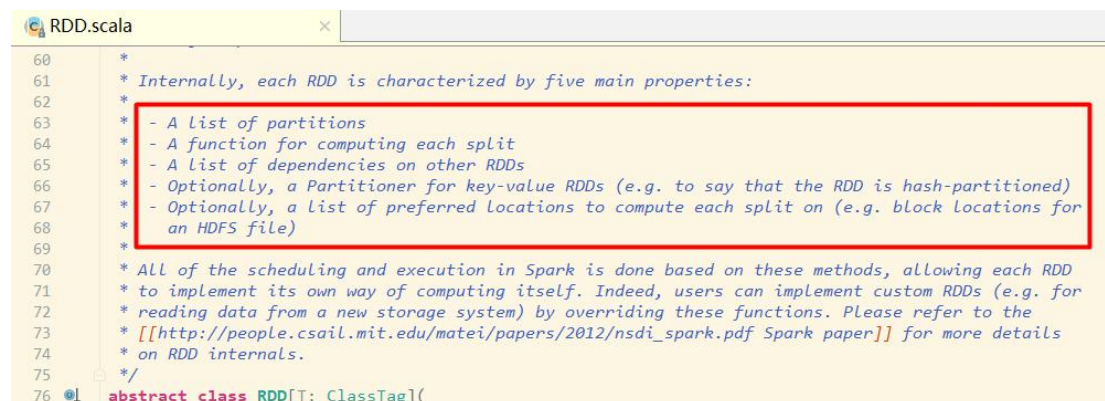
2) RDD 是什么?

RDD(Resilient Distributed Dataset)叫做弹性分布式数据集,是 Spark 中最基本的数据抽象,代表一个不可变、可分区、里面的元素可并行计算的集合。单词拆解:

- Resilient : 它是弹性的, RDD 里面的数据可以保存在内存中或者磁盘里面;
- Distributed : 它里面的元素是分布式存储的,可以用于分布式计算;
- Dataset: 它是一个集合,可以存放很多元素。

3) RDD 主要属性

进入 RDD 的源码中看下:



```

60  *
61  * Internally, each RDD is characterized by five main properties:
62  *
63  * - A list of partitions
64  * - A function for computing each split
65  * - A list of dependencies on other RDDs
66  * - Optionally, a Partitioner for key-value RDDs (e.g. to say that the RDD is hash-partitioned)
67  * - Optionally, a list of preferred locations to compute each split on (e.g. block locations for
68  *   an HDFS file)
69  *
70  * All of the scheduling and execution in Spark is done based on these methods, allowing each RDD
71  * to implement its own way of computing itself. Indeed, users can implement custom RDDs (e.g. for
72  * reading data from a new storage system) by overriding these functions. Please refer to the
73  * [[http://people.csail.mit.edu/matei/papers/2012/nsdi_spark.pdf Spark paper]] for more details
74  * on RDD internals.
75  */
76  abstract class RDD[T: ClassTag](

```

RDD 源码

在源码中可以看到有对 RDD 介绍的注释,我们来翻译下:

1. **A list of partitions** : 一组分片 (Partition)/一个分区 (Partition) 列表, 即数据集的基本组成单位。对于 RDD 来说, 每个分片都会被一个计算任务处理, 分片数决定并行度。用户可以在创建 RDD 时指定 RDD 的分片个数, 如果没有指定, 那么就会采用默认值。
2. **A function for computing each split** : 一个函数会被作用在每一个分区。Spark 中 RDD 的计算是以分片为单位的, compute 函数会被作用到每个分区上。
3. **A list of dependencies on other RDDs** : 一个 RDD 会依赖于其他多个 RDD。RDD 的每次转换都会生成一个新的 RDD, 所以 RDD 之间就会形成类似于流水线一样的前后依赖关系。在部分分区数据丢失时, Spark 可以通过这个依赖关系重新计算丢失的分区数据, 而不是对 RDD 的所有分区进行重新计算。(Spark 的容错机制)
4. **Optionally, a Partitioner for key-value RDDs (e.g. to say that the RDD is hash-partitioned)**: 可选项, 对于 KV 类型的 RDD 会有一个 Partitioner, 即 RDD 的分区函数, 默认为 HashPartitioner。
5. **Optionally, a list of preferred locations to compute each split on (e.g. block locations for an HDFS file)**: 可选项, 一个列表, 存储存取每个 Partition 的优先位置 (preferred location)。对于一个 HDFS 文件来说, 这个列表保存的就是每个 Partition 所在的块的位置。按照“移动数据不如移动计算”的理念, Spark 在进行任务调度的时候, 会尽可能选择那些存有数据的 worker 节点来进行任务计算。

总结

RDD 是一个数据集的表示, 不仅表示了数据集, 还表示了这个数据集从哪来, 如何计算, 主要属性包括:

1. 分区列表
2. 计算函数
3. 依赖关系
4. 分区函数 (默认是 hash)
5. 最佳位置

分区列表、分区函数、最佳位置, 这三个属性其实说的就是数据集在哪, 在哪计算更合适, 如何分区;

计算函数、依赖关系, 这两个属性其实说的是数据集怎么来的。

2. RDD-API

1) RDD 的创建方式

1. 由外部存储系统的数据集创建，包括本地的文件系统，还有所有 Hadoop 支持的数据集，比如 HDFS、Cassandra、HBase 等：

```
val rdd1 =  
sc.textFile("hdfs://node1:8020/wordcount/input/words.txt")
```

2. 通过已有的 RDD 经过算子转换生成新的 RDD：

```
val rdd2=rdd1.flatMap(_.split(" "))
```

3. 由一个已经存在的 Scala 集合创建：

```
val rdd3 = sc.parallelize(Array(1,2,3,4,5,6,7,8)) 或者  
val rdd4 = sc.makeRDD(List(1,2,3,4,5,6,7,8))
```

makeRDD 方法底层调用了 parallelize 方法：

```
def makeRDD[T: ClassTag](  
  seq: Seq[T],  
  numSlices: Int = defaultParallelism): RDD[T] = withScope {  
  parallelize(seq, numSlices)  
}
```

RDD 源码

2) RDD 的算子分类

RDD 的算子分为两类：

1. **Transformation** 转换操作：**返回一个新的 RDD**
2. **Action** 动作操作：**返回值不是 RDD (无返回值或返回其他的)**

💡 注意：

- 1、RDD 不实际存储真正要计算的数据，而是记录了数据的位置在哪里，数据的转换关系(调用了什么方法，传入什么函数)。
- 2、RDD 中的所有转换都是惰性求值/延迟执行的，也就是说并不会直接计算。只有当发生一个要求返回结果给 Driver 的 Action 动作时，这些转换才会真正运行。
- 3、之所以使用惰性求值/延迟执行，是因为这样可以在 Action 时对 RDD 操作形成

DAG 有向无环图进行 Stage 的划分和并行优化，这种设计让 Spark 更加有效率地运行。

3) Transformation 转换算子

转换算子	含义
<code>map(func)</code>	返回一个新的 RDD，该 RDD 由每一个输入元素经过 func 函数转换后组成
<code>filter(func)</code>	返回一个新的 RDD，该 RDD 由经过 func 函数计算后返回值为 true 的输入元素组成
<code>flatMap(func)</code>	类似于 map，但是每一个输入元素可以被映射为 0 或多个输出元素(所以 func 应该返回一个序列，而不是单一元素)
<code>mapPartitions(func)</code>	类似于 map，但独立地在 RDD 的每一个分片上运行，因此在类型为 T 的 RDD 上运行时，func 的函数类型必须是 <code>Iterator[T] => Iterator[U]</code>
<code>mapPartitionsWithIndex(func)</code>	类似于 mapPartitions，但 func 带有一个整数参数表示分片的索引值，因此在类型为 T 的 RDD 上运行时，func 的函数类型必须是 <code>(Int, Iterator[T]) => Iterator[U]</code>
<code>sample(withReplacement, fraction, seed)</code>	根据 fraction 指定的比例对数据进行采样，可以选择是否使用随机数进行替换，seed 用于指定随机数生成器种子
<code>union(otherDataset)</code>	对源 RDD 和参数 RDD 求并集后返回一个新的 RDD
<code>intersection(otherDataset)</code>	对源 RDD 和参数 RDD 求交集后返回一个新的 RDD
<code>distinct([numTasks])</code>	对源 RDD 进行去重后返回一个新的 RDD
<code>groupByKey([numTasks])</code>	在一个 (K, V) 的 RDD 上调用，返回一个 (K, Iterator[V]) 的 RDD
<code>reduceByKey(func, [numTasks])</code>	在一个 (K, V) 的 RDD 上调用，返回一个 (K, V) 的 RDD，使用指定的 reduce 函数，将相同 key 的值聚合到一起，与 groupByKey 类似，reduce 任

转换算子	含义
	任务的个数可以通过第二个可选的参数来设置
<code>aggregateByKey(zeroValue)(seqOp, combOp, [numTasks])</code>	对 PairRDD 中相同的 Key 值进行聚合操作, 在聚合过程中同样使用了一个中立的初始值。和 aggregate 函数类似, aggregateByKey 返回值的类型不需要和 RDD 中 value 的类型一致
<code>sortByKey([ascending], [numTasks])</code>	在一个 (K, V) 的 RDD 上调用, K 必须实现 Ordered 接口, 返回一个按照 key 进行排序的 (K, V) 的 RDD
<code>sortBy(func, [ascending], [numTasks])</code>	与 sortByKey 类似, 但是更灵活
<code>join(otherDataset, [numTasks])</code>	在类型为 (K, V) 和 (K, W) 的 RDD 上调用, 返回一个相同 key 对应的所有元素对在一起的 (K, (V, W)) 的 RDD
<code>cogroup(otherDataset, [numTasks])</code>	在类型为 (K, V) 和 (K, W) 的 RDD 上调用, 返回一个 (K, (Iterable, Iterable)) 类型的 RDD
<code>cartesian(otherDataset)</code>	笛卡尔积
<code>pipe(command, [envVars])</code>	对 rdd 进行管道操作
<code>coalesce(numPartitions)</code>	减少 RDD 的分区数到指定值。在过滤大量数据之后, 可以执行此操作
<code>repartition(numPartitions)</code>	重新给 RDD 分区

4) Action 动作算子

动作算子	含义
<code>reduce(func)</code>	通过 func 函数聚集 RDD 中的所有元素, 这个功能必须是可交换且可并联的
<code>collect()</code>	在驱动程序中, 以数组的形式返回数据集的所有元素
<code>count()</code>	返回 RDD 的元素个数
<code>first()</code>	返回 RDD 的第一个元素 (类似于 take(1))

动作算子	含义
<code>take(n)</code>	返回一个由数据集的前 <code>n</code> 个元素组成的数组
<code>takeSample(withReplacement, num, [seed])</code>	返回一个数组，该数组由从数据集中随机采样的 <code>num</code> 个元素组成，可以选择是否用随机数替换不足的部分， <code>seed</code> 用于指定随机数生成器种子
<code>takeOrdered(n, [ordering])</code>	返回自然顺序或者自定义顺序的前 <code>n</code> 个元素
<code>saveAsTextFile(path)</code>	将数据集的元素以 <code>textfile</code> 的形式保存到 HDFS 文件系统或者其他支持的文件系统，对于每个元素，Spark 将会调用 <code>toString</code> 方法，将它转换为文件中的文本
<code>saveAsSequenceFile(path)</code>	将数据集中的元素以 Hadoop <code>sequencefile</code> 的格式保存到指定的目录下，可以使 HDFS 或者其他 Hadoop 支持的文件系统
<code>saveAsObjectFile(path)</code>	将数据集的元素，以 Java 序列化的方式保存到指定的目录下
<code>countByKey()</code>	针对 <code>(K, V)</code> 类型的 RDD，返回一个 <code>(K, Int)</code> 的 map，表示每一个 <code>key</code> 对应的元素个数
<code>foreach(func)</code>	在数据集的每一个元素上，运行函数 <code>func</code> 进行更新
<code>foreachPartition(func)</code>	在数据集的每一个分区上，运行函数 <code>func</code>

统计操作：

算子	含义
<code>count</code>	个数
<code>mean</code>	均值
<code>sum</code>	求和
<code>max</code>	最大值
<code>min</code>	最小值
<code>variance</code>	方差
<code>sampleVariance</code>	从采样中计算方差
<code>stdev</code>	标准差:衡量数据的离散程度

算子	含义
sampleStdev	采样的标准差
stats	查看统计结果

4) RDD 算子练习

- 需求：

给定一个键值对 RDD：

```
val rdd = sc.parallelize(Array(("spark",2),("hadoop",6),("hadoop",4),("spark",6)))
```

key 表示图书名称，value 表示某天图书销量

请计算每个键对应的平均值，也就是计算每种图书的每天平均销量。

最终结果：("spark", 4), ("hadoop", 5)。

- 答案 1：

```
val rdd = sc.parallelize(Array(("spark",2),("hadoop",6),("hadoop",4),("spark",6)))
val rdd2 = rdd.groupByKey()
rdd2.collect
//Array[(String, Iterable[Int])] = Array((spark,CompactBuffer(2, 6)), (hadoop,CompactBuffer(6, 4)))
rdd2.mapValues(v=>v.sum/v.size).collect
Array[(String, Int)] = Array((spark,4), (hadoop,5))
```

- 答案 2：

```
val rdd = sc.parallelize(Array(("spark",2),("hadoop",6),("hadoop",4),("spark",6)))
val rdd2 = rdd.groupByKey()
rdd2.collect
//Array[(String, Iterable[Int])] = Array((spark,CompactBuffer(2, 6)), (hadoop,CompactBuffer(6, 4)))

val rdd3 = rdd2.map(t=>(t._1,t._2.sum /t._2.size))
rdd3.collect
//Array[(String, Int)] = Array((spark,4), (hadoop,5))
```

3. RDD 的持久化/缓存

在实际开发中某些 RDD 的计算或转换可能会比较耗费时间,如果这些 RDD 后续还会频繁的被使用到,那么可以将这些 RDD 进行持久化/缓存,这样下次再使用到的时候就不用再重新计算了,提高了程序运行的效率。

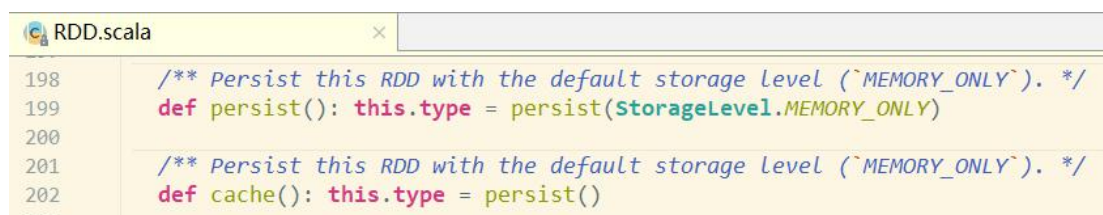
```
val rdd1 = sc.textFile("hdfs://node01:8020/words.txt")
val rdd2 = rdd1.flatMap(x=>x.split(" ")).map((_,1)).reduceByKey(_+_ )
rdd2.cache //缓存/持久化
rdd2.sortBy(_._2,false).collect//触发action,会去读取HDFS的文件,rdd2会真正执行持久化
rdd2.sortBy(_._2,false).collect//触发action,会去读缓存中的数据,执行速度会比之前快,因为rdd2已经持久化到内存中了
```

持久化/缓存 API 详解

- persist 方法和 cache 方法

RDD 通过 persist 或 cache 方法可以将前面的计算结果缓存,但是并不是这两个方法被调用时立即缓存,而是触发后面的 action 时,该 RDD 将会被缓存在计算节点的内存中,并供后面重用。

通过查看 RDD 的源码发现 cache 最终也是调用了 persist 无参方法(默认存储只存在内存中):



```
RDD.scala
198 /** Persist this RDD with the default storage level (`MEMORY_ONLY`). */
199 def persist(): this.type = persist(StorageLevel.MEMORY_ONLY)
200
201 /** Persist this RDD with the default storage level (`MEMORY_ONLY`). */
202 def cache(): this.type = persist()
```

RDD 源码

- 存储级别

默认的存储级别都是仅在内存存储一份,Spark 的存储级别还有好多种,存储级别在 object StorageLevel 中定义的。

持久化级别	说明
MORY_ONLY(默认)	将 RDD 以非序列化的 Java 对象存储在 JVM 中。如果没有足够的内存存储 RDD,则某些分区将不会被缓存,每次需要时都会重新计算。这是默认级别
MORY_AND_DISK(开发中可以使用这个)	将 RDD 以非序列化的 Java 对象存储在 JVM 中。如果数据在内存中放不下,则溢写到磁盘上。需要时则会从磁盘上读

持久化级别	说明
	取
MEMORY_ONLY_SER (Java and Scala)	将 RDD 以序列化的 Java 对象(每个分区一个字节数组)的方式存储。这通常比非序列化对象(deserialized objects)更具空间效率，特别是在使用快速序列化的情况下，但是这种方式读取数据会消耗更多的 CPU
MEMORY_AND_DISK_SER (Java and Scala)	与 MEMORY_ONLY_SER 类似，但如果数据在内存中放不下，则溢写到磁盘上，而不是每次需要重新计算它们
DISK_ONLY	将 RDD 分区存储在磁盘上
MEMORY_ONLY_2, MEMORY_AND_DISK_2 等	与上面的储存级别相同，只不过将持久化数据存为两份，备份每个分区存储在两个集群节点上
OFF_HEAP(实验中)	与 MEMORY_ONLY_SER 类似，但将数据存储在堆外内存中。(即不是直接存储在 JVM 内存中)

总结：

1. RDD 持久化/缓存的目的是为了提高后续操作的速度
2. 缓存的级别有很多，默认只存在内存中, 开发中使用 `memory_and_disk`
3. 只有执行 `action` 操作的时候才会真正将 RDD 数据进行持久化/缓存
4. 实际开发中如果某一个 RDD 后续会被频繁的使用, 可以将该 RDD 进行持久化/缓存

4. RDD 容错机制 Checkpoint

持久化的局限：

持久化/缓存可以把数据放在内存中，虽然是快速的，但是也是最不可靠的；也可以把数据放在磁盘上，也不是完全可靠的！例如磁盘会损坏等。

问题解决：

Checkpoint 的产生就是为了更加可靠的数据持久化，在 Checkpoint 的时候一般把数据放在在 HDFS 上，这就天然的借助了 HDFS 天生的高容错、高可靠来实现数据最大程度上的安全，实现了 RDD 的容错和高可用。

用法：

```
SparkContext.setCheckpointDir("目录") //HDFS 的目录
```

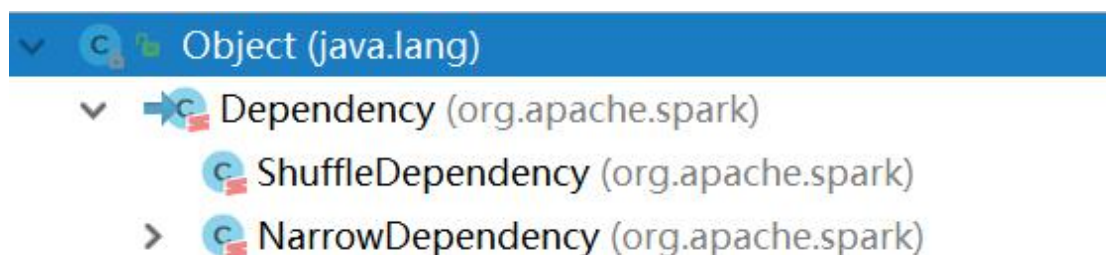
```
RDD.checkpoint
```

- **总结：**
- **开发中如何保证数据的安全性及读取效率：** 可以对频繁使用且重要的数据，先做缓存/持久化，再做 checkpoint 操作。
- **持久化和 Checkpoint 的区别：**
 1. 位置： Persist 和 Cache 只能保存在本地的磁盘和内存中(或者堆外内存--实验中) Checkpoint 可以保存数据到 HDFS 这类可靠的存储上。
 2. 生命周期： Cache 和 Persist 的 RDD 会在程序结束后会被清除或者手动调用 unpersist 方法 Checkpoint 的 RDD 在程序结束后依然存在，不会被删除。

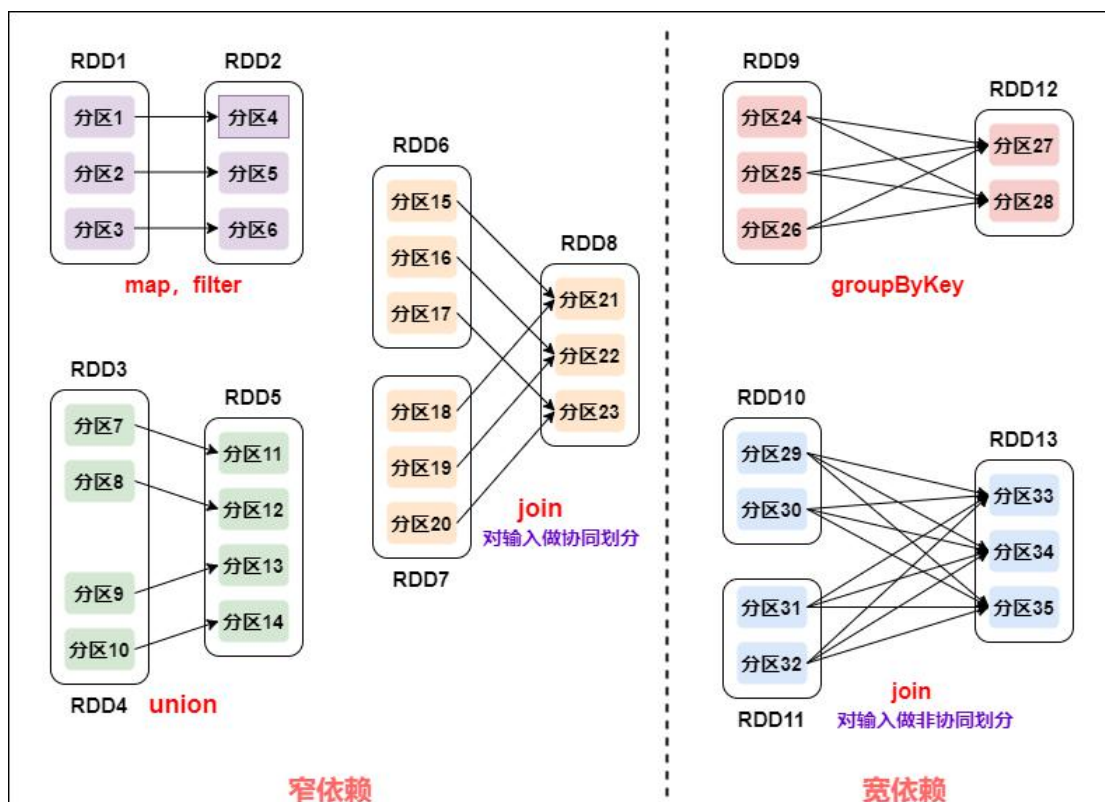
5. RDD 依赖关系

1) 宽窄依赖

- **两种依赖关系类型：** RDD 和它依赖的父 RDD 的关系有两种不同的类型，即 **宽依赖**(wide dependency/shuffle dependency) **窄依赖**(narrow dependency)



- **图解：**



宽窄依赖

- 如何区分宽窄依赖:

窄依赖:父 RDD 的一个分区只会被子 RDD 的一个分区依赖;

宽依赖:父 RDD 的一个分区会被子 RDD 的多个分区依赖(涉及到 shuffle)。

2) 为什么要设计宽窄依赖

1. 对于窄依赖:

窄依赖的多个分区可以并行计算;

窄依赖的一个分区的数据如果丢失只需要重新计算对应的分区的数据就可以了。

2. 对于宽依赖:

划分 Stage(阶段)的依据:对于宽依赖,必须等到上一阶段计算完成才能计算下一阶段。

6. DAG 的生成和划分 Stage

1) DAG 介绍

- DAG 是什么：

DAG (Directed Acyclic Graph 有向无环图) 指的是数据转换执行的过程，有方向，无闭环 (其实就是 RDD 执行的流程)；

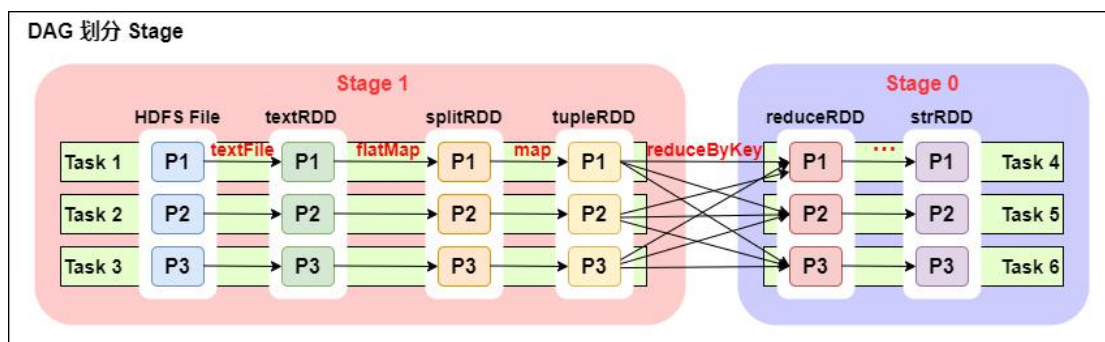
原始的 RDD 通过一系列的转换操作就形成了 DAG 有向无环图，任务执行时，可以按照 DAG 的描述，执行真正的计算 (数据被操作的一个过程)。

- DAG 的边界

开始:通过 SparkContext 创建的 RDD;

结束:触发 Action，一旦触发 Action 就形成了一个完整的 DAG。

2) DAG 划分 Stage



DAG 划分 Stage

一个 Spark 程序可以有多个 DAG (有几个 Action，就有几个 DAG，上图最后只有一个 Action (图中未表现)，那么就是一个 DAG)。

一个 DAG 可以有多个 Stage (根据宽依赖/shuffle 进行划分)。

同一个 Stage 可以有多个 Task 并行执行 (task 数=分区数，如上图，Stage1 中有三个分区 P1、P2、P3，对应的也有三个 Task)。

可以看到这个 DAG 中只 reduceByKey 操作是一个宽依赖，Spark 内核会以此为边界将其前后划分成不同的 Stage。

同时我们可以注意到，在图中 Stage1 中，从 textFile 到 flatMap 到 map 都是窄依赖，这几步操作可以形成一个流水线操作，通过 flatMap 操作生成的

partition 可以不用等待整个 RDD 计算结束，而是继续进行 map 操作，这样大大提高了计算的效率。

- 为什么要划分 Stage? --并行计算

一个复杂的业务逻辑如果有 shuffle，那么就意味着前面阶段产生结果后，才能执行下一个阶段，即下一个阶段的计算要依赖上一个阶段的数据。那么我们按照 shuffle 进行划分(也就是按照宽依赖就行划分)，就可以将一个 DAG 划分成多个 Stage/阶段，在同一个 Stage 中，会有多个算子操作，可以形成一个 pipeline 流水线，流水线内的多个平行的分区可以并行执行。

- 如何划分 DAG 的 stage?

对于窄依赖，partition 的转换处理在 stage 中完成计算，不划分(将窄依赖尽量放在在同一个 stage 中，可以实现流水线计算)。

对于宽依赖，由于有 shuffle 的存在，只能在父 RDD 处理完成后，才能开始接下来的计算，也就是说需要划分 stage。

总结：

Spark 会根据 shuffle/宽依赖使用回溯算法来对 DAG 进行 Stage 划分，从后往前，遇到宽依赖就断开，遇到窄依赖就把当前的 RDD 加入到当前的 stage/阶段中

具体的划分算法请参见 AMP 实验室发表的论文：《Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing》
http://xueshu.baidu.com/usercenter/paper/show?paperid=b33564e60f0a7e7a1889a9da10963461&site=xueshu_se

7. RDD 累加器和广播变量

在默认情况下，当 Spark 在集群的多个不同节点的多个任务上并行运行一个函数时，它会把函数中涉及到的每个变量，在每个任务上都生成一个副本。但是，有时候需要在多个任务之间共享变量，或者在任务(Task)和任务控制节点(Driver Program)之间共享变量。

为了满足这种需求，Spark 提供了两种类型的变量：

1. **累加器 accumulators**：累加器支持在所有不同节点之间进行累加计算(比如计数或者求和)。

2. **广播变量 broadcast variables**: 广播变量用来把变量在所有节点的内存之间进行共享，在每个机器上缓存一个只读的变量，而不是为机器上的每个任务都生成一个副本。

1) 累加器

1. 不使用累加器

```
var counter = 0
val data = Seq(1, 2, 3)
data.foreach(x => counter += x)
println("Counter value: " + counter)
```

运行结果:

```
Counter value: 6
```

如果我们将 data 转换成 RDD，再来重新计算:

```
var counter = 0
val data = Seq(1, 2, 3)
var rdd = sc.parallelize(data)
rdd.foreach(x => counter += x)
println("Counter value: " + counter)
```

运行结果:

```
Counter value: 0
```

2. 使用累加器

通常在向 Spark 传递函数时，比如使用 map() 函数或者用 filter() 传条件时，可以使用驱动器程序中定义的变量，但是集群中运行的每个任务都会得到这些变量的一份新的副本，更新这些副本的值也不会影响驱动器中的对应变量。这时使用累加器就可以实现我们想要的效果:

```
val xx: Accumulator[Int] = sc.accumulator(0)
```

3. 代码示例:

```
import org.apache.spark.rdd.RDD
import org.apache.spark.{Accumulator, SparkConf, SparkContext}

object AccumulatorTest {
  def main(args: Array[String]): Unit = {
    val conf: SparkConf = new SparkConf().setAppName("wc").setMaster("local[*]")
```



```

val sc: SparkContext = new SparkContext(conf)
sc.setLogLevel("WARN")

//使用scala 集合完成累加
var counter1: Int = 0;
var data = Seq(1,2,3)
data.foreach(x => counter1 += x )
println(counter1)//6

println("+++++")

//使用RDD 进行累加
var counter2: Int = 0;
val dataRDD: RDD[Int] = sc.parallelize(data) //分布式集合的[1,2,3]
dataRDD.foreach(x => counter2 += x)
println(counter2)//0
//注意: 上面的RDD 操作运行结果是0
//因为foreach 中的函数是传递给Worker 中的Executor 执行,用到了counter2 变量
//而counter2 变量在Driver 端定义的,在传递给Executor 的时候,各个Executor 都有了一份counter2
//最后各个Executor 将各自个x 加到自己的counter2 上面了,和Driver 端的counter2 没有关系

//那这个问题得解决啊! 不能因为使用了Spark 连累加都做不了了啊!
//如果解决?--- 使用累加器
val counter3: Accumulator[Int] = sc.accumulator(0)
dataRDD.foreach(x => counter3 += x)
println(counter3)//6
}
}

```

2) 广播变量

1. 不使用广播变量
2. 使用广播变量
3. 代码示例:

关键词: `sc.broadcast()`

```

import org.apache.spark.broadcast.Broadcast
import org.apache.spark.rdd.RDD
import org.apache.spark.{SparkConf, SparkContext}

object BroadcastVariablesTest {

```

```
def main(args: Array[String]): Unit = {
    val conf: SparkConf = new SparkConf().setAppName("wc").setMaster("local[*]")
    val sc: SparkContext = new SparkContext(conf)
    sc.setLogLevel("WARN")

    //不使用广播变量
    val kvFruit: RDD[(Int, String)] = sc.parallelize(List((1,"apple"),(2,"orange"),
    (3,"banana"),(4,"grape")))
    val fruitMap: collection.Map[Int, String] =kvFruit.collectAsMap
    //scala.collection.Map[Int,String] = Map(2 -> orange, 4 -> grape, 1 -> apple, 3
    -> banana)
    val fruitIds: RDD[Int] = sc.parallelize(List(2,4,1,3))
    //根据水果编号取水果名称
    val fruitNames: RDD[String] = fruitIds.map(x=>fruitMap(x))
    fruitNames.foreach(println)
    //注意:以上代码看似一点问题没有,但是考虑到数据量如果较大,且Task 数较多,
    //那么会导致,被各个Task 共用到的fruitMap 会被多次传输
    //应该要减少fruitMap 的传输,一台机器上一个,被该台机器中的Task 共用即可
    //如何做到?--- 使用广播变量
    //注意:广播变量的值不能被修改,如需修改可以将数据存到外部数据源,如MySQL、Redis
    println("=====")
    val BroadcastFruitMap: Broadcast[collection.Map[Int, String]] = sc.broadcast(fr
    uitMap)
    val fruitNames2: RDD[String] = fruitIds.map(x=>BroadcastFruitMap.value(x))
    fruitNames2.foreach(println)

}
}
```

三、Spark SQL

1. 数据分析方式

1) 命令式

在前面的 RDD 部分，非常明显可以感觉的到是命令式的，主要特征是通过一个算子，可以得到一个结果，通过结果再进行后续计算。

```
sc.textFile("../")
    .flatMap(_.split(" "))
    .map((_, 1))
```

```
.reduceByKey(_ + _)  
.collect()
```

1. 命令式的优点

- 操作粒度更细，能够控制数据的每一个处理环节；
- 操作更明确，步骤更清晰，容易维护；
- 支持半/非结构化数据的操作。

2. 命令式的缺点

- 需要一定的代码功底；
- 写起来比较麻烦。

2) SQL

对于一些数据科学家/数据库管理员/DBA，要求他们为了做一个非常简单的查询，写一大堆代码，明显是一件非常残忍的事情，所以 SQL on Hadoop 是一个非常重要的方向。

```
SELECT  
  name,  
  age,  
  school  
FROM students  
WHERE age > 10
```

1. SQL 的优点

表达非常清晰，比如说这段 SQL 明显就是为了查询三个字段，条件是查询年龄大于 10 岁的。

2. SQL 的缺点

- 试想一下 3 层嵌套的 SQL 维护起来应该挺力不从心的吧；
- 试想一下如果使用 SQL 来实现机器学习算法也挺为难的吧。

3) 总结

SQL 擅长数据分析和通过简单的语法表示查询，命令式操作适合过程式处理和算法性的处理。

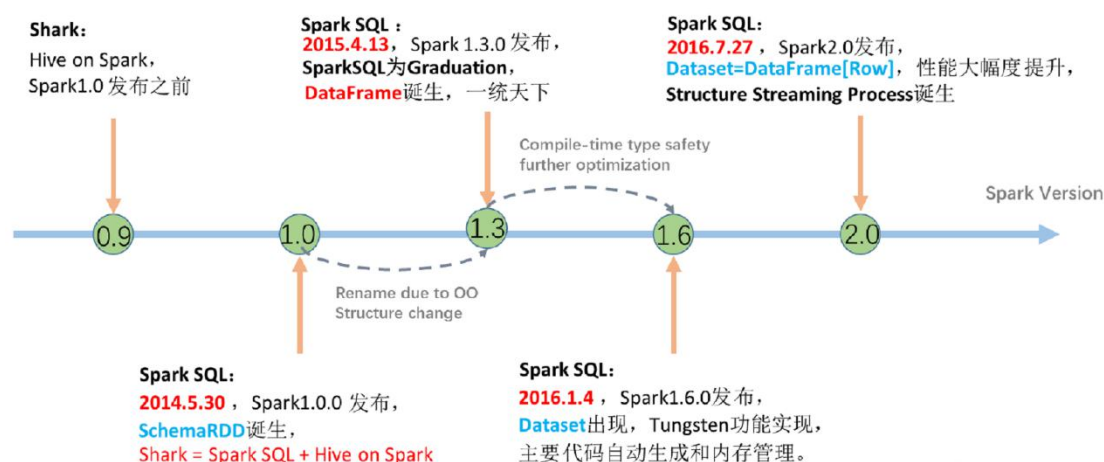
在 Spark 出现之前，对于结构化数据的查询和处理，一个工具一向只能支持 SQL 或者命令式，使用者被迫要使用多个工具来适应两种场景，并且多个工具配合起来比较费劲。

而 Spark 出现了以后，统一了两种数据处理范式是一种革新性的进步。

2. SparkSQL 前世今生

SQL 是数据分析领域一个非常重要的范式，所以 Spark 一直想要支持这种范式，而伴随着一些决策失误，这个过程其实还是非常曲折的。

1) 发展历史



- **Hive**

解决的问题:

Hive 实现了 SQL on Hadoop，使用 MapReduce 执行任务 简化了 MapReduce 任务。

新的问题:

Hive 的查询延迟比较高，原因是使用 MapReduce 做计算。

- **Shark**

解决的问题:

Shark 改写 Hive 的物理执行计划，使用 Spark 代替 MapReduce 物理引擎 使用列式内存存储。以上两点使得 Shark 的查询效率很高。

新的问题：

Shark 执行计划的生成严重依赖 Hive，想要增加新的优化非常困难；

Hive 是进程级别的并行，Spark 是线程级别的并行，所以 Hive 中很多线程不安全的代码不适用于 Spark；

由于以上问题，Shark 维护了 Hive 的一个分支，并且无法合并进主线，难以为继；

在 2014 年 7 月 1 日的 Spark Summit 上，Databricks 宣布终止对 Shark 的开发，将重点放到 Spark SQL 上。

- **SparkSQL-DataFrame**

解决的问题：

Spark SQL 执行计划和优化交给优化器 Catalyst；

内建了一套简单的 SQL 解析器，可以不使用 HQL；

还引入和 DataFrame 这样的 DSL API，完全可以不依赖任何 Hive 的组件

新的问题：

对于初期版本的 SparkSQL，依然有挺多问题，例如只能支持 SQL 的使用，不能很好的兼容命令式，入口不够统一等。

- **SparkSQL-Dataset**

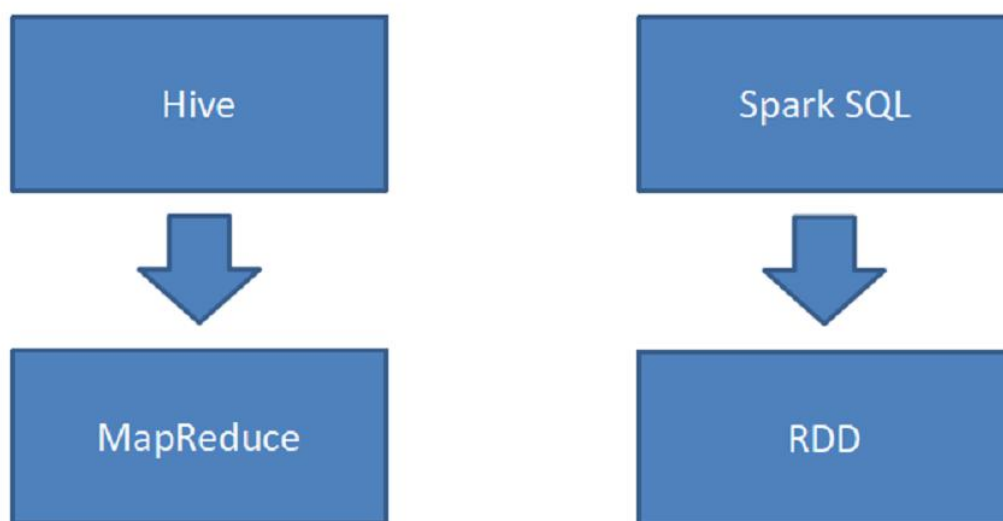
SparkSQL 在 1.6 时代，增加了一个新的 API，叫做 Dataset，Dataset 统一和结合了 SQL 的访问和命令式 API 的使用，这是一个划时代的进步。

在 Dataset 中可以轻易的做到使用 SQL 查询并且筛选数据，然后使用命令式 API 进行探索式分析。

3. Hive 和 SparkSQL

Hive 是将 SQL 转为 MapReduce。

SparkSQL 可以理解成是将 SQL 解析成：“RDD + 优化” 再执行。



4. 数据分类和 SparkSQL 适用场景

1) 结构化数据

一般指数据有固定的 **Schema** (约束), 例如在用户表中, name 字段是 String 型, 那么每一条数据的 name 字段值都可以当作 String 来使用:

id	name	url	alexa	country
1	Google	https://www.google.cm/	1	USA
2	淘宝	https://www.taobao.com/	13	CN
3	菜鸟教程	https://www.runoob.com/	4689	CN
4	微博	http://weibo.com/	20	CN
5	Facebook	https://www.facebook.com/	3	USA

2) 半结构化数据

般指的是数据没有固定的 Schema, 但是数据本身是有结构的。

- 没有固定 Schema

指的是半结构化数据是没有固定的 Schema 的，可以理解为没有显式指定 Schema。

比如说一个用户信息的 JSON 文件，

第 1 条数据的 phone_num 有可能是数字，

第 2 条数据的 phone_num 虽说应该也是数字，但是如果指定为 String，也是可以的，

因为没有指定 Schema，没有显式的强制的约束。

- 有结构

虽说半结构化数据是没有显式指定 Schema 的，也没有约束，但是半结构化数据本身是有有隐式的结构的，也就是数据自身可以描述自身。

例如 JSON 文件，其中的某一条数据是有字段这个概念的，每个字段也有类型的概念，所以说 JSON 是可以描述自身的，也就是数据本身携带有元信息。

3) 总结

- 数据分类总结：

•	定义	特点	举例
结构化数据	有固定的 Schema	有预定义的 Schema	关系型数据库的表
半结构化数据	没有固定的 Schema，但是有结构	没有固定的 Schema，有结构信息，数据一般是自描述的	指一些有结构的文件格式，例如 JSON
非结构化数据	没有固定 Schema，也没有结构	没有固定 Schema，也没有结构	指图片/音频之类的格式

- Spark 处理什么样的数据？

RDD 主要用于处理非结构化数据、半结构化数据、结构化；

SparkSQL 主要用于处理结构化数据(较为规范的半结构化数据也可以处理)。

- 总结：

SparkSQL 是一个既支持 SQL 又支持命令式数据处理的工具；

SparkSQL 的主要适用场景是处理结构化数据(较为规范的半结构化数据也可以处理)。

5. Spark SQL 数据抽象

1) DataFrame

- 什么是 DataFrame

DataFrame 的前身是 SchemaRDD, 从 Spark 1.3.0 开始 SchemaRDD 更名为 DataFrame。并不再直接继承自 RDD, 而是自己实现了 RDD 的绝大多数功能。DataFrame 是一种以 RDD 为基础的分布式数据集, 类似于传统数据库的二维表格, 带有 Schema 元信息(可以理解为数据库的列名和类型)。

- 总结:

DataFrame 就是一个分布式的表;

DataFrame = RDD - 泛型 + SQL 的操作 + 优化。

2) DataSet

- DataSet:

DataSet 是在 Spark1.6 中添加的新的接口。

与 RDD 相比, 保存了更多的描述信息, 概念上等同于关系型数据库中的二维表。与 DataFrame 相比, 保存了类型信息, 是强类型的, 提供了编译时类型检查。调用 Dataset 的方法先会生成逻辑计划, 然后被 spark 的优化器进行优化, 最终生成物理计划, 然后提交到集群中运行!

DataSet 包含了 DataFrame 的功能。

Spark2.0 中两者统一, DataFrame 表示为 DataSet[Row], 即 DataSet 的子集。

DataFrame 其实就是 DataSet[Row]:

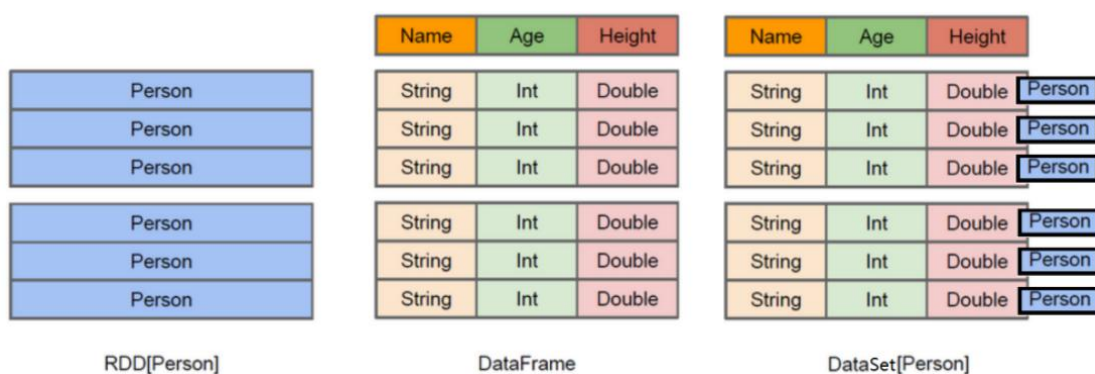

```
package object sql {

  /**
   * Converts a logical plan into zero or
   * with the query planner and is not de
   * writing libraries should instead con
   * [[org.apache.spark.sql.sources]]
   */
  @DeveloperApi
  @InterfaceStability.Unstable
  type Strategy = SparkStrategy

  type DataFrame = Dataset[Row]
}
```

3) RDD、DataFrame、DataSet 的区别

1. 结构图解：



- **RDD[Person]:**
以 Person 为类型参数，但不了解 其内部结构。
- **DataFrame:**
提供了详细的结构信息 schema 列的名称和类型。这样看起来就像一张表了。
- **DataSet[Person]**

不光有 schema 信息，还有类型信息。

2. 数据图解：

- 假设 RDD 中的两行数据长这样：

- `RDD[Person]`：

1,	张三,	23
2,	李四,	35

- 那么 DataFrame 中的数据长这样：

`DataFrame = RDD[Person] - 泛型 + Schema + SQL 操作 + 优化：`

ID:String	Name:String	Age:int
1	张三	23
2	李四	35

- 那么 Dataset 中的数据长这样：

`Dataset[Person] = DataFrame + 泛型：`

value:People[age: bigint, id: bigint, name:string]
People(id=1, name="张三", age=23)
People(id=1, name="李四", age=35)

- Dataset 也可能长这样:`Dataset[Row]`：

即 `DataFrame = DataSet[Row]`：

value:String
1, 张三, 23
2, 李四, 35

4) 总结

DataFrame = RDD - 泛型 + Schema + SQL + 优化

DataSet = DataFrame + 泛型

DataSet = RDD + Schema + SQL + 优化

6. Spark SQL 应用

- 在 spark2.0 版本之前
SQLContext 是创建 DataFrame 和执行 SQL 的入口。
HiveContext 通过 hive sql 语句操作 hive 表数据，兼容 hive 操作，
hiveContext 继承自 SQLContext。
- 在 spark2.0 之后
这些都统一于 SparkSession, SparkSession 封装了 SqlContext 及
HiveContext;
实现了 SQLContext 及 HiveContext 所有功能;
通过 SparkSession 还可以获取到 SparkConetxt。

1) 创建 DataFrame/DataSet

- 读取文本文件:
 1. 在本地创建一个文件，有 id、name、age 三列，用空格分隔，然后上传到 hdfs 上。

```
vim /root/person.txt
```

```
1 zhangsan 20
2 lisi 29
3 wangwu 25
4 zhaoliu 30
5 tianqi 35
6 kobe 40
```

2. 打开 spark-shell

```
spark/bin/spark-shell
```

创建 RDD

```
val lineRDD= sc.textFile("hdfs://node1:8020/person.txt").map(_.split(" "))
//RDD[Array[String]]
```

3. 定义 case class(相当于表的 schema)

```
case class Person(id:Int, name:String, age:Int)
```

4. 将 RDD 和 case class 关联 val personRDD = lineRDD.map(x => Person(x(0).toInt, x(1), x(2).toInt)) //RDD[Person]

5. 将 RDD 转换成 DataFrame

```
val personDF = personRDD.toDF //DataFrame
```

6. 查看数据和 schema

```
personDF.show
```

```
+---+-----+---+
| id|   name|age|
+---+-----+---+
|  1|zhangsan| 20|
|  2|   lisi| 29|
|  3| wangwu| 25|
|  4| zhaoliu| 30|
|  5|  tianqi| 35|
|  6|   kobe| 40|
+---+-----+---+
```

```
personDF.printSchema
```

7. 注册表

```
personDF.createOrReplaceTempView("t_person")
```

8. 执行 SQL

```
spark.sql("select id,name from t_person where id > 3").show
```

9. 也可以通过 SparkSession 构建 DataFrame

```
val dataframe=spark.read.text("hdfs://node1:8020/person.txt")
dataframe.show //注意: 直接读取的文本文件没有完整 schema 信息
dataframe.printSchema
```

- 读取 json 文件:

```
val jsonDF= spark.read.json("file:///resources/people.json")
```

接下来就可以使用 DataFrame 的函数操作

```
jsonDF.show
```

注意：直接读取 json 文件有 schema 信息，因为 json 文件本身含有 Schema 信息，SparkSQL 可以自动解析。

- 读取 parquet 文件:

```
val parquetDF=spark.read.parquet("file:///resources/users.parquet")
```

接下来就可以使用 DataFrame 的函数操作

```
parquetDF.show
```

注意：直接读取 parquet 文件有 schema 信息，因为 parquet 文件中保存了列的信息。

2) 两种查询风格：DSL 和 SQL

- 准备工作:

先读取文件并转换为 DataFrame 或 DataSet:

```
val lineRDD= sc.textFile("hdfs://node1:8020/person.txt").map(_.split(" "))
case class Person(id:Int, name:String, age:Int)
val personRDD = lineRDD.map(x => Person(x(0).toInt, x(1), x(2).toInt))
val personDF = personRDD.toDF
personDF.show
//val personDS = personRDD.toDS
//personDS.show
```

- DSL 风格:

SparkSQL 提供了一个领域特定语言 (DSL) 以方便操作结构化数据

1. 查看 name 字段的数据

```
personDF.select(personDF.col("name")).show
personDF.select(personDF("name")).show
personDF.select(col("name")).show
personDF.select("name").show
```

2. 查看 name 和 age 字段数据

```
personDF.select("name", "age").show
```

3. 查询所有的 name 和 age，并将 age+1

```
personDF.select(personDF.col("name"), personDF.col("age") + 1).show
personDF.select(personDF("name"), personDF("age") + 1).show
personDF.select(col("name"), col("age") + 1).show
personDF.select("name", "age").show
//personDF.select("name", "age"+1).show
personDF.select($"name", $"age", $"age"+1).show
```

4. 过滤 age 大于等于 25 的，使用 filter 方法过滤

```
personDF.filter(col("age") >= 25).show
personDF.filter($"age" >= 25).show
```

5. 统计年龄大于 30 的人数

```
personDF.filter(col("age")>30).count()
personDF.filter($"age" >30).count()
```

6. 按年龄进行分组并统计相同年龄的人数

```
personDF.groupBy("age").count().show
```

• SQL 风格：

DataFrame 的一个强大之处就是我们可以将它看作是一个关系型数据表，然后通过可以在程序中使用 `spark.sql()` 来执行 SQL 查询，结果将作为一个 DataFrame 返回。

如果想使用 SQL 风格的语法，需要将 DataFrame 注册成表，采用如下的方式：

```
personDF.createOrReplaceTempView("t_person")
spark.sql("select * from t_person").show
```

1. 显示表的描述信息

```
spark.sql("desc t_person").show
```

2. 查询年龄最大的前两名

```
spark.sql("select * from t_person order by age desc limit 2").show
```

3. 查询年龄大于 30 的人的信息

```
spark.sql("select * from t_person where age > 30 ").show
```

4. 使用 SQL 风格完成 DSL 中的需求

```
spark.sql("select name, age + 1 from t_person").show
spark.sql("select name, age from t_person where age > 25").show
spark.sql("select count(age) from t_person where age > 30").show
spark.sql("select age, count(age) from t_person group by age").show
```

- 总结:

1. DataFrame 和 DataSet 都可以通过 RDD 来进行创建;
2. 也可以通过读取普通文本创建--注意:直接读取没有完整的约束, 需要通过 RDD+Schema;
3. 通过 json/parquet 会有完整的约束;
4. 不管是 DataFrame 还是 DataSet 都可以注册成表, 之后就可以使用 SQL 进行查询了! 也可以使用 DSL!

3) Spark SQL 完成 WordCount

- SQL 风格:

```
import org.apache.spark.SparkContext
import org.apache.spark.sql.{DataFrame, Dataset, SparkSession}

object WordCount {
  def main(args: Array[String]): Unit = {
    //1. 创建SparkSession
    val spark: SparkSession = SparkSession.builder().master("local[*]").appName("SparkSQL").getOrCreate()
    val sc: SparkContext = spark.sparkContext
    sc.setLogLevel("WARN")
    //2. 读取文件
    val fileDF: DataFrame = spark.read.text("D:\\data\\words.txt")
    val fileDS: Dataset[String] = spark.read.textFile("D:\\data\\words.txt")
    //fileDF.show()
    //fileDS.show()
    //3. 对每一行按照空格进行切分并压平
```

```
//fileDF.flatMap(_._split(" ")) //注意: 错误, 因为DF 没有泛型, 不知道_是String
import spark.implicits._
val wordDS: Dataset[String] = fileDS.flatMap(_._split(" "))//注意: 正确, 因为DS 有泛型, 知道_是String
//wordDS.show()
/*
+-----+
|value|
+-----+
|hello|
| me|
|hello|
| you|
...
*/
//4. 对上面的数据进行WordCount
wordDS.createOrReplaceTempView("t_word")
val sql =
  """
  |select value ,count(value) as count
  |from t_word
  |group by value
  |order by count desc
  """.stripMargin
spark.sql(sql).show()

sc.stop()
spark.stop()
}
}
```

- DSL 风格:

```
import org.apache.spark.SparkContext
import org.apache.spark.sql.{DataFrame, Dataset, SparkSession}

object WordCount2 {
  def main(args: Array[String]): Unit = {
    //1. 创建SparkSession
    val spark: SparkSession = SparkSession.builder().master("local[*]").appName("SparkSQL").getOrCreate()
    val sc: SparkContext = spark.sparkContext
    sc.setLogLevel("WARN")
  }
}
```



```
//2. 读取文件
val fileDF: DataFrame = spark.read.text("D:\\data\\words.txt")
val fileDS: Dataset[String] = spark.read.textFile("D:\\data\\words.txt")

//fileDF.show()
//fileDS.show()

//3. 对每一行按照空格进行切分并压平
//fileDF.flatMap(_.split(" ")) //注意: 错误, 因为DF 没有泛型, 不知道_是String
import spark.implicits._
val wordDS: Dataset[String] = fileDS.flatMap(_.split(" "))//注意: 正确, 因为DS 有泛型, 知道_是String
//wordDS.show()

/*
+-----+
|value|
+-----+
|hello|
| me|
|hello|
| you|
...
*/

//4. 对上面的数据进行WordCount
wordDS.groupBy("value").count().orderBy($"count".desc).show()

sc.stop()
spark.stop()
}
}
```

4) Spark SQL 多数据源交互

- 读数据:

读取 json 文件:

```
spark.read.json("D:\\data\\output\\json").show()
```

读取 csv 文件:

```
spark.read.csv("D:\\data\\output\\csv").toDF("id","name","age").show()
```

读取 parquet 文件:

```
spark.read.parquet("D:\\data\\output\\parquet").show()
```

读取 mysql 表:

```
val prop = new Properties()
prop.setProperty("user", "root")
prop.setProperty("password", "root")
spark.read.jdbc(
  "jdbc:mysql://localhost:3306/bigdata?characterEncoding=UTF-8", "person", prop).show()
```

- 写数据:

写入 json 文件:

```
personDF.write.json("D:\\data\\output\\json")
```

写入 csv 文件:

```
personDF.write.csv("D:\\data\\output\\csv")
```

写入 parquet 文件:

```
personDF.write.parquet("D:\\data\\output\\parquet")
```

写入 mysql 表:

```
val prop = new Properties()
prop.setProperty("user", "root")
prop.setProperty("password", "root")
personDF.write.mode(SaveMode.Overwrite).jdbc(
  "jdbc:mysql://localhost:3306/bigdata?characterEncoding=UTF-8", "person", prop)
```

四、Spark Streaming

Spark Streaming 是一个基于 Spark Core 之上的**实时计算框架**，可以从很多数据源消费数据并对数据进行实时的处理，具有高吞吐量和容错能力强等特点。



Spark Streaming 的特点:

1. 易用

可以像编写离线批处理一样去编写流式程序，支持 java/scala/python 语言。

2. 容错

SparkStreaming 在没有额外代码和配置的情况下可以恢复丢失的工作。

3. 易整合到 Spark 体系

流式处理与批处理和交互式查询相结合。

1. 整体流程

Spark Streaming 中，会有一个接收器组件 Receiver，作为一个长期运行的 task 跑在一个 Executor 上。Receiver 接收外部的数据流形成 input DStream。DStream 会被按照时间间隔划分成一批一批的 RDD，当批处理间隔缩短到秒级时，便可以用于处理实时数据流。时间间隔的大小可以由参数指定，一般设在 500 毫秒到几秒之间。

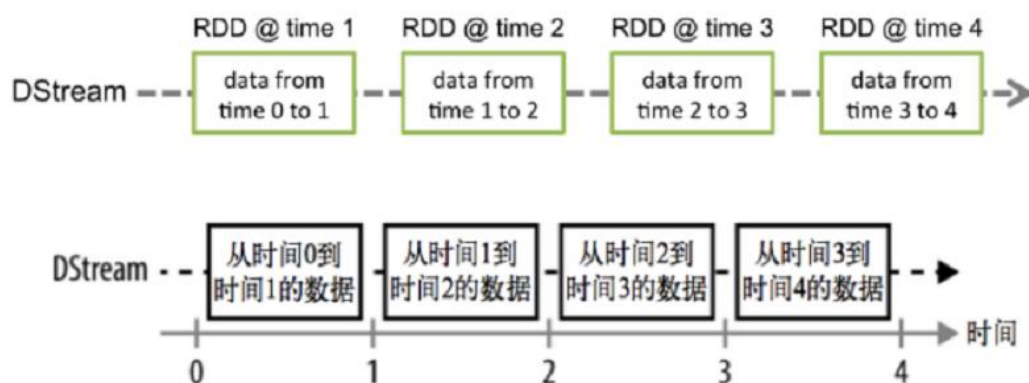
对 DStream 进行操作就是对 RDD 进行操作，计算处理的结果可以传给外部系统。Spark Streaming 的工作流程像下面的图所示一样，接受到实时数据后，给数据分批次，然后传给 Spark Engine 处理最后生成该批次的结果。

2. 数据抽象

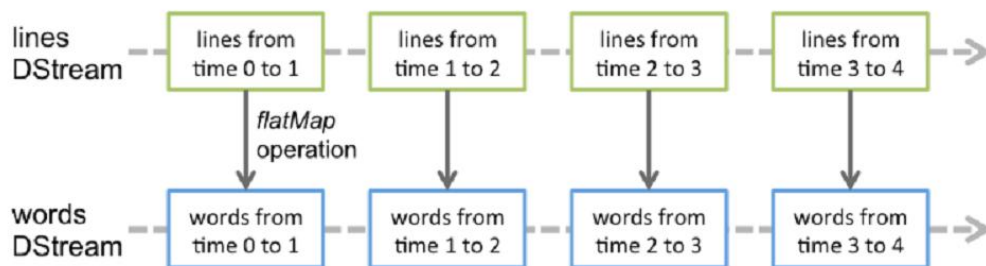
Spark Streaming 的基础抽象是 DStream(Discretized Stream，离散化数据流，连续不断的数据流)，代表持续性的数据流和经过各种 Spark 算子操作后的结果数据流。

可以从以下多个角度深入理解 DStream：

1. DStream 本质上就是一系列时间上连续的 RDD



2. 对 DStream 的数据的操作也是按照 RDD 为单位来进行的



3. 容错性，底层 RDD 之间存在依赖关系，DStream 直接也有依赖关系，RDD 具有容错性，那么 DStream 也具有容错性

4. 准实时性/近实时性

Spark Streaming 将流式计算分解成多个 Spark Job，对于每一时间段数据的处理都会经过 Spark DAG 图分解以及 Spark 的任务集的调度过程。

对于目前版本的 Spark Streaming 而言，其最小的 Batch Size 的选取在 0.5~5 秒钟之间。

所以 Spark Streaming 能够满足流式准实时计算场景，对实时性要求非常高的如高频实时交易场景则不太适合。

• 总结

简单来说 DStream 就是对 RDD 的封装，你对 DStream 进行操作，就是对 RDD 进行操作。

对于 DataFrame/DataSet/DStream 来说本质上都可以理解成 RDD。

3. DStream 相关操作

DStream 上的操作与 RDD 的类似，分为以下两种：

1. Transformations(转换)
2. Output Operations(输出)/Action

1) Transformations

以下是常见 Transformation——都是无状态转换：即每个批次的处理不依赖于之前批次的数据：

Transformation	含义
<code>map(func)</code>	对 DStream 中的各个元素进行 func 函数操作, 然后返回一个新的 DStream
<code>flatMap(func)</code>	与 map 方法类似, 只不过各个输入项可以被输出为零个或多个输出项
<code>filter(func)</code>	过滤出所有函数 func 返回值为 true 的 DStream 元素并返回一个新的 DStream
<code>union(otherStream)</code>	将源 DStream 和输入参数为 otherDStream 的元素合并, 并返回一个新的 DStream
<code>reduceByKey(func, [numTasks])</code>	利用 func 函数对源 DStream 中的 key 进行聚合操作, 然后返回新的 (K, V) 对构成的 DStream
<code>join(otherStream, [numTasks])</code>	输入为 (K, V)、(K, W) 类型的 DStream, 返回一个新的 (K, (V, W)) 类型的 DStream
<code>transform(func)</code>	通过 RDD-to-RDD 函数作用于 DStream 中的各个 RDD, 可以是任意的 RDD 操作, 从而返回一个新的 RDD

除此之外还有一类特殊的 Transformations——有状态转换：当前批次的处理需要使用之前批次的数据或者中间结果。

有状态转换包括基于追踪状态变化的转换 (`updateStateByKey`) 和滑动窗口的转换：

1. **`UpdateStateByKey(func)`**
2. **Window Operations 窗口操作**

2) Output/Action

Output Operations 可以将 DStream 的数据输出到外部的数据库或文件系统。当某个 Output Operations 被调用时, spark streaming 程序才会开始真正的计算过程(与 RDD 的 Action 类似)。

Output Operation	含义
<code>print()</code>	打印到控制台
<code>saveAsTextFiles(prefix, [suffix])</code>	保存流的内容为文本文件, 文件名为 "prefix-TIME_IN_MS[. suffix]"
<code>saveAsObjectFiles(prefix, [suffix])</code>	保存流的内容为 SequenceFile, 文件名为

Output Operation	含义
	"prefix-TIME_IN_MS[. suffix]"
saveAsHadoopFiles(prefix, [suffix])	保存流的内容为 hadoop 文件，文件名为 "prefix-TIME_IN_MS[. suffix]"
foreachRDD(func)	对 Dstream 里面的每个 RDD 执行 func

4. Spark Streaming 完成实时需求

1) WordCount

- 首先在 linux 服务器上安装 nc 工具
nc 是 netcat 的简称，原本是用来设置路由器，我们可以利用它向某个端口发送数据 `yum install -y nc`
- 启动一个服务端并开放 9999 端口，等一下往这个端口发数据
`nc -lk 9999`
- 发送数据
- 接收数据，代码示例：

```
import org.apache.spark.streaming.dstream.{DStream, ReceiverInputDStream}
import org.apache.spark.{SparkConf, SparkContext}
import org.apache.spark.streaming.{Seconds, StreamingContext}

object WordCount {
  def main(args: Array[String]): Unit = {
    //1. 创建StreamingContext
    //spark.master should be set as local[n], n > 1
    val conf = new SparkConf().setAppName("wc").setMaster("local[*]")
    val sc = new SparkContext(conf)
    sc.setLogLevel("WARN")
    val ssc = new StreamingContext(sc, Seconds(5)) //5 表示 5 秒中对数据进行切分形成一个 RDD
    //2. 监听Socket 接收数据
    //ReceiverInputDStream 就是接收到的所有的数据组成的RDD, 封装成了DStream, 接下来对 DStream 进行操作就是对 RDD 进行操作
    val dataDStream: ReceiverInputDStream[String] = ssc.socketTextStream("node01", 9999)
    //3. 操作数据
    val wordDStream: DStream[String] = dataDStream.flatMap(_.split(" "))
```

```
val wordAndOneDStream: DStream[(String, Int)] = wordDStream.map(_._1)
val wordAndCount: DStream[(String, Int)] = wordAndOneDStream.reduceByKey(_+_ )
wordAndCount.print()
ssc.start()//开启
ssc.awaitTermination()//等待停止
}
```

2) updateStateByKey

- 问题:

在上面的那个案例中存在这样一个问题:

每个批次的单词次数都被正确的统计出来,但是结果不能累加!

如果需要累加需要使用 updateStateByKey(func)来更新状态。

代码示例:

```
import org.apache.spark.streaming.dstream.{DStream, ReceiverInputDStream}
import org.apache.spark.streaming.{Seconds, StreamingContext}
import org.apache.spark.{SparkConf, SparkContext}

object WordCount2 {
  def main(args: Array[String]): Unit = {
    //1. 创建StreamingContext
    //spark.master should be set as local[n], n > 1
    val conf = new SparkConf().setAppName("wc").setMaster("local[*]")
    val sc = new SparkContext(conf)
    sc.setLogLevel("WARN")
    val ssc = new StreamingContext(sc, Seconds(5))//5 表示5 秒中对数据进行切分形成一个
    RDD
    //requirement failed: ....Please set it by StreamingContext.checkpoint().
    //注意:我们在下面使用到了updateStateByKey 对当前数据和历史数据进行累加
    //那么历史数据存在哪?我们需要给他设置一个checkpoint 目录
    ssc.checkpoint("./wc")//开发中HDFS
    //2. 监听Socket 接收数据
    //ReceiverInputDStream 就是接收到的所有的数据组成的RDD,封装成了DStream,接下来对
    DStream 进行操作就是对RDD 进行操作
    val dataDStream: ReceiverInputDStream[String] = ssc.socketTextStream("node01", 9
    999)
    //3. 操作数据
    val wordDStream: DStream[String] = dataDStream.flatMap(_.split(" "))
```

```

val wordAndOneDStream: DStream[(String, Int)] = wordDStream.map((_,1))
//val wordAndCount: DStream[(String, Int)] = wordAndOneDStream.reduceByKey(_+_ )
//=====使用updateStateByKey 对当前数据和历史数据进行累加
=====
val wordAndCount: DStream[(String, Int)] =wordAndOneDStream.updateStateByKey(updateFunc)
wordAndCount.print()
ssc.start()//开启
ssc.awaitTermination()//等待优雅停止
}

//currentValues: 当前批次的 value 值, 如:1,1,1 (以测试数据中的hadoop 为例)
//historyValue: 之前累计的历史值, 第一次没有值是0, 第二次是3
//目标是把当前数据+历史数据返回作为新的结果(下次的历史数据)
def updateFunc(currentValues:Seq[Int], historyValue:Option[Int] ):Option[Int] ={
    val result: Int = currentValues.sum + historyValue.getOrElse(0)
    Some(result)
}
}

```

3) reduceByKeyAndWindow

使用上面的代码已经能够完成对所有历史数据的聚合,但是实际中可能会有一些需求,需要对指定时间范围的数据进行统计。

比如:

百度/微博的热搜排行榜 统计最近 24 小时的热搜词,每隔 5 分钟更新一次,所以面对这样的需求我们需要使用窗口操作 Window Operations。

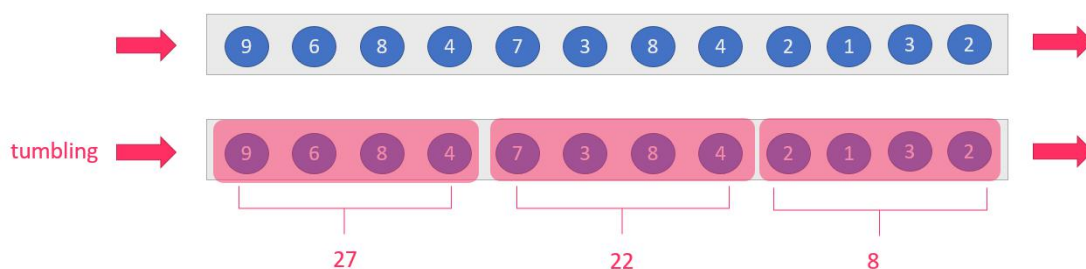
图解:

我们先提出一个问题: 统计经过某红绿灯的汽车数量之和?

假设在一个红绿灯处,我们每隔 15 秒统计一次通过此红绿灯的汽车数量,如下图:

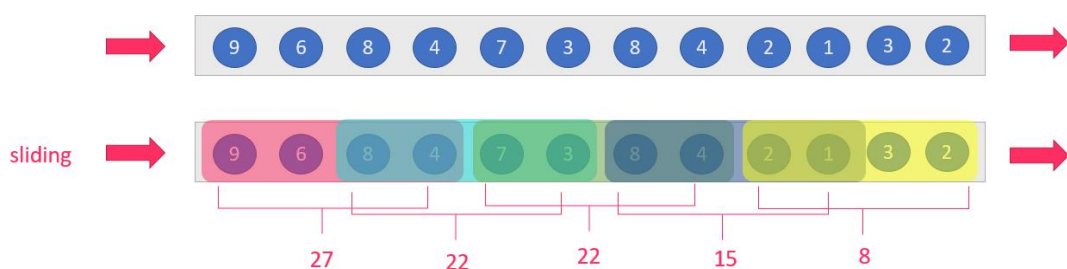


可以把汽车的经过看成一个流,无穷的流,不断有汽车经过此红绿灯,因此无法统计总共的汽车数量。但是,我们可以换一种思路,每隔 15 秒,我们都将与上一次的结果进行 sum 操作(滑动聚合,但是这个结果似乎还是无法回答我们的问题,根本原因在于流是无界的,我们不能限制流,但可以在有一个有界的范围内处理无界的流数据。



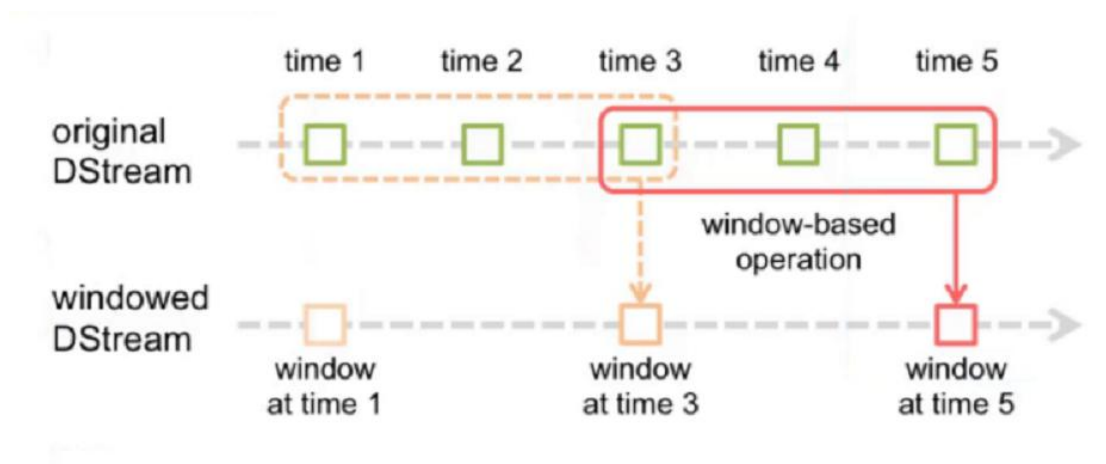
因此，我们需要换一个问题的提法：每分钟经过某红绿灯的汽车数量之和？这个问题，就相当于一个定义了一个 Window（窗口），window 的界限是 1 分钟，且每分钟内的数据互不干扰，因此也可以称为翻滚（不重合）窗口，如下图：第一分钟的数量为 8，第二分钟是 22，第三分钟是 27。。。这样，1 个小时内会有 60 个 window。

再考虑一种情况，每 30 秒统计一次过去 1 分钟的汽车数量之和：



此时，window 出现了重合。这样，1 个小时内会有 120 个 window。

滑动窗口转换操作的计算过程如下图所示：



我们可以事先设定一个滑动窗口的长度（也就是窗口的持续时间），并且设定滑动窗口的时间间隔（每隔多长时间执行一次计算），

比如设置滑动窗口的长度（也就是窗口的持续时间）为 24h，设置滑动窗口的时间间隔（每隔多长时间执行一次计算）为 1h

那么意思就是:每隔 1H 计算最近 24H 的数据



蓝色的是DStream
黄色的是RDD

切分批次:5s
窗口长度:10s
滑动间隔:5s

滑动间隔5s<窗口长度10s--最后结果:数据会被重复(重新)计算
对应到业务中就是每隔5秒计算最近10s的数据

窗口长度/大小 10s

窗口长度/大小 10s



切分批次:5s
窗口长度:10s
滑动间隔:10s

滑动间隔10s=窗口长度10s--最后结果:数据不会丢失,且只计算一次
对应到业务中就是每隔10秒计算最近10s的数据

窗口长度/大小 10s

窗口长度/大小 10s



切分批次:5s
窗口长度:10s
滑动间隔:15s

滑动间隔15s>窗口长度10s--最后结果:数据会丢失
对应到业务中是不被允许的

窗口长度/大小 10s

窗口长度/大小 10s

代码示例:

```
import org.apache.spark.streaming.dstream.{DStream, ReceiverInputDStream}
import org.apache.spark.streaming.{Seconds, StreamingContext}
import org.apache.spark.{SparkConf, SparkContext}

object WordCount3 {
  def main(args: Array[String]): Unit = {
    //1. 创建StreamingContext
    //spark.master should be set as local[n], n > 1
    val conf = new SparkConf().setAppName("wc").setMaster("local[*]")
    val sc = new SparkContext(conf)
    sc.setLogLevel("WARN")
    val ssc = new StreamingContext(sc, Seconds(5)) //5 表示5秒中对数据进行切分形成一个RDD
    //2. 监听Socket 接收数据
    //ReceiverInputDStream 就是接收到的所有的数据组成的RDD, 封装成了DStream, 接下来对DStream 进行操作就是对RDD 进行操作
    val dataDStream: ReceiverInputDStream[String] = ssc.socketTextStream("node01", 9999)
    //3. 操作数据
```

```

val wordDStream: DStream[String] = dataDStream.flatMap(_.split(" "))
val wordAndOneDStream: DStream[(String, Int)] = wordDStream.map((_,1))

//4. 使用窗口函数进行WordCount 计数
//reduceFunc: (V, V) => V, 集合函数
//windowDuration: Duration, 窗口长度/宽度
//slideDuration: Duration, 窗口滑动间隔
//注意:windowDuration 和slideDuration 必须是batchDuration 的倍数
//windowDuration=slideDuration: 数据不会丢失也不会重复计算== 开发中会使用
//windowDuration>slideDuration: 数据会重复计算== 开发中会使用
//windowDuration<slideDuration: 数据会丢失
//下面的代码表示:
//windowDuration=10
//slideDuration=5
//那么执行结果就是每隔5s 计算最近10s 的数据
//比如开发中让你统计最近1 小时的数据, 每隔1 分钟计算一次, 那么参数该如何设置?
//windowDuration=Minutes(60)
//slideDuration=Minutes(1)

val wordAndCount: DStream[(String, Int)] = wordAndOneDStream.reduceByKeyAndWindow(
  (a:Int,b:Int)=>a+b, Seconds(10), Seconds(5))

wordAndCount.print()

ssc.start()//开启
ssc.awaitTermination()//等待优雅停止
}
}

```

五、Structured Streaming

在 2.0 之前, Spark Streaming 作为核心 API 的扩展, 针对实时数据流, 提供了一套可扩展、高吞吐、可容错的流式计算模型。Spark Streaming 会接收实时数据源的数据, 并切分成很多小的 batches, 然后被 Spark Engine 执行, 产出同样由很多小的 batchs 组成的结果流。本质上, 这是一种 micro-batch (微批处理) 的方式处理, 用批的思想去处理流数据. 这种设计让 **Spark Streaming 面对复杂的流式处理场景时捉襟见肘**。

spark streaming 这种构建在微批处理上的流计算引擎, 比较突出的问题就是处理延时较高 (无法优化到秒以下的数量级), 以及无法支持基于 event_time 的时间窗口做聚合逻辑。

spark 在 2.0 版本中发布了新的流计算的 API, Structured Streaming/结构化流。

Structured Streaming 是一个基于 Spark SQL 引擎的可扩展、容错的流处理引擎。统一了流、批的编程模型，你可以使用静态数据批处理一样的方式来编写流式计算操作。并且支持基于 `event_time` 的时间窗口的处理逻辑。

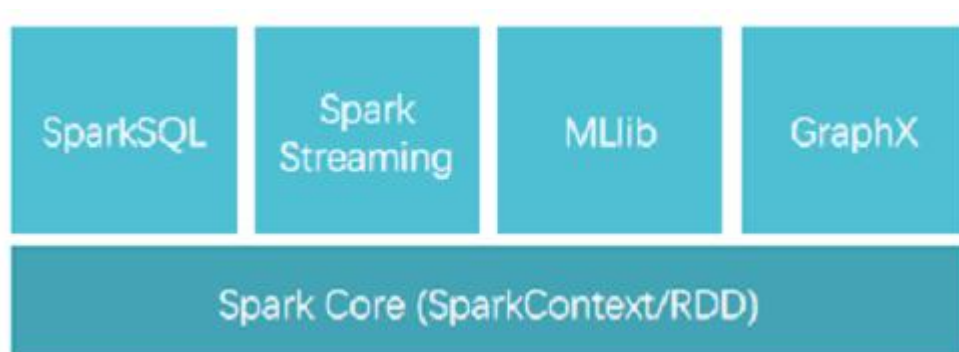
随着数据不断地到达，Spark 引擎会以一种增量的方式来执行这些操作，并且持续更新结算结果。可以使用 Scala、Java、Python 或 R 中的 `DataSet` / `DataFrame` API 来表示流聚合、事件时间窗口、流到批连接等。此外，Structured Streaming 会通过 checkpoint 和预写日志等机制来实现 Exactly-Once 语义。简单来说，对于开发人员来说，根本不用去考虑是流式计算，还是批处理，只要使用同样的方式来编写计算操作即可，Structured Streaming 提供了快速、可扩展、容错、端到端的一次性流处理，而用户无需考虑更多细节。

默认情况下，结构化流式查询使用微批处理引擎进行处理，该引擎将数据流作为一系列小批处理作业进行处理，从而实现端到端的延迟，最短可达 100 毫秒，并且完全可以保证一次容错。**自 Spark 2.3 以来，引入了一种新的低延迟处理模式，称为连续处理，它可以在至少一次保证的情况下实现低至 1 毫秒的端到端延迟。也就是类似于 Flink 那样的实时流，而不是小批量处理。**实际开发可以根据应用程序要求选择处理模式，但是连续处理在使用的时候仍然有很多限制，目前大部分情况还是应该采用小批量模式。

1. API

- **Spark Streaming 时代 - DStream-RDD**

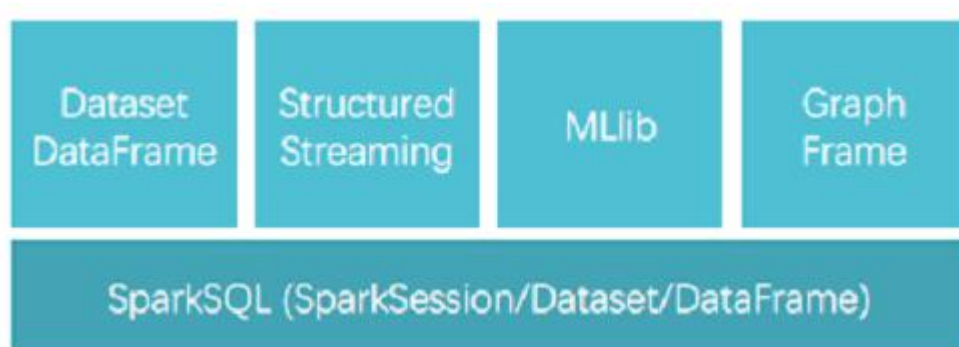
Spark Streaming 采用的数据抽象是 DStream，而本质上就是时间上连续的 RDD，对数据流的操作就是针对 RDD 的操作。



- **Structured Streaming 时代 - DataSet/DataFrame -RDD**

Structured Streaming 是 Spark2.0 新增的可扩展和高容错性的实时计算框架，它构建于 Spark SQL 引擎，把流式计算也统一到 DataFrame/Dataset 里去了。

Structured Streaming 相比于 Spark Streaming 的进步就类似于 Dataset 相比于 RDD 的进步。



2. 核心思想

Structured Streaming 最核心的思想就是将实时到达的数据看作是一个不断追加的 unbound table 无界表，到达流的每个数据项(RDD)就像是表中的一个新行被附加到无边界的表中. 这样用户就可以用静态结构化数据的批处理查询方式进行流计算，如可以使用 SQL 对到来的每一行数据进行实时查询处理。

3. 应用场景

Structured Streaming 将数据源映射为类似于关系数据库中的表，然后将经过计算得到的结果映射为另一张表，完全以结构化的方式去操作流式数据，**这种编程模型非常有利于处理分析结构化的实时数据**；

4. Structured Streaming 实战

1) 读取 Socket 数据

```
import org.apache.spark.SparkContext
import org.apache.spark.sql.streaming.Trigger
import org.apache.spark.sql.{DataFrame, Dataset, Row, SparkSession}

object WordCount {
```

```
def main(args: Array[String]): Unit = {
    //1. 创建SparkSession, 因为StructuredStreaming 的数据模型也是DataFrame/DataSet
    val spark: SparkSession = SparkSession.builder().master("local[*]").appName("SparkSQL").getOrCreate()
    val sc: SparkContext = spark.sparkContext
    sc.setLogLevel("WARN")
    //2. 接收数据
    val dataDF: DataFrame = spark.readStream
        .option("host", "node01")
        .option("port", 9999)
        .format("socket")
        .load()
    //3. 处理数据
    import spark.implicits._
    val dataDS: Dataset[String] = dataDF.as[String]
    val wordDS: Dataset[String] = dataDS.flatMap(_.split(" "))
    val result: Dataset[Row] = wordDS.groupBy("value").count().sort($"count".desc)
    //result.show()
    //Queries with streaming sources must be executed with writeStream.start();
    result.writeStream
        .format("console")//往控制台写
        .outputMode("complete")//每次将所有的数据写出
        .trigger(Trigger.ProcessingTime(0))//触发时间间隔,0 表示尽可能的快
        //option("checkpointLocation", ".ckp")//设置checkpoint 目录,socket 不支持数据恢复,所以第二次启动会报错,需要注释
        .start()//开启
        .awaitTermination()//等待停止
}
}
```

2) 读取目录下文本数据

```
import org.apache.spark.SparkContext
import org.apache.spark.sql.streaming.Trigger
import org.apache.spark.sql.types.StructType
import org.apache.spark.sql.{DataFrame, Dataset, Row, SparkSession}
/**
 * {"name": "json", "age": 23, "hobby": "running"}
 * {"name": "charles", "age": 32, "hobby": "basketball"}
 * {"name": "tom", "age": 28, "hobby": "football"}
 * {"name": "lili", "age": 24, "hobby": "running"}
 * {"name": "bob", "age": 20, "hobby": "swimming"}
 * 统计年龄小于 25 岁的人群的爱好排行榜
 */
```

```

*/
object WordCount2 {
  def main(args: Array[String]): Unit = {
    //1. 创建SparkSession, 因为StructuredStreaming 的数据模型也是DataFrame/DataSet
    val spark: SparkSession = SparkSession.builder().master("local[*]").appName("SparkSQL").getOrCreate()
    val sc: SparkContext = spark.sparkContext
    sc.setLogLevel("WARN")
    val Schema: StructType = new StructType()
      .add("name", "string")
      .add("age", "integer")
      .add("hobby", "string")
    //2. 接收数据
    import spark.implicits._
    // Schema must be specified when creating a streaming source DataFrame.
    val dataDF: DataFrame = spark.readStream.schema(Schema).json("D:\\data\\spark\\data")
    //3. 处理数据
    val result: Dataset[Row] = dataDF.filter($"age" < 25).groupBy("hobby").count().sort($"count".desc)
    //4. 输出结果
    result.writeStream
      .format("console")
      .outputMode("complete")
      .trigger(Trigger.ProcessingTime(0))
      .start()
      .awaitTermination()
  }
}

```

3) 计算操作

获得到 Source 之后的基本数据处理方式和之前学习的 DataFrame、DataSet 一致，不再赘述。

官网示例代码：

```

case class DeviceData(device: String, deviceType: String, signal: Double, time: DateTime)
val df: DataFrame = ... // streaming DataFrame with IOT device data with schema { device: string, deviceType: string, signal: double, time: string }
val ds: Dataset[DeviceData] = df.as[DeviceData] // streaming Dataset with IOT device data
// Select the devices which have signal more than 10

```

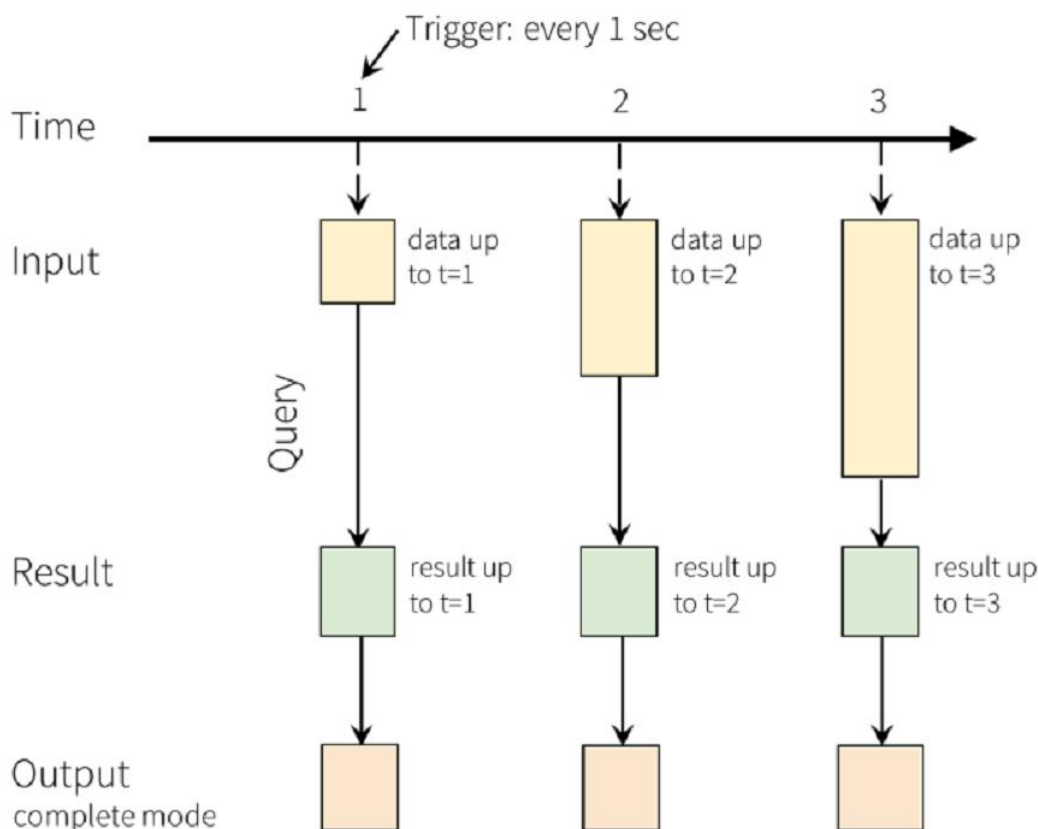
```
df.select("device").where("signal > 10")      // using untyped APIs
ds.filter(_.signal > 10).map(_.device)        // using typed APIs
// Running count of the number of updates for each device type
df.groupBy("deviceType").count()             // using untyped API
// Running average signal for each device type
import org.apache.spark.sql.expressions.scalalang.typed
ds.groupByKey(_.deviceType).agg(typed.avg(_.signal)) // using typed API
```

4) 输出

计算结果可以选择输出到多种设备并进行如下设定：

1. output mode: 以哪种方式将 result table 的数据写入 sink, 即是全部输出 complete 还是只输出新增数据;
2. format/output sink 的一些细节: 数据格式、位置等。如 console;
3. query name: 指定查询的标识。类似 tempview 的名字;
4. trigger interval: 触发间隔, 如果不指定, 默认会尽可能快速地处理数据;
5. checkpointLocation: 一般是 hdfs 上的目录。注意: Socket 不支持数据恢复, 如果设置了, 第二次启动会报错, Kafka 支持。

output mode:



Programming Model for Structured Streaming

每当结果表更新时，我们都希望将更改后的结果行写入外部接收器。

这里有三种输出模型：

1. **Append mode**: 默认模式，新增的行才输出，每次更新结果集时，只将新添加到结果集的结果行输出到接收器。仅支持那些添加到结果表中的行永远不会更改的查询。因此，此模式保证每行仅输出一次。例如，仅查询 `select`, `where`, `map`, `flatMap`, `filter`, `join` 等会支持追加模式。不支持聚合
2. **Complete mode**: 所有内容都输出，每次触发后，整个结果表将输出到接收器。聚合查询支持此功能。仅适用于包含聚合操作的查询。
3. **Update mode**: 更新的行才输出，每次更新结果集时，仅将被更新的结果行输出到接收器(自 Spark 2.1.1 起可用)，不支持排序

output sink:

Sink	Supported Output Modes	Options	Fault-tolerant	Notes
File Sink	Append	path: path to the output directory, must be specified. For file-format-specific options, see the related methods in <code>DataFrameWriter</code> (Scala/Java/Python/R). E.g. for "parquet" format options see <code>DataFrameWriter.parquet()</code>	Yes (exactly-once)	Supports writes to partitioned tables. Partitioning by time may be useful.
Kafka Sink	Append, Update, Complete	See the Kafka Integration Guide	Yes (at-least-once)	More details in the Kafka Integration Guide
Foreach Sink	Append, Update, Complete	None	Depends on <code>ForeachWriter</code> implementation	More details in the next section
ForeachBatch Sink	Append, Update, Complete	None	Depends on the implementation	More details in the next section
Console Sink	Append, Update, Complete	<code>numRows</code> : Number of rows to print every trigger (default: 20) <code>truncate</code> : Whether to truncate the output if too long (default: true)	No	
Memory Sink	Append, Complete	None	No. But in Complete Mode, restarted query will recreate the full table.	Table name is the query name.

- 说明:

File sink: 输出存储到一个目录中。支持 `parquet` 文件, 以及 `append` 模式。

```
writeStream
  .format("parquet")           // can be "orc", "json", "csv", etc.
  .option("path", "path/to/destination/dir")
  .start()
```

Kafka sink: 将输出存储到 Kafka 中的一个或多个 topics 中。

```
writeStream
  .format("kafka")
  .option("kafka.bootstrap.servers", "host1:port1,host2:port2")
  .option("topic", "updates")
  .start()
```

Foreach sink: 对输出中的记录运行任意计算

```
writeStream
  .foreach(...)
  .start()
```

Console sink: 将输出打印到控制台

```
writeStream
  .format("console")
  .start()
```

六、Spark 的两种核心 Shuffle

在 MapReduce 框架中，Shuffle 阶段是连接 Map 与 Reduce 之间的桥梁，Map 阶段通过 Shuffle 过程将数据输出到 Reduce 阶段中。由于 Shuffle 涉及磁盘的读写和网络 I/O，因此 Shuffle 性能的高低直接影响整个程序的性能。Spark 也有 Map 阶段和 Reduce 阶段，因此也会出现 Shuffle。

Spark Shuffle

Spark Shuffle 分为两种：一种是基于 Hash 的 Shuffle；另一种是基于 Sort 的 Shuffle。先介绍下它们的发展历程，有助于我们更好的理解 Shuffle：

在 Spark 1.1 之前，Spark 中只实现了一种 Shuffle 方式，即基于 Hash 的 Shuffle。在 Spark 1.1 版本中引入了基于 Sort 的 Shuffle 实现方式，并且 Spark 1.2 版本之后，默认的实现方式从基于 Hash 的 Shuffle 修改为基于 Sort 的 Shuffle 实现方式，即使用的 ShuffleManager 从默认的 hash 修改为 sort。在 Spark 2.0 版本中，Hash Shuffle 方式已经不再使用。

Spark 之所以一开始就提供基于 Hash 的 Shuffle 实现机制，其主要目的之一就是为了避免不需要的排序，大家想下 Hadoop 中的 MapReduce，是将 sort 作为固定步骤，有许多并不需要排序的任务，MapReduce 也会对其进行排序，造成了许多不必要的开销。

在基于 Hash 的 Shuffle 实现方式中，每个 Mapper 阶段的 Task 会为每个 Reduce 阶段的 Task 生成一个文件，通常会产生大量的文件（即对应为 $M \times R$ 个中间文件，其中，M 表示 Mapper 阶段的 Task 个数，R 表示 Reduce 阶段的 Task 个数）伴随大量的随机磁盘 I/O 操作与大量的内存开销。

为了缓解上述问题，在 Spark 0.8.1 版本中为基于 Hash 的 Shuffle 实现引入了 Shuffle Consolidate 机制（即文件合并机制），将 Mapper 端生成的中间文件进行合并的处理机制。通过配置属性 `spark.shuffle.compressFiles=true`，减少中间生成的文件数量。通过文件合并，可以将中间文件的生成方式修改为每个执行单位为每个 Reduce 阶段的 Task 生成一个文件。

执行单位对应为：每个 Mapper 端的 Cores 数 / 每个 Task 分配的 Cores 数（默认为 1）。最终可以将文件个数从 $M \times R$ 修改为 $E \times C / T \times R$ ，其中，E 表示 Executors 个数，C 表示可用 Cores 个数，T 表示 Task 分配的 Cores 数。

Spark 1.1 版本引入了 Sort Shuffle：

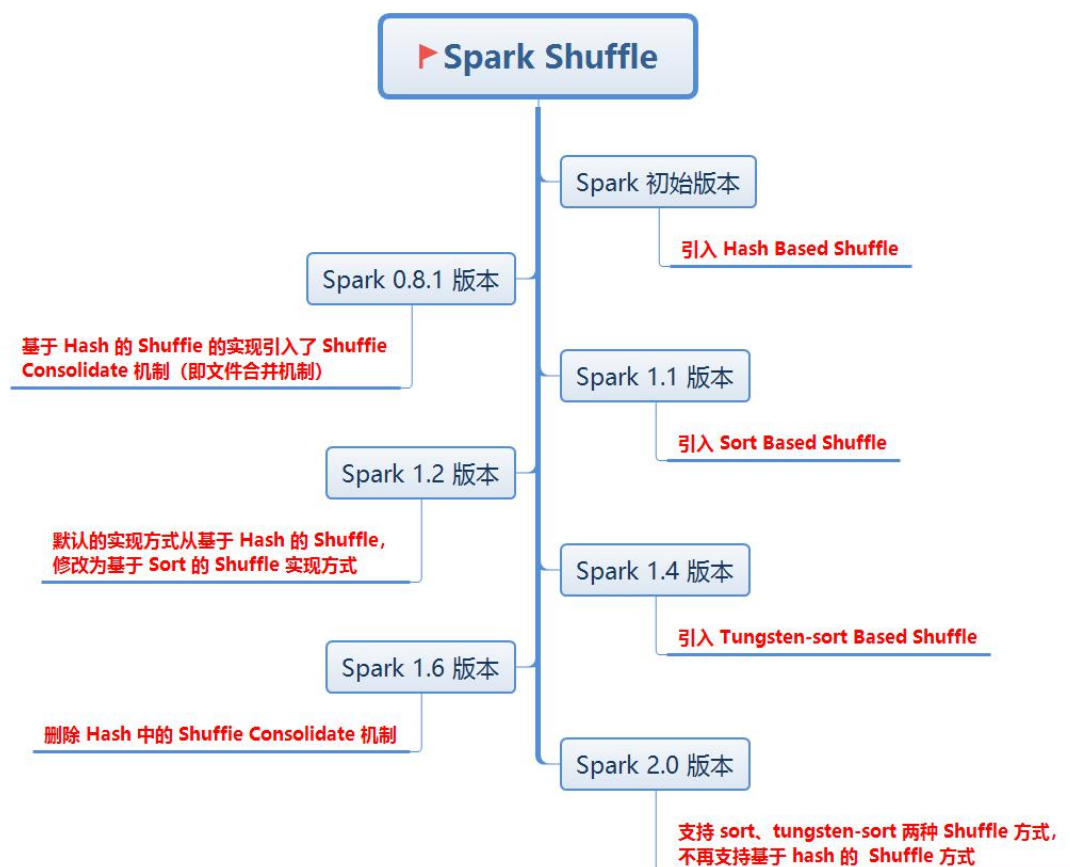
基于 Hash 的 Shuffle 的实现方式中，生成的中间结果文件的个数都会依赖于 Reduce 阶段的 Task 个数，即 Reduce 端的并行度，因此文件数仍然不可控，无法真正解决问题。为了更好地解决问题，在 Spark1.1 版本引入了基于 Sort 的 Shuffle 实现方式，并且在 Spark 1.2 版本之后，默认的实现方式也从基于 Hash 的 Shuffle，修改为基于 Sort 的 Shuffle 实现方式，即使用的 ShuffleManager 从默认的 hash 修改为 sort。

在基于 Sort 的 Shuffle 中，每个 Mapper 阶段的 Task 不会为每 Reduce 阶段的 Task 生成一个单独的文件，而是全部写到一个数据（Data）文件中，同时生成一个索引（Index）文件，Reduce 阶段的各个 Task 可以通过该索引文件获取相关的数据。避免产生大量文件的直接收益就是降低随机磁盘 I/O 与内存的开销。最终生成的文件个数减少到 $2 * M$ ，其中 M 表示 Mapper 阶段的 Task 个数，每个 Mapper 阶段的 Task 分别生成两个文件（1 个数据文件、1 个索引文件），最终的文件个数为 M 个数据文件与 M 个索引文件。因此，最终文件个数是 $2 * M$ 个。

从 Spark 1.4 版本开始，在 Shuffle 过程中也引入了基于 Tungsten-Sort 的 Shuffle 实现方式，通过 Tungsten 项目所做的优化，可以极大提高 Spark 在数据处理上的性能。（Tungsten 翻译为中文是钨丝）

注：在一些特定的应用场景下，采用基于 Hash 实现 Shuffle 机制的性能会超过基于 Sort 的 Shuffle 实现机制。

一张图了解下 Spark Shuffle 的迭代历史：



Spark Shuffle 迭代历史

为什么 Spark 最终还是放弃了 HashShuffle，使用了 Sorted-Based Shuffle？

我们可以从 Spark 最根本要优化和迫切要解决的问题中找到答案，使用 HashShuffle 的 Spark 在 Shuffle 时产生大量的文件。当数据量越来越多时，产生的文件量是不可控的，这严重制约了 Spark 的性能及扩展能力，所以 Spark 必须要解决这个问题，减少 Mapper 端 ShuffleWriter 产生的文件数量，这样便可以让 Spark 从几百台集群的规模瞬间变成可以支持几千台，甚至几万台集群的规模。

但使用 Sorted-Based Shuffle 就完美了吗，答案是否定的，Sorted-Based Shuffle 也有缺点，其缺点反而是它排序的特性，它强制要求数据在 Mapper 端必须先进行排序，所以导致它排序的速度有点慢。好在出现了 Tungsten-Sort Shuffle，它对排序算法进行了改进，优化了排序的速度。Tungsten-Sort Shuffle 已经并入了 Sorted-Based Shuffle，Spark 的引擎会自动识别程序需要的是 Sorted-Based Shuffle，还是 Tungsten-Sort Shuffle。

下面详细剖析每个 Shuffle 的底层执行原理：

一、Hash Shuffle 解析

以下的讨论都假设每个 Executor 有 1 个 cpu core。

1. HashShuffleManager

shuffle write 阶段，主要就是在一个 stage 结束计算之后，为了下一个 stage 可以执行 shuffle 类的算子（比如 reduceByKey），而将每个 task 处理的数据按 key 进行“划分”。所谓“划分”，就是对相同的 key 执行 hash 算法，从而将相同 key 都写入同一个磁盘文件中，而每一个磁盘文件都只属于下游 stage 的一个 task。在将数据写入磁盘之前，会先将数据写入内存缓冲中，当内存缓冲填满之后，才会溢写到磁盘文件中去。

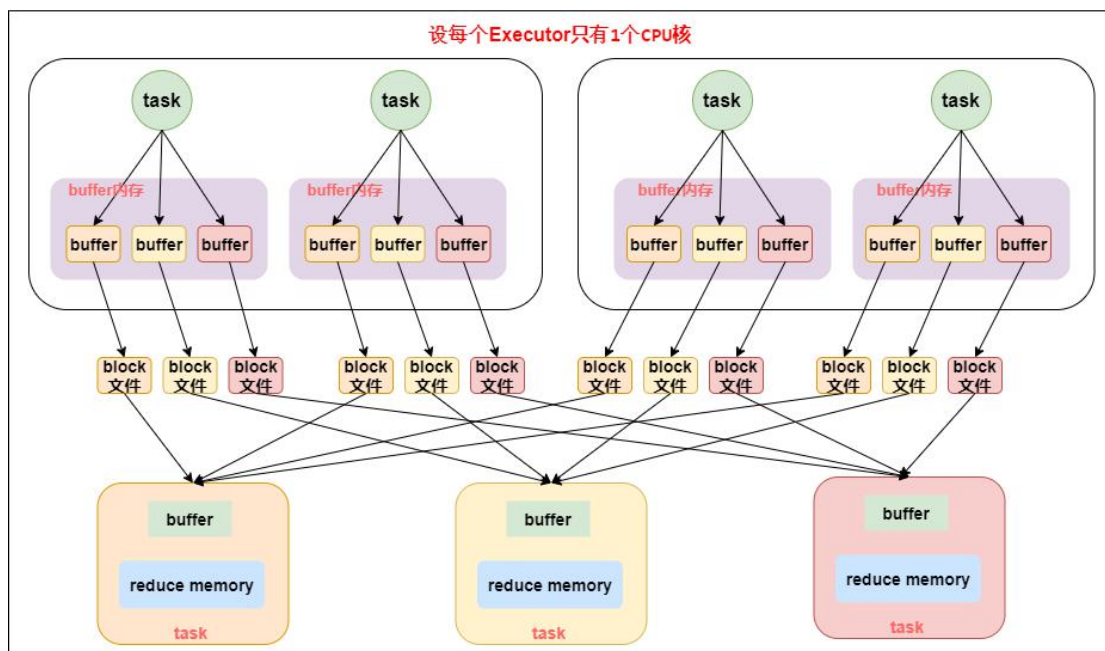
下一个 stage 的 task 有多少个，当前 stage 的每个 task 就要创建多少份磁盘文件。比如下一个 stage 总共有 100 个 task，那么当前 stage 的每个 task 都要创建 100 份磁盘文件。如果当前 stage 有 50 个 task，总共有 10 个 Executor，每个 Executor 执行 5 个 task，那么每个 Executor 上总共就要创建 500 个磁盘文件，所有 Executor 上会创建 5000 个磁盘文件。由此可见，未经优化的 shuffle write 操作所产生的磁盘文件的数量是极其惊人的。

shuffle read 阶段，通常就是一个 stage 刚开始时要做的事情。此时该 stage 的每一个 task 就需要将上一个 stage 的计算结果中的所有相同 key，从各个节点上通过网络都拉取到自己所在的节点上，然后进行 key 的聚合或连接等操作。

由于 shuffle write 的过程中，map task 给下游 stage 的每个 reduce task 都创建了一个磁盘文件，因此 shuffle read 的过程中，每个 reduce task 只要从上游 stage 的所有 map task 所在节点上，拉取属于自己的那一个磁盘文件即可。

shuffle read 的拉取过程是一边拉取一边进行聚合的。每个 shuffle read task 都会有一个自己的 buffer 缓冲，每次都只能拉取与 buffer 缓冲相同大小的数据，然后通过内存中的一个 Map 进行聚合等操作。聚合完一批数据后，再拉取下一批数据，并放到 buffer 缓冲中进行聚合操作。以此类推，直到最后将所有数据拉取完，并得到最终的结果。

HashShuffleManager 工作原理如下图所示：



未优化的 HashShuffleManager 工作原理

2. 优化的 HashShuffleManager

为了优化 HashShuffleManager 我们可以设置一个参数：

`spark.shuffle consolidateFiles`，该参数默认值为 `false`，将其设置为 `true` 即可开启优化机制，通常来说，如果我们使用 HashShuffleManager，那么都建议开启这个选项。

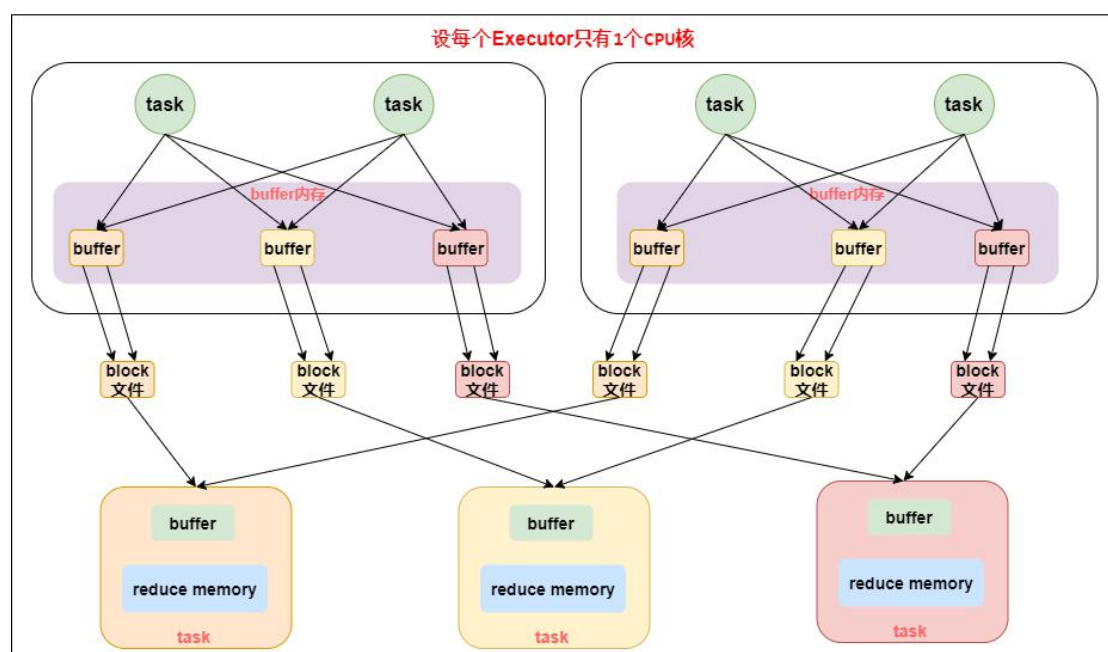
开启 `consolidate` 机制之后，在 shuffle write 过程中，task 就不是为下游 stage 的每个 task 创建一个磁盘文件了，此时会出现 `shuffleFileGroup` 的概念，每个 `shuffleFileGroup` 会对应一批磁盘文件，磁盘文件的数量与下游 stage 的 task 数量是相同的。一个 Executor 上有多少个 cpu core，就可以并行执行多少个 task。而第一批并行执行的每个 task 都会创建一个 `shuffleFileGroup`，并将数据写入对应的磁盘文件内。

当 Executor 的 cpu core 执行完一批 task，接着执行下一批 task 时，下一批 task 就会复用之前已有的 `shuffleFileGroup`，包括其中的磁盘文件，也就是说，此时 task 会将数据写入已有的磁盘文件中，而不会写入新的磁盘文件中。因此，`consolidate` 机制允许不同的 task 复用同一批磁盘文件，这样就可以有效将多个 task 的磁盘文件进行一定程度上的合并，从而大幅度减少磁盘文件的数量，进而提升 shuffle write 的性能。

假设第二个 stage 有 100 个 task，第一个 stage 有 50 个 task，总共还是有 10 个 Executor (Executor CPU 个数为 1)，每个 Executor 执行 5 个 task。那么原本使用未经优化的 HashShuffleManager 时，每个 Executor 会产生 500 个磁盘文件，所有 Executor 会产生 5000 个磁盘文件的。但是此时经过优化之后，每个 Executor 创建的磁盘文件的数量的计算公式为： $\text{cpu core 的数量} * \text{下一个 stage 的 task 数量}$ ，也就是说，每个 Executor 此时只会创建 100 个磁盘文件，所有 Executor 只会创建 1000 个磁盘文件。

这个功能优点明显，但为什么 Spark 一直没有在基于 Hash Shuffle 的实现中将功能设置为默认选项呢，官方给出的说法是这个功能还欠稳定。

优化后的 HashShuffleManager 工作原理如下图所示：



优化后的 HashShuffleManager 工作原理

基于 Hash 的 Shuffle 机制的优缺点

优点：

- 可以省略不必要的排序开销。
- 避免了排序所需的内存开销。

缺点：

- 生产的文件过多，会对文件系统造成压力。
- 大量小文件的随机读写带来一定的磁盘开销。
- 数据块写入时所需的缓存空间也会随之增加，对内存造成压力。

二、SortShuffle 解析

SortShuffleManager 的运行机制主要分成三种：

1. **普通运行机制**；
2. **bypass 运行机制**，当 shuffle read task 的数量小于等于 `spark.shuffle.sort.bypassMergeThreshold` 参数的值时（默认为 200），就会启用 bypass 机制；
3. **Tungsten Sort 运行机制**，开启此运行机制需设置配置项 `spark.shuffle.manager=tungsten-sort`。开启此项配置也不能保证就一定采用此运行机制（后面会解释）。

1. 普通运行机制

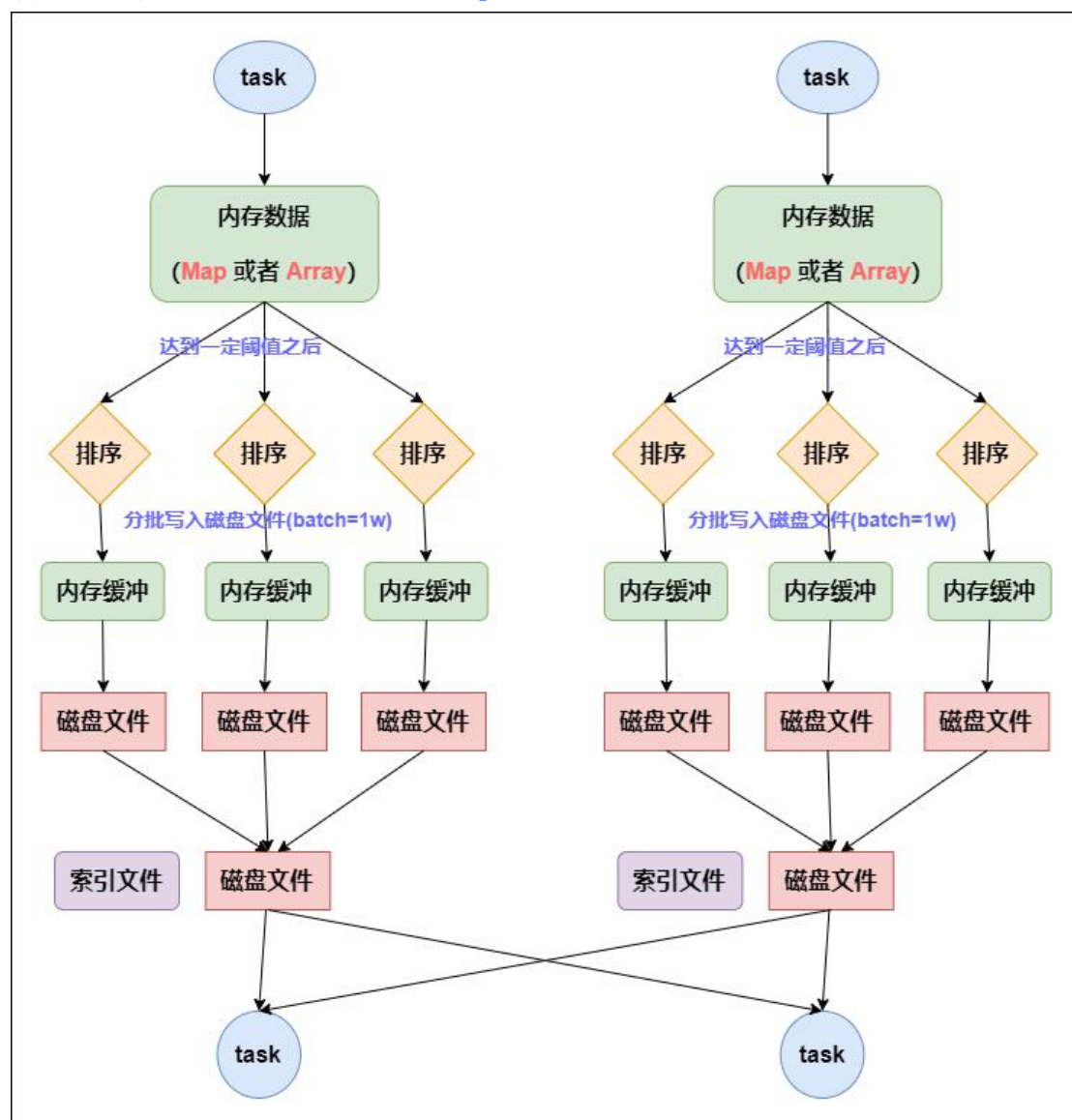
在该模式下，**数据会先写入一个内存数据结构中**，此时根据不同的 shuffle 算子，可能选用不同的数据结构。如果是 `reduceByKey` 这种聚合类的 shuffle 算子，那么会选用 Map 数据结构，一边通过 Map 进行聚合，一边写入内存；如果是 `join` 这种普通的 shuffle 算子，那么会选用 Array 数据结构，直接写入内存。接着，每写一条数据进入内存数据结构之后，就会判断一下，是否达到了某个临界阈值。如果达到临界阈值的话，那么就会尝试将内存数据结构中的数据溢写到磁盘，然后清空内存数据结构。

在溢写到磁盘文件之前，会先根据 key 对内存数据结构中已有的数据进行排序。排序过后，会分批将数据写入磁盘文件。默认的 batch 数量是 10000 条，也就是说，排序好的数据，会以每批 1 万条数据的形式分批写入磁盘文件。写入磁盘文件是通过 Java 的 `BufferedOutputStream` 实现的。**`BufferedOutputStream` 是 Java 的缓冲输出流，首先会将数据缓冲在内存中，当内存缓冲满溢之后再一次写入磁盘文件中，这样可以减少磁盘 IO 次数，提升性能。**

一个 task 将所有数据写入内存数据结构的过程中，会发生多次磁盘溢写操作，也就产生多个临时文件。最后会将之前所有的临时磁盘文件都进行合并，这就是 **merge 过程**，此时会将之前所有临时磁盘文件中的数据读取出来，然后依次写入最终的磁盘文件之中。此外，由于一个 task 就只对应一个磁盘文件，也就意味着该 task 为下游 stage 的 task 准备的数据都在这一个文件中，因此还会单独写一份**索引文件**，其中标识了下游各个 task 的数据在文件中的 start offset 与 end offset。

SortShuffleManager 由于有一个磁盘文件 merge 的过程,因此大大减少了文件数量。比如第一个 stage 有 50 个 task, 总共有 10 个 Executor, 每个 Executor 执行 5 个 task, 而第二个 stage 有 100 个 task。由于每个 task 最终只有一个磁盘文件, 因此此时每个 Executor 上只有 5 个磁盘文件, 所有 Executor 只有 50 个磁盘文件。

普通运行机制的 SortShuffleManager 工作原理如下图所示:



普通运行机制的 SortShuffleManager 工作原理

2. bypass 运行机制

Reducer 端任务数比较少的情况下, 基于 Hash Shuffle 实现机制明显比基于 Sort Shuffle 实现机制要快, 因此基于 Sort shuffle 实现机制提供了一个回退方案,

就是 **bypass 运行机制**。对于 Reducer 端任务数少于配置属性 `spark.shuffle.sort.bypassMergeThreshold` 设置的个数时，使用带 Hash 风格的回退计划。

bypass 运行机制的触发条件如下：

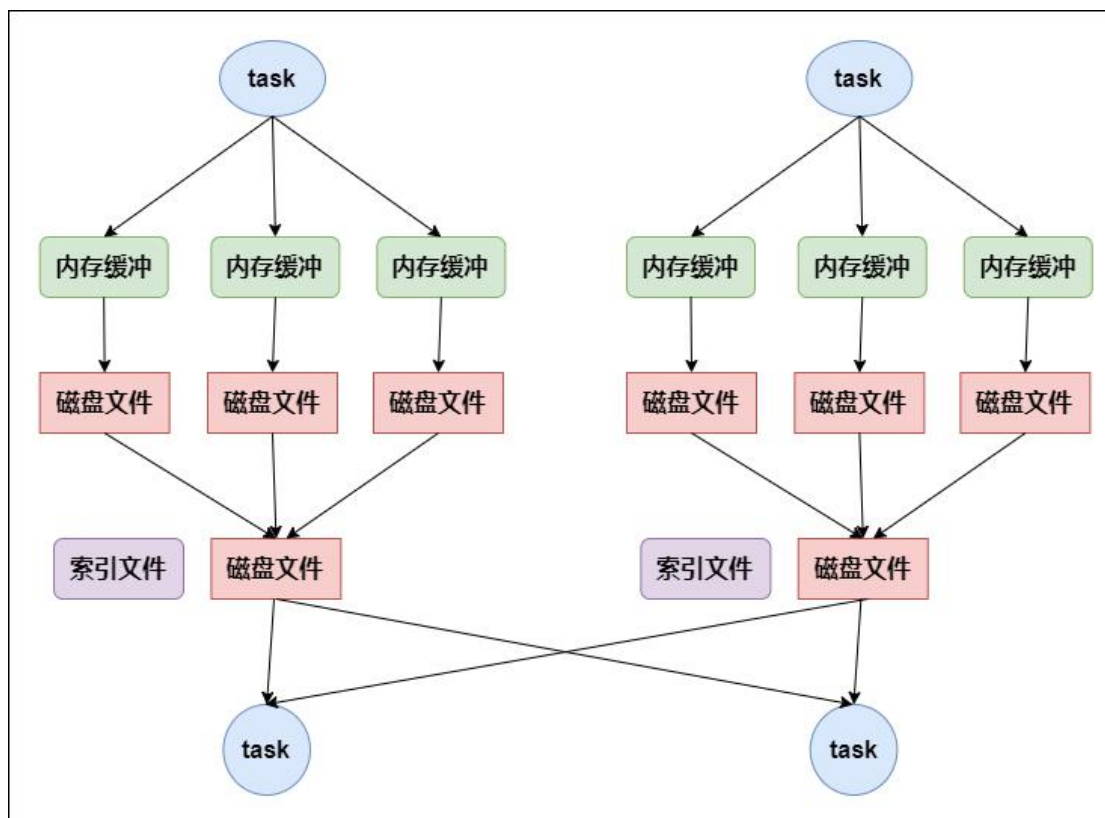
- shuffle map task 数量小于 `spark.shuffle.sort.bypassMergeThreshold=200` 参数的值。
- 不是聚合类的 shuffle 算子。

此时，每个 task 会为每个下游 task 都创建一个临时磁盘文件，并将数据按 key 进行 hash 然后根据 key 的 hash 值，将 key 写入对应的磁盘文件之中。当然，写入磁盘文件时也是先写入内存缓冲，缓冲写满之后再溢写到磁盘文件的。最后，同样会将所有临时磁盘文件都合并成一个磁盘文件，并创建一个单独的索引文件。

该过程的磁盘写机制其实跟未经优化的 HashShuffleManager 是一模一样的，因为都要创建数量惊人的磁盘文件，只是在最后会做一个磁盘文件的合并而已。因此少量的最终磁盘文件，也让该机制相对未经优化的 HashShuffleManager 来说，shuffle read 的性能会更好。

而该机制与普通 SortShuffleManager 运行机制的不同在于：第一，磁盘写机制不同；第二，不会进行排序。也就是说，**启用该机制的最大好处在于，shuffle write 过程中，不需要进行数据的排序操作**，也就节省掉了这部分的性能开销。

bypass 运行机制的 SortShuffleManager 工作原理如下图所示：



bypass 运行机制的 SortShuffleManager 工作原理

3. Tungsten Sort Shuffle 运行机制

基于 Tungsten Sort 的 Shuffle 实现机制主要是借助 Tungsten 项目所做的优化来高效处理 Shuffle。

Spark 提供了配置属性，用于选择具体的 Shuffle 实现机制，但需要说明的是，虽然默认情况下 Spark 默认开启的是基于 SortShuffle 实现机制，但实际上，参考 Shuffle 的框架内核部分可知基于 SortShuffle 的实现机制与基于 Tungsten Sort Shuffle 实现机制都是使用 SortShuffleManager，而内部使用的具体的实现机制，是通过提供的两个方法进行判断的：

对应非基于 Tungsten Sort 时，通过 `SortShuffleWriter.shouldBypassMergeSort` 方法判断是否需要回退到 Hash 风格的 Shuffle 实现机制，当该方法返回的条件不满足时，则通过 `SortShuffleManager.canUseSerializedShuffle` 方法判断是否需要采用基于 Tungsten Sort Shuffle 实现机制，而当这两个方法返回都为 false，即都不满足对应的条件时，会自动采用普通运行机制。

因此，当设置了 `spark.shuffle.manager=tungsten-sort` 时，也不能保证就一定采用基于 Tungsten Sort 的 Shuffle 实现机制。

要实现 Tungsten Sort Shuffle 机制需要满足以下条件：

1. Shuffle 依赖中不带聚合操作或没有对输出进行排序的要求。
2. Shuffle 的序列化器支持序列化值的重定位（当前仅支持 KryoSerializer Spark SQL 框架自定义的序列化器）。
3. Shuffle 过程中的输出分区个数少于 16777216 个。

实际上，使用过程中还有其他一些限制，如引入 Page 形式的内存管理模型后，内部单条记录的长度不能超过 128 MB（具体内存模型可以参考 PackedRecordPointer 类）。另外，分区个数的限制也是该内存模型导致的。所以，目前使用基于 Tungsten Sort Shuffle 实现机制条件还是比较苛刻的。

基于 Sort 的 Shuffle 机制的优缺点

优点：

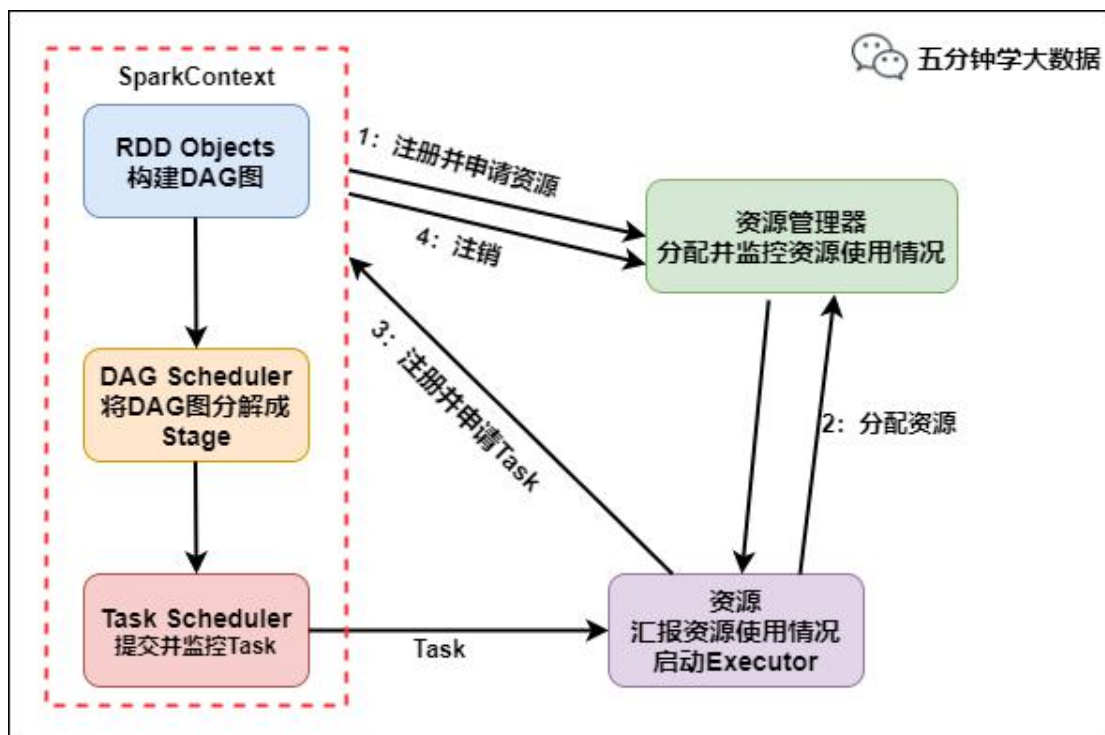
- 小文件的数量大量减少，Mapper 端的内存占用变少；
- Spark 不仅可以处理小规模的数据，即使处理大规模的数据，也不会很容易达到性能瓶颈。

缺点：

- 如果 Mapper 中 Task 的数量过大，依旧会产生很多小文件，此时在 Shuffle 传数据的过程中到 Reducer 端，Reducer 会需要同时大量地记录进行反序列化，导致大量内存消耗和 GC 负担巨大，造成系统缓慢，甚至崩溃；
- 强制了在 Mapper 端必须要排序，即使数据本身并不需要排序；
- 它要基于记录本身进行排序，这就是 Sort-Based Shuffle 最致命的性能消耗。

七、Spark 底层执行原理

Spark 运行流程



Spark 运行流程

具体运行流程如下：

1. SparkContext 向资源管理器注册并向资源管理器申请运行 Executor
2. 资源管理器分配 Executor，然后资源管理器启动 Executor
3. Executor 发送心跳至资源管理器
4. SparkContext 构建 DAG 有向无环图
5. 将 DAG 分解成 Stage (TaskSet)
6. 把 Stage 发送给 TaskScheduler
7. Executor 向 SparkContext 申请 Task
8. TaskScheduler 将 Task 发送给 Executor 运行
9. 同时 SparkContext 将应用程序代码发放给 Executor
10. Task 在 Executor 上运行，运行完毕释放所有资源

1. 从代码角度看 DAG 图的构建

```
Val lines1 = sc.textFile(inputPath1).map(...).map(...)
```

```
Val lines2 = sc.textFile(inputPath2).map(...)
```

```
Val lines3 = sc.textFile(inputPath3)
```



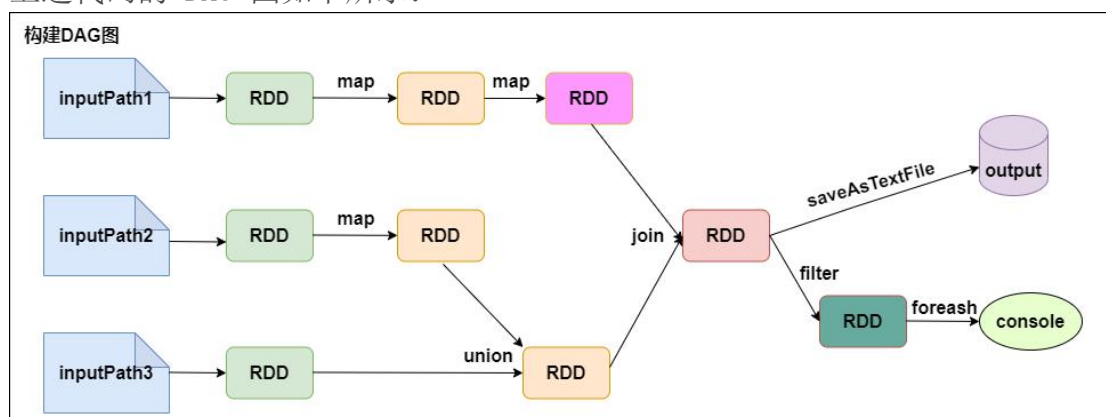
```
Val dtinone1 = lines2.union(lines3)
```

```
Val dtinone = lines1.join(dtinone1)
```

```
dtinone.saveAsTextFile(...)
```

```
dtinone.filter(...).foreach(...)
```

上述代码的 DAG 图如下所示：



构建 DAG 图

Spark 内核会在需要计算发生的时刻绘制一张关于计算路径的有向无环图，也就是如上图所示的 DAG。

Spark 的计算发生在 RDD 的 Action 操作，而对 Action 之前的所有 Transformation，Spark 只是记录下 RDD 生成的轨迹，而不会触发真正的计算。

2. 将 DAG 划分为 Stage 核心算法

一个 Application 可以有多个 job 多个 Stage：

Spark Application 中可以因为不同的 Action 触发众多的 job，一个 Application 中可以有很多的 job，每个 job 是由一个或者多个 Stage 构成的，后面的 Stage 依赖于前面的 Stage，也就是说只有前面依赖的 Stage 计算完毕后，后面的 Stage 才会运行。

划分依据：

Stage 划分的依据就是宽依赖，像 reduceByKey，groupByKey 等算子，会导致宽依赖的产生。

回顾下宽窄依赖的划分原则：

窄依赖：父 RDD 的一个分区只会被子 RDD 的一个分区依赖。即一对一或者多对一的关系，可理解为独生子女。常见的窄依赖有：map、filter、union、mapPartitions、

mapValues、join（父 RDD 是 hash-partitioned）等。

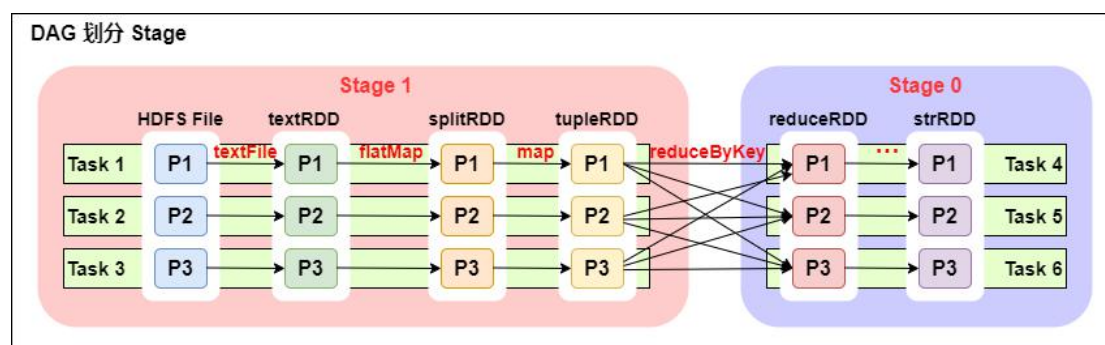
宽依赖：父 RDD 的一个分区会被子 RDD 的多个分区依赖（涉及到 shuffle）。即一对多的关系，可理解为超生。常见的宽依赖有 groupByKey、partitionBy、reduceByKey、join（父 RDD 不是 hash-partitioned）等。

核心算法：回溯算法

从后往前回溯/反向解析，遇到窄依赖加入本 Stage，遇见宽依赖进行 Stage 切分。

Spark 内核会从触发 Action 操作的那个 RDD 开始**从后往前推**，首先会为最后一个 RDD 创建一个 Stage，然后继续倒推，如果发现对某个 RDD 是宽依赖，那么就会将宽依赖的那个 RDD 创建一个新的 Stage，那个 RDD 就是新的 Stage 的最后一个 RDD。然后依次类推，继续倒推，根据窄依赖或者宽依赖进行 Stage 的划分，直到所有的 RDD 全部遍历完成为止。

3. 将 DAG 划分为 Stage 剖析



DAG 划分 Stage

一个 Spark 程序可以有多个 DAG（有几个 Action，就有几个 DAG，上图最后只有一个 Action（图中未表现），那么就是一个 DAG）。

一个 DAG 可以有多个 Stage（根据宽依赖/shuffle 进行划分）。

同一个 Stage 可以有多个 Task 并行执行（task 数=分区数，如上图，Stage1 中有三个分区 P1、P2、P3，对应的也有三个 Task）。

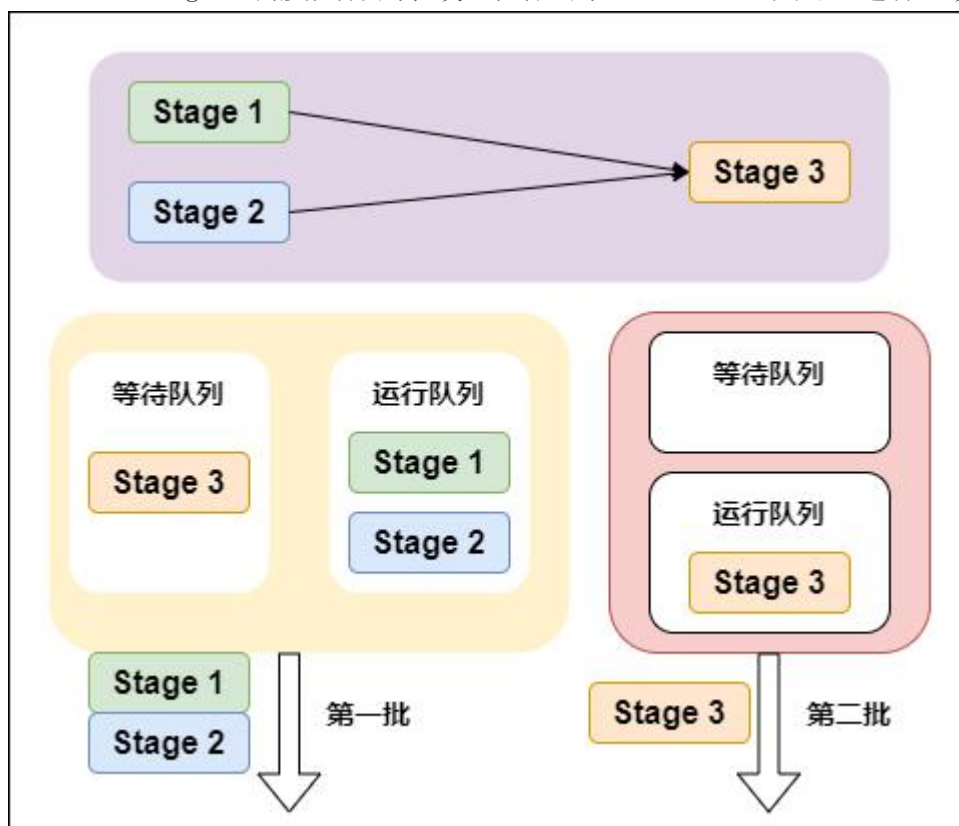
可以看到这个 DAG 中只 reduceByKey 操作是一个宽依赖，Spark 内核会以此为边界将其前后划分成不同的 Stage。

同时我们可以注意到，在图中 Stage1 中，从 textFile 到 flatMap 到 map 都是窄依赖，这几步操作可以形成一个流水线操作，通过 flatMap 操作生成的 partition 可以不用等待整个 RDD 计算结束，而是继续进行 map 操作，这样大大提高了计算的效率。

4. 提交 Stages

调度阶段的提交，最终会被转换成一个任务集的提交，DAGScheduler 通过 TaskScheduler 接口提交任务集，这个任务集最终会触发 TaskScheduler 构建一个 TaskSetManager 的实例来管理这个任务集的生命周期，对于 DAGScheduler 来说，提交调度阶段的工作到此就完成了。

而 TaskScheduler 的具体实现则会在得到计算资源的时候，进一步通过 TaskSetManager 调度具体的任务到对应的 Executor 节点上进行运算。



5. 监控 Job、Task、Executor

1. DAGScheduler 监控 Job 与 Task:

要保证相互依赖的作业调度阶段能够得到顺利的调度执行，DAGScheduler 需要监控当前作业调度阶段乃至任务的完成情况。

这通过对外暴露一系列的回调函数来实现的，对于 TaskScheduler 来说，这些回调函数主要包括任务的开始结束失败、任务集的失败，DAGScheduler 根据这些任务的生命周期信息进一步维护作业和调度阶段的状态信息。

2. DAGScheduler 监控 Executor 的生命状态:

TaskScheduler 通过回调函数通知 DAGScheduler 具体的 Executor 的生命状态, 如果某一个 Executor 崩溃了, 则对应的调度阶段任务集的 ShuffleMapTask 的输出结果也将标志为不可用, 这将导致对应任务集状态的变更, 进而重新执行相关计算任务, 以获取丢失的相关数据。

6. 获取任务执行结果

1. 结果 DAGScheduler:

一个具体的任务在 Executor 中执行完毕后, 其结果需要以某种形式返回给 DAGScheduler, 根据任务类型的不同, 任务结果的返回方式也不同。

2. 两种结果, 中间结果与最终结果:

对于 FinalStage 所对应的任务, 返回给 DAGScheduler 的是运算结果本身。而对于中间调度阶段对应的任务 ShuffleMapTask, 返回给 DAGScheduler 的是一个 MapStatus 里的相关存储信息, 而非结果本身, 这些存储位置信息将作为下一个调度阶段的任务获取输入数据的依据。

3. 两种类型, DirectTaskResult 与 IndirectTaskResult:

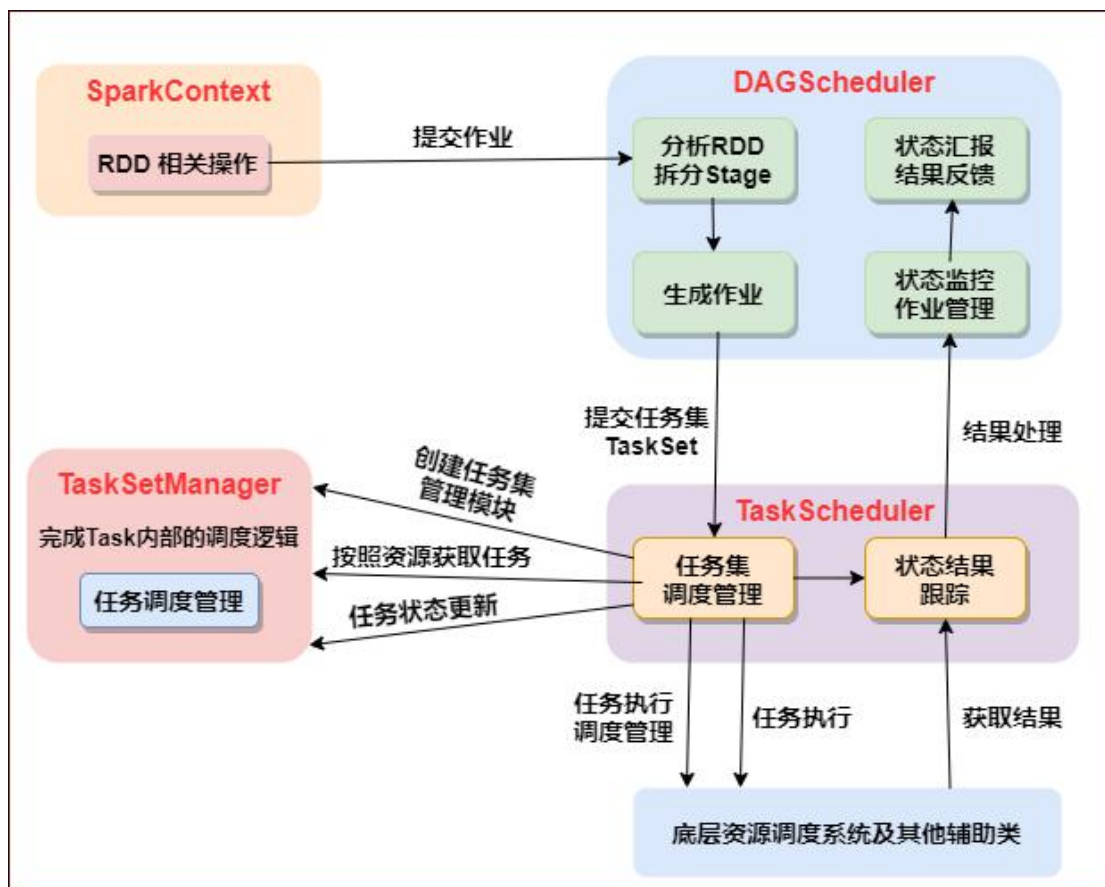
根据任务结果大小的不同, ResultTask 返回的结果又分为两类:

如果结果足够小, 则直接放在 DirectTaskResult 对象内中。

如果超过特定尺寸则在 Executor 端会将 DirectTaskResult 先序列化, 再把序列化的结果作为一个数据块存放在 BlockManager 中, 然后将 BlockManager 返回的 BlockID 放在 IndirectTaskResult 对象中返回给 TaskScheduler, TaskScheduler 进而调用 TaskResultGetter 将 IndirectTaskResult 中的 BlockID 取出并通过 BlockManager 最终取得对应的 DirectTaskResult。

7. 任务调度总体诠释

一张图说明任务总体调度:



任务总体调度

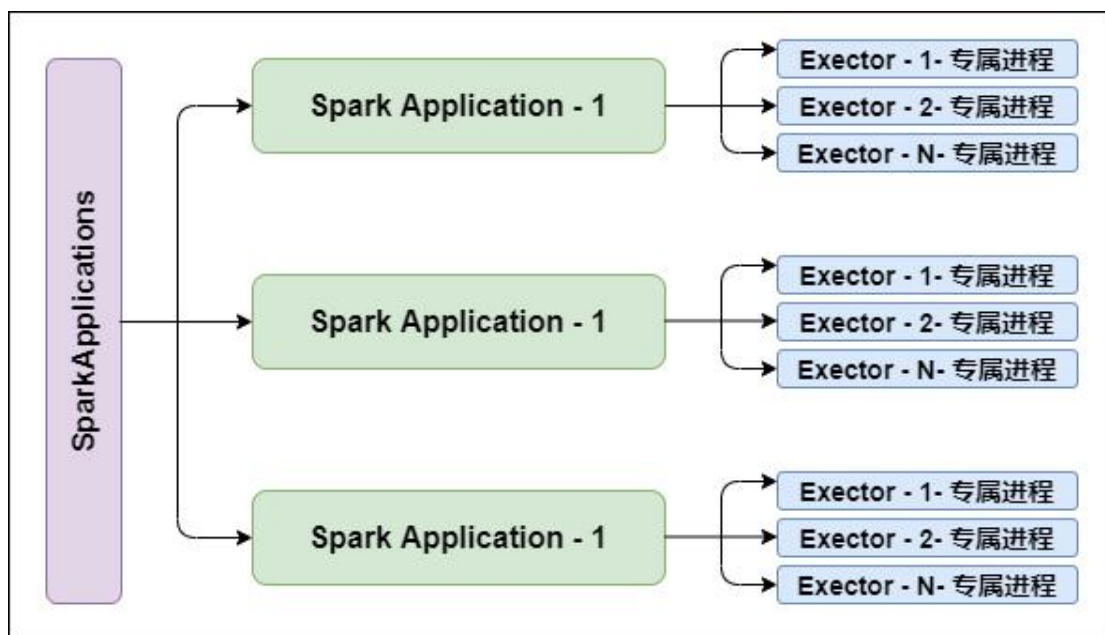
Spark 运行架构特点

1. Executor 进程专属

每个 Application 获取专属的 Executor 进程，该进程在 Application 期间一直驻留，并以多线程方式运行 Tasks。

Spark Application 不能跨应用程序共享数据，除非将数据写入到外部存储系统。

如图所示：

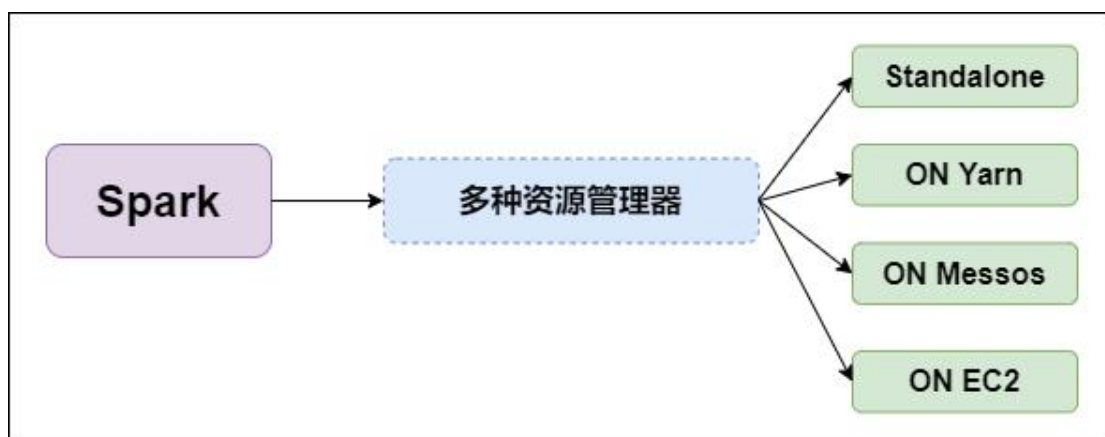


Executor 进程专属

2. 支持多种资源管理器

Spark 与资源管理器无关，只要能够获取 Executor 进程，并能保持相互通信就可以了。

Spark 支持资源管理器包含： Standalone、On Mesos、On YARN、Or On EC2。
如图所示：

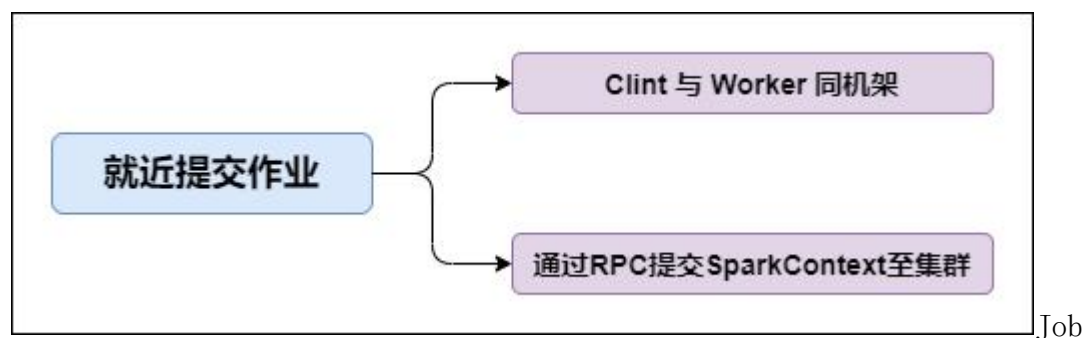


支持多种资源管理器

3. Job 提交就近原则

提交 SparkContext 的 Client 应该靠近 Worker 节点(运行 Executor 的节点), 最好是在同一个 Rack(机架)里, 因为 Spark Application 运行过程中 SparkContext 和 Executor 之间有大量的信息交换; 如果想在远程集群中运行, 最好使用 RPC 将 SparkContext 提交给集群, **不要远离 Worker 运行 SparkContext**。

如图所示:



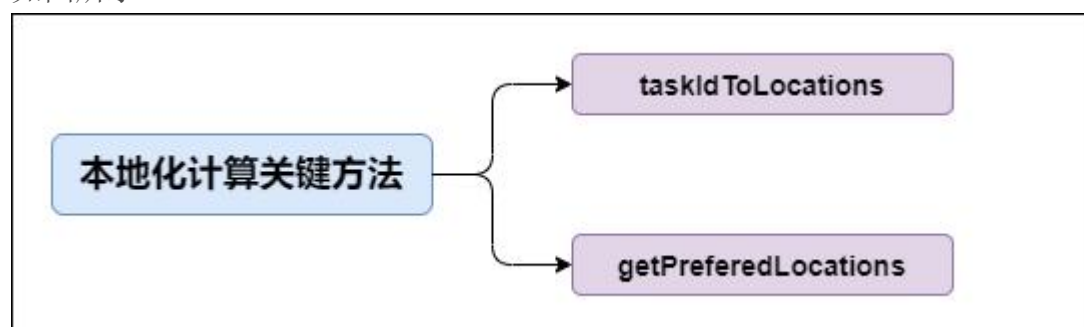
提交就近原则

4. 移动程序而非移动数据的原则执行

移动程序而非移动数据的原则执行, Task 采用了数据本地性和推测执行的优化机制。

关键方法: taskIdToLocations、getPreferedLocations。

如图所示:



数据本地性

八、Spark 数据倾斜

就是数据分到各个区的数量不太均匀, 可以自定义分区器, 想怎么分就怎么分。

Spark 中的数据倾斜问题主要指 shuffle 过程中出现的数据倾斜问题, 是由于不同的 key 对应的数据量不同导致的不同 task 所处理的数据量不同的问题。

例如，reduced 端一共要处理 100 万条数据，第一个和第二个 task 分别被分配到了 1 万条数据，计算 5 分钟内完成，第三个 task 分配到了 98 万数据，此时第三个 task 可能需要 10 个小时完成，这使得整个 Spark 作业需要 10 个小时才能运行完成，这就是数据倾斜所带来的后果。

注意，要区分开**数据倾斜**与**数据过量**这两种情况，数据倾斜是指少数 task 被分配了绝大多数的数据，因此少数 task 运行缓慢；数据过量是指所有 task 被分配的数据量都很大，相差不多，所有 task 都运行缓慢。

数据倾斜的表现：

1. Spark 作业的大部分 task 都执行迅速，只有有限的几个 task 执行的非常慢，此时可能出现了数据倾斜，作业可以运行，但是运行得非常慢；
2. Spark 作业的大部分 task 都执行迅速，但是有的 task 在运行过程中会突然报出 OOM，反复执行几次都在某一个 task 报出 OOM 错误，此时可能出现了数据倾斜，作业无法正常运行。 定位数据倾斜问题：
3. 查阅代码中的 shuffle 算子，例如 reduceByKey、countByKey、groupByKey、join 等算子，根据代码逻辑判断此处是否会出现数据倾斜；
4. 查看 Spark 作业的 log 文件，log 文件对于错误的记录会精确到代码的某一行，可以根据异常定位到的代码位置来明确错误发生在第几个 stage，对应的 shuffle 算子是哪一个；

1. 预聚合原始数据

1. 避免 shuffle 过程

绝大多数情况下，Spark 作业的数据来源都是 Hive 表，这些 Hive 表基本都是经过 ETL 之后的昨天的数据。 为了避免数据倾斜，我们可以考虑避免 shuffle 过程，如果避免了 shuffle 过程，那么从根本上就消除了发生数据倾斜问题的可能。如果 Spark 作业的数据来源于 Hive 表，那么可以先在 Hive 表中对数据进行聚合，例如按照 key 进行分组，将同一 key 对应的所有 value 用一种特殊的格式拼接到一个字符串里去，这样，一个 key 就只有一条数据了；之后，对一个 key 的所有 value 进行处理时，只需要进行 map 操作即可，无需再进行任何的 shuffle 操作。通过上述方式就避免了执行 shuffle 操作，也就不可能会发生任何的数据倾斜问题。

对于 Hive 表中数据的操作，不一定是拼接成一个字符串，也可以是直接对 key 的每一条数据进行累计计算。 **要区分开，处理的数据量大和数据倾斜的区别。**

2. 增大 key 粒度（减小数据倾斜可能性，增大每个 task 的数据量）

如果没有办法对每个 key 聚合出来一条数据，在特定场景下，可以考虑扩大 key 的聚合粒度。

例如，目前有 10 万条用户数据，当前 key 的粒度是（省，城市，区，日期），现在我们考虑扩大粒度，将 key 的粒度扩大为（省，城市，日期），这样的话，key 的数量会减少，key 之间的数据量差异也有可能会减少，由此可以减轻数据倾斜的现象和问题。（此方法只针对特定类型的数据有效，当应用场景不适宜时，会加重数据倾斜）

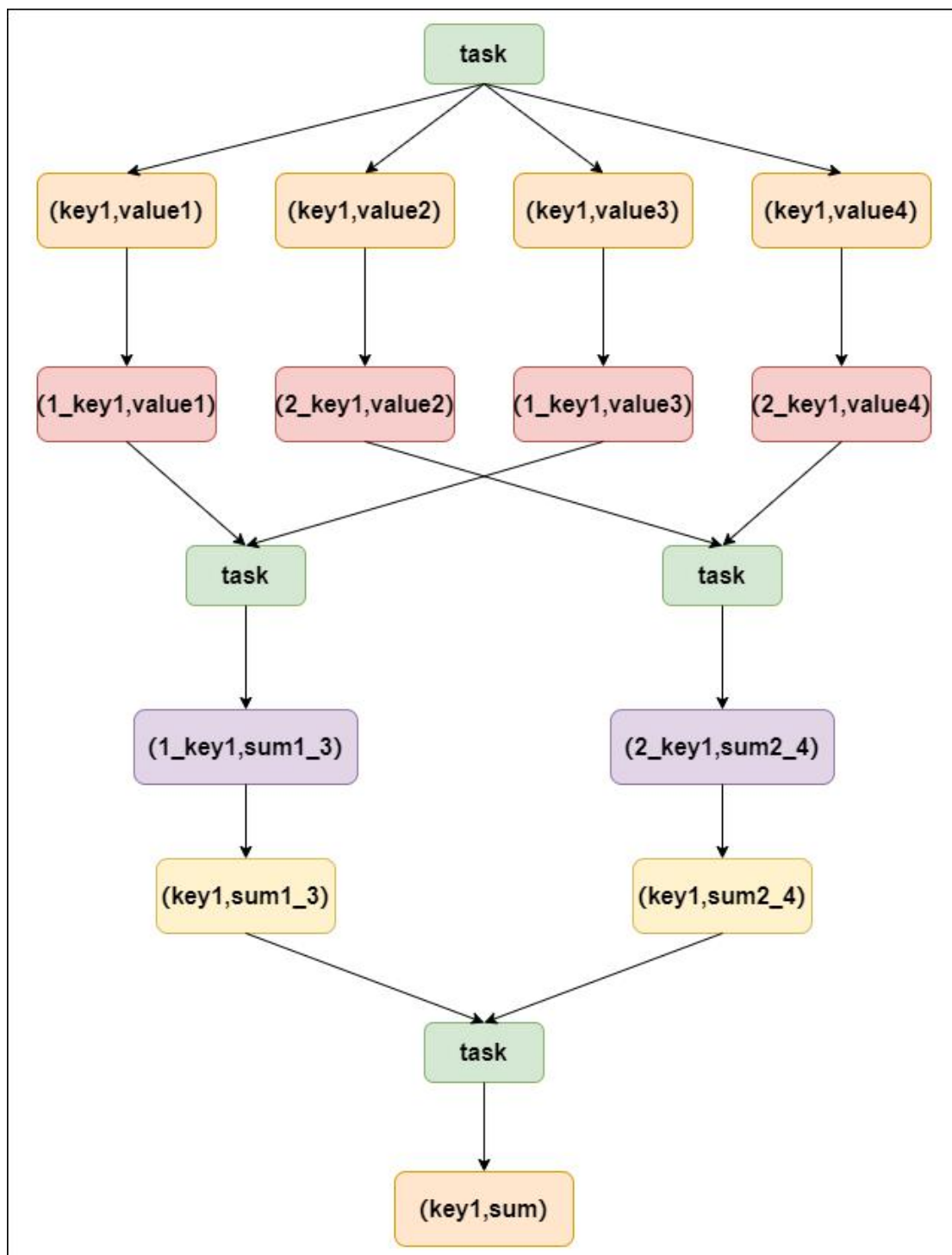
2. 预处理导致倾斜的 key

1. 过滤

如果在 Spark 作业中允许丢弃某些数据，那么可以考虑将可能导致数据倾斜的 key 进行过滤，滤除可能导致数据倾斜的 key 对应的数据，这样，在 Spark 作业中就不会发生数据倾斜了。

2. 使用随机 key

当使用了类似于 groupByKey、reduceByKey 这样的算子时，可以考虑使用随机 key 实现双重聚合，如下图所示：



随机 key 实现双重聚合

首先，通过 map 算子给每个数据的 key 添加随机数前缀，对 key 进行打散，将原先一样的 key 变成不一样的 key，然后进行第一次聚合，这样就可以让原本被一个 task 处理的数据分散到多个 task 上去做局部聚合；随后，去除掉每个 key 的前缀，再次进行聚合。

此方法对于由 `groupByKey`、`reduceByKey` 这类算子造成的数据倾斜有比较好的效果，仅仅适用于聚合类的 `shuffle` 操作，适用范围相对较窄。如果是 `join` 类的 `shuffle` 操作，还得用其他的解决方案。

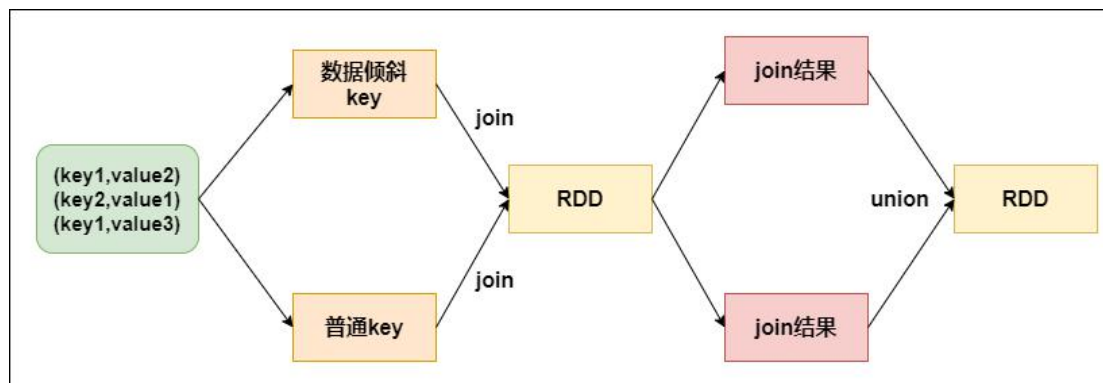
此方法也是前几种方案没有比较好的效果时要尝试的解决方案。

3. sample 采样对倾斜 key 单独进行 join

在 Spark 中，如果某个 RDD 只有一个 key，那么在 `shuffle` 过程中会默认将此 key 对应的数据打散，由不同的 `reduce` 端 task 进行处理。

所以当由单个 key 导致数据倾斜时，可有将发生数据倾斜的 key 单独提取出来，组成一个 RDD，然后用这个原本会导致倾斜的 key 组成的 RDD 和其他 RDD 单独 `join`，此时，根据 Spark 的运行机制，此 RDD 中的数据会在 `shuffle` 阶段被分散到多个 task 中去进行 `join` 操作。

倾斜 key 单独 `join` 的流程如下图所示：



倾斜 key 单独 `join` 流程

适用场景分析：

对于 RDD 中的数据，可以将其转换为一个中间表，或者是直接使用 `countByKey()` 的方式，看一下这个 RDD 中各个 key 对应的数据量，此时如果你发现整个 RDD 就一个 key 的数据量特别多，那么就可以考虑使用这种方法。

当数据量非常大时，可以考虑使用 `sample` 采样获取 10% 的数据，然后分析这 10% 的数据中哪个 key 可能会导致数据倾斜，然后将这个 key 对应的数据单独提取出来。

不适用场景分析：

如果一个 RDD 中导致数据倾斜的 key 很多，那么此方案不适用。

3. 提高 `reduce` 并行度

当方案一和方案二对于数据倾斜的处理没有很好的效果时，可以考虑提高 shuffle 过程中的 reduce 端并行度，reduce 端并行度的提高就增加了 reduce 端 task 的数量，那么每个 task 分配到的数据量就会相应减少，由此缓解数据倾斜问题。

1. reduce 端并行度的设置

在大部分的 shuffle 算子中，都可以传入一个并行度的设置参数，比如 `reduceByKey(500)`，这个参数会决定 shuffle 过程中 reduce 端的并行度，在进行 shuffle 操作的时候，就会对应着创建指定数量的 reduce task。对于 Spark SQL 中的 shuffle 类语句，比如 `group by`、`join` 等，需要设置一个参数，即 `spark.sql.shuffle.partitions`，该参数代表了 shuffle read task 的并行度，该值默认是 200，对于很多场景来说都有点过小。

增加 shuffle read task 的数量，可以让原本分配给一个 task 的多个 key 分配给多个 task，从而让每个 task 处理比原来更少的数据。

举例来说，如果原本有 5 个 key，每个 key 对应 10 条数据，这 5 个 key 都是分配给一个 task 的，那么这个 task 就要处理 50 条数据。而增加了 shuffle read task 以后，每个 task 就分配到一个 key，即每个 task 就处理 10 条数据，那么自然每个 task 的执行时间都会变短了。

2. reduce 端并行度设置存在的缺陷

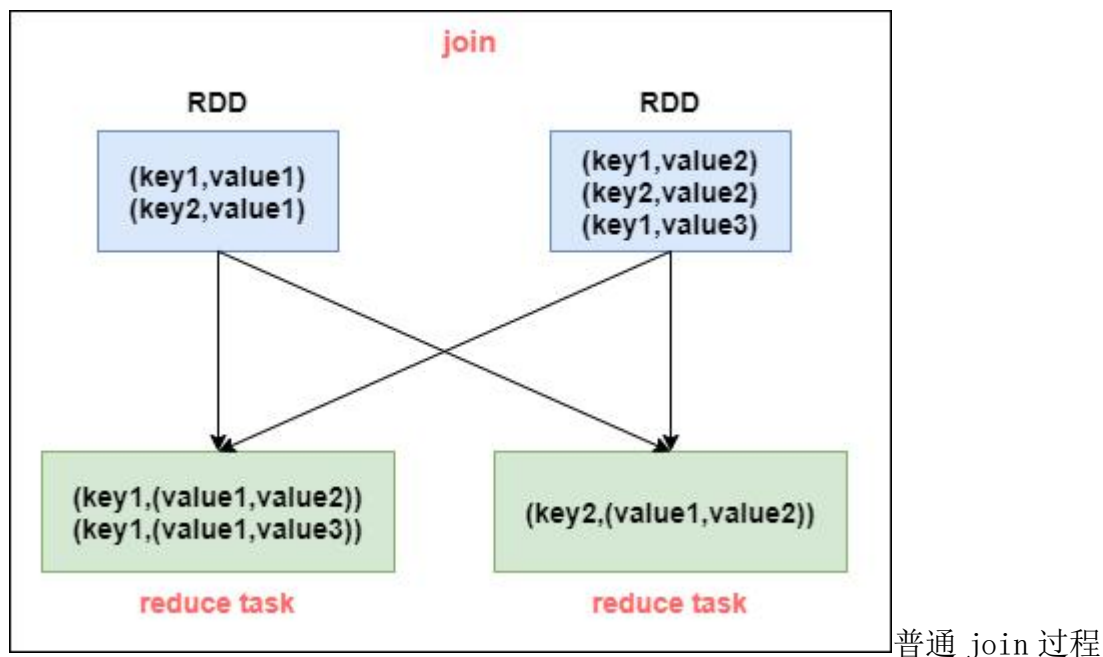
提高 reduce 端并行度并没有从根本上改变数据倾斜的本质和问题（方案一和方案二从根本上避免了数据倾斜的发生），只是尽可能地去缓解和减轻 shuffle reduce task 的数据压力，以及数据倾斜的问题，适用于有较多 key 对应的数据量都比较大的情况。

该方案通常无法彻底解决数据倾斜，因为如果出现一些极端情况，比如某个 key 对应的数据量有 100 万，那么无论你的 task 数量增加到多少，这个对应着 100 万数据的 key 肯定还是会分配到一个 task 中去处理，因此注定还是会发生数据倾斜的。所以这种方案只能说是在发现数据倾斜时尝试使用的一种手段，尝试去用最简单的方法缓解数据倾斜而已，或者是和其他方案结合起来使用。

在理想情况下，reduce 端并行度提升后，会在一定程度上减轻数据倾斜的问题，甚至基本消除数据倾斜；但是，在一些情况下，只会让原来由于数据倾斜而运行缓慢的 task 运行速度稍有提升，或者避免了某些 task 的 OOM 问题，但是，仍然运行缓慢，此时，要及时放弃方案三，开始尝试后面的方案。

4. 使用 map join

正常情况下，join 操作都会执行 shuffle 过程，并且执行的是 reduce join，也就是先将所有相同的 key 和对应的 value 汇聚到一个 reduce task 中，然后再进行 join。普通 join 的过程如下图所示：



普通的 join 是会走 shuffle 过程的，而一旦 shuffle，就相当于会将相同 key 的数据拉取到一个 shuffle read task 中再进行 join，此时就是 reduce join。但是如果一个 RDD 是比较小的，则可以采用广播小 RDD 全量数据+map 算子来实现与 join 同样的效果，也就是 map join，此时就不会发生 shuffle 操作，也就不会发生数据倾斜。

注意：RDD 是并不能直接进行广播的，只能将 RDD 内部的数据通过 collect 拉取到 Driver 内存然后再进行广播。

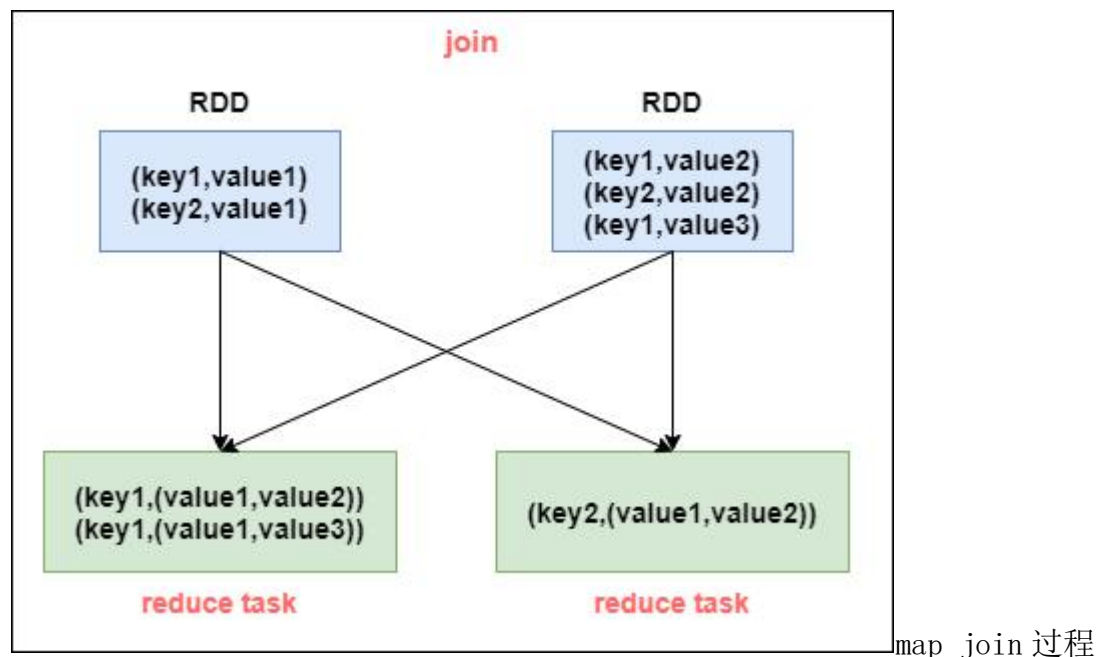
1. 核心思路：

不使用 join 算子进行连接操作，而使用 broadcast 变量与 map 类算子实现 join 操作，进而完全规避掉 shuffle 类的操作，彻底避免数据倾斜的发生和出现。将较小 RDD 中的数据直接通过 collect 算子拉取到 Driver 端的内存中来，然后对其创建一个 broadcast 变量；接着对另外一个 RDD 执行 map 类算子，在算子函数内，从 broadcast 变量中获取较小 RDD 的全量数据，与当前 RDD 的每一条数据按照连接 key 进行比对，如果连接 key 相同的话，那么就将两个 RDD 的数据用你需要的方式连接起来。

根据上述思路，根本不会发生 shuffle 操作，从根本上杜绝了 join 操作可能导致的数据倾斜问题。

当 join 操作有数据倾斜问题并且其中一个 RDD 的数据量较小时，可以优先考虑这种方式，效果非常好。

map join 的过程如下图所示：



2. 不适用场景分析：

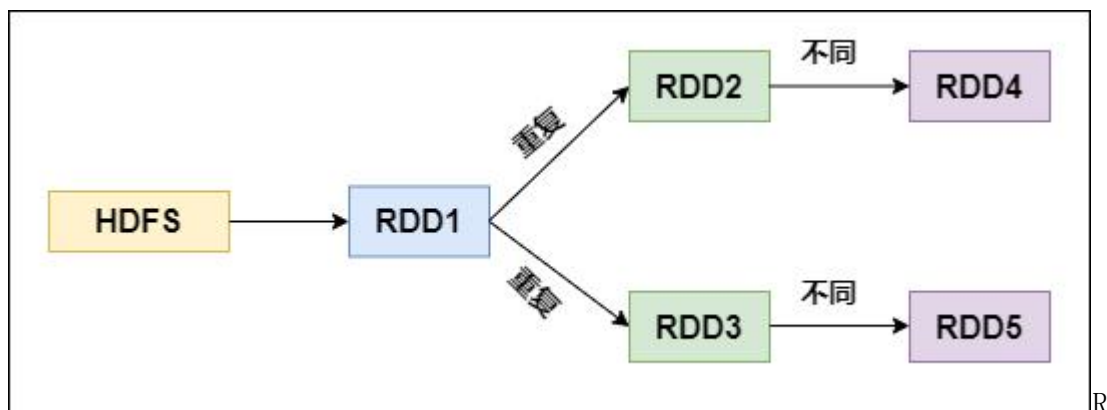
由于 Spark 的广播变量是在每个 Executor 中保存一个副本，如果两个 RDD 数据量都比较大，那么如果将一个数据量比较大的 RDD 做成广播变量，那么很有可能会造成内存溢出。

九、Spark 性能优化

Spark 调优之 RDD 算子调优

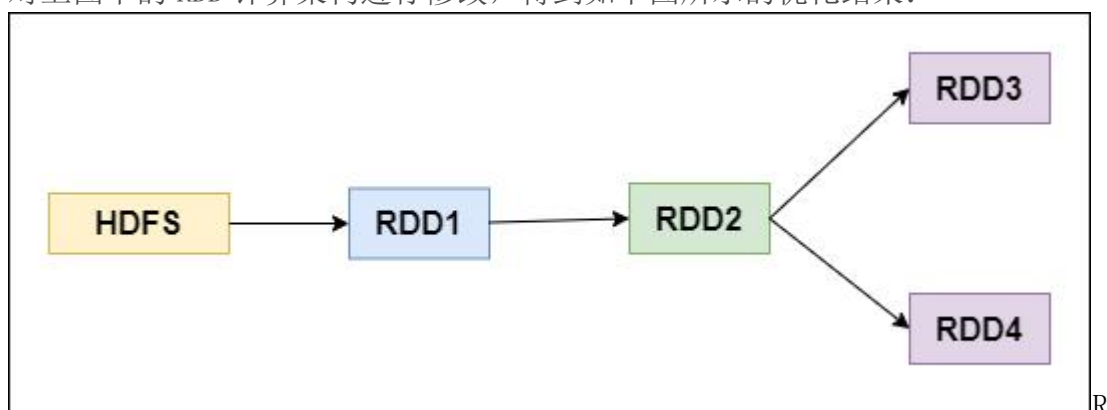
1. RDD 复用

在对 RDD 进行算子时，要避免相同的算子和计算逻辑之下对 RDD 进行重复的计算，如下图所示：



DD 的重复计算

对上图中的 RDD 计算架构进行修改，得到如下图所示的优化结果：



DD 架构优化

2. 尽早 filter

获取到初始 RDD 后，应该考虑**尽早地过滤掉不需要的数据**，进而减少对内存的占用，从而提升 Spark 作业的运行效率。

本文首发于公众号：五分钟学大数据，欢迎围观！回复【书籍】即可获得上百本大数据书籍

3. 读取大量小文件-用 wholeTextFiles

当我们将一个文本文件读取为 RDD 时，输入的每一行都会成为 RDD 的一个元素。也可以将多个完整的文本文件一次性读取为一个 pairRDD，其中键是文件名，值是文件内容。

```
val input:RDD[String] = sc.textFile("dir/*.log")
```

如果传递目录，则将目录下的所有文件读取作为 RDD。文件路径支持通配符。但是这样对于大量的小文件读取效率并不高，应该使用 **wholeTextFiles** 返回值为 RDD[(String, String)]，其中 Key 是文件的名称，Value 是文件的内容。

```
def wholeTextFiles(path: String, minPartitions: Int = defaultMinPartitions): RDD[(String, String)]
```

wholeTextFiles 读取小文件：

```
val filesRDD: RDD[(String, String)] =
sc.wholeTextFiles("D:\\data\\files", minPartitions = 3)
val linesRDD: RDD[String] = filesRDD.flatMap(_._2.split("\\r\\n"))
val wordsRDD: RDD[String] = linesRDD.flatMap(_.split(" "))
wordsRDD.map(_._1).reduceByKey(_ + _).collect().foreach(println)
```

4. mapPartition 和 foreachPartition

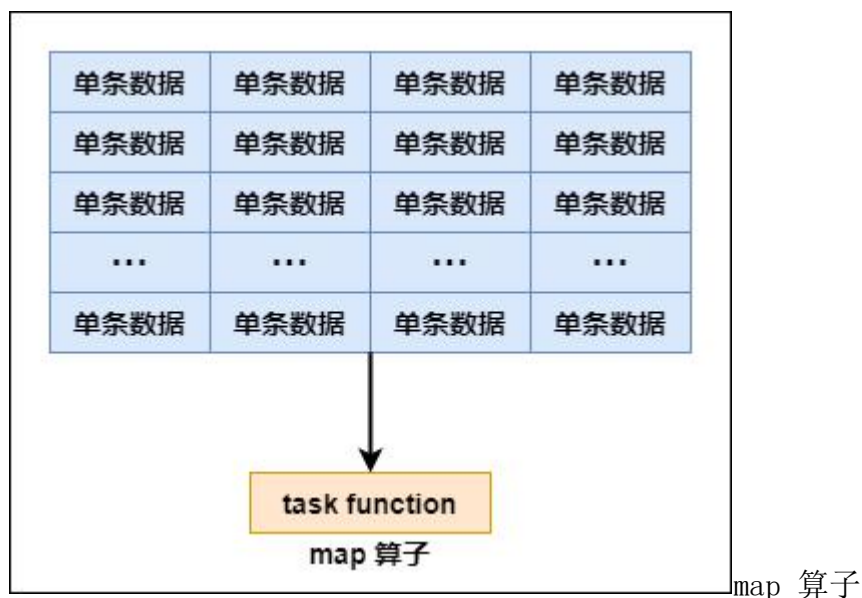
- mapPartitions

map(_...) 表示每一个元素

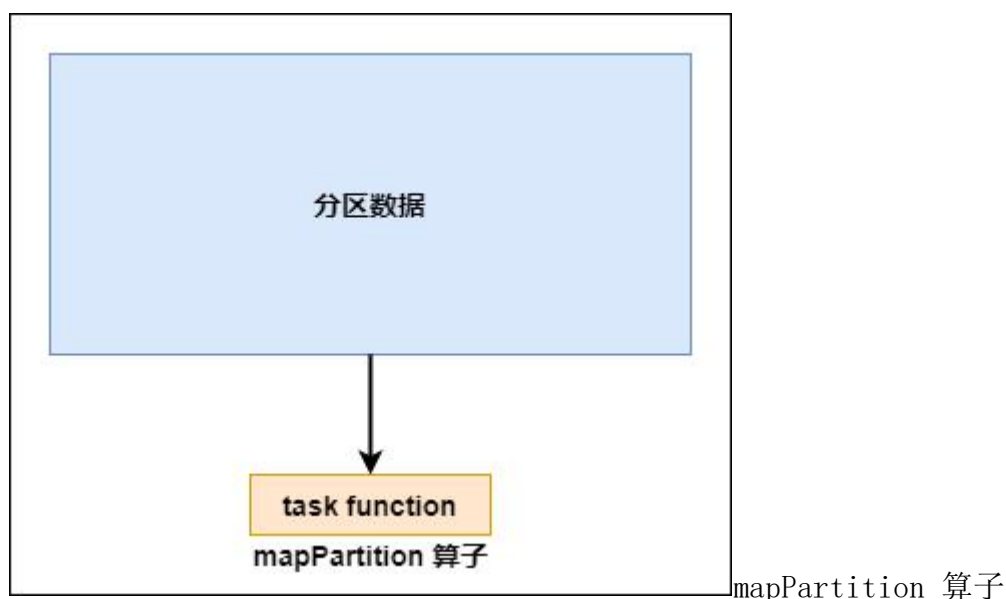
mapPartitions(_...) 表示每个分区的数据组成的迭代器

普通的 map 算子对 RDD 中的每一个元素进行操作，而 mapPartitions 算子对 RDD 中每一个分区进行操作。

如果是普通的 map 算子，假设一个 partition 有 1 万条数据，那么 map 算子中的 function 要执行 1 万次，也就是对每个元素进行操作。



如果是 mapPartition 算子，由于一个 task 处理一个 RDD 的 partition，那么一个 task 只会执行一次 function，function 一次接收所有的 partition 数据，效率比较高。



比如，当要把 RDD 中的所有数据通过 JDBC 写入数据，如果使用 map 算子，那么需要对 RDD 中的每一个元素都创建一个数据库连接，这样对资源的消耗很大，如果使用 mapPartitions 算子，那么针对一个分区的数据，只需要建立一个数据库连接。mapPartitions 算子也存在一些缺点：对于普通的 map 操作，一次处理一条数据，如果在处理了 2000 条数据后内存不足，那么可以将已经处理完的 2000 条数据从内存中垃圾回收掉；但是如果使用 mapPartitions 算子，但数据量非常大时，function 一次处理一个分区的数据，如果一旦内存不足，此时无法回收内存，就可能会 OOM，即内存溢出。

因此，mapPartitions 算子适用于数据量不是特别大的时候，此时使用 mapPartitions 算子对性能的提升效果还是不错的。（当数据量很大的时候，一旦使用 mapPartitions 算子，就会直接 OOM）

在项目中，应该首先估算一下 RDD 的数据量、每个 partition 的数据量，以及分配给每个 Executor 的内存资源，如果资源允许，可以考虑使用 mapPartitions 算子代替 map。

- foreachPartition

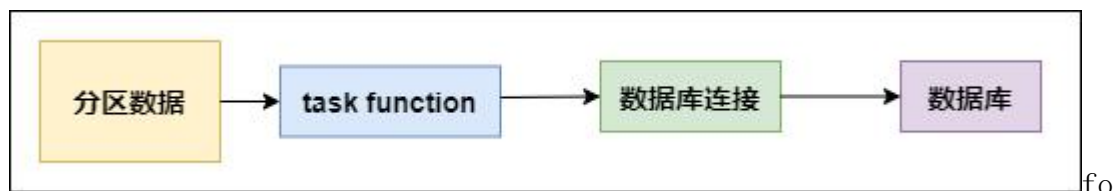
rrd.foreach(_....) 表示每一个元素

rrd.forPartitions(_....) 表示每个分区的数据组成的迭代器

在生产环境中，通常使用 foreachPartition 算子来完成数据库的写入，通过 foreachPartition 算子的特性，可以优化写数据库的性能。

如果使用 foreach 算子完成数据库的操作，由于 foreach 算子是遍历 RDD 的每条数据，因此，每条数据都会建立一个数据库连接，这是对资源的极大浪费，因此，**对于写数据库操作，我们应当使用 foreachPartition 算子。**

与 mapPartitions 算子非常相似，foreachPartition 是将 RDD 的每个分区作为遍历对象，一次处理一个分区的数据，也就是说，如果涉及数据库的相关操作，一个分区的数据只需要创建一次数据库连接，如下图所示：



foreachPartition 算子

使用了 foreachPartition 算子后，可以获得以下的性能提升：

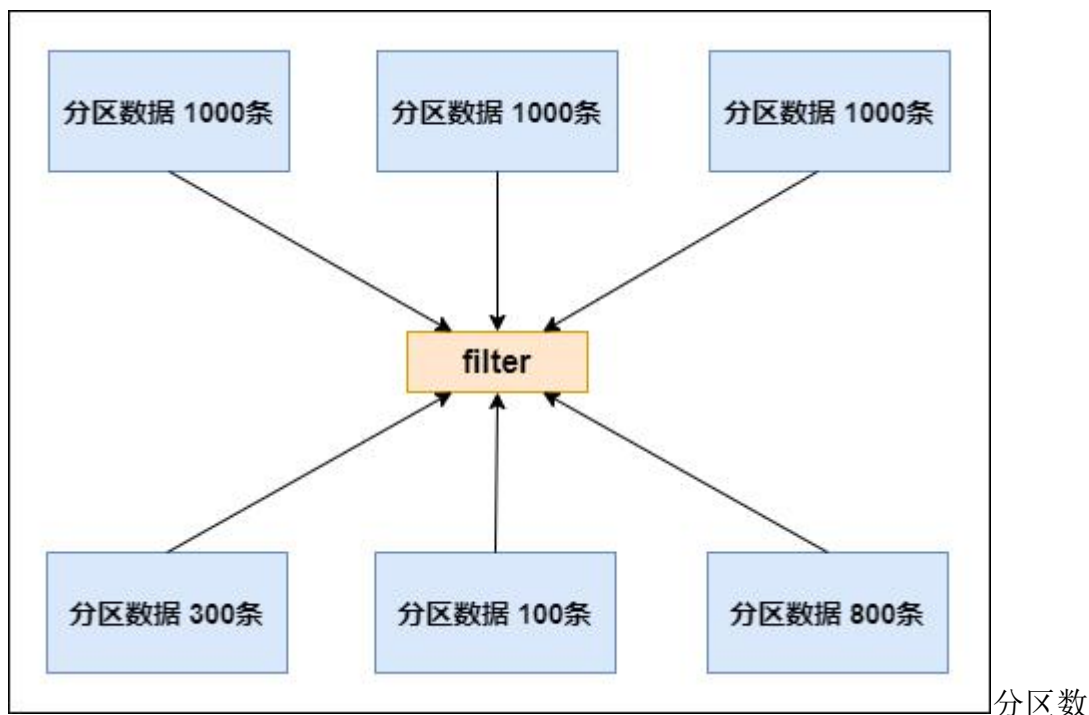
1. 对于我们写的 function 函数，一次处理一整个分区的数据；
2. 对于一个分区内的数据，创建唯一的数据库连接；
3. 只需要向数据库发送一次 SQL 语句和多组参数；

在生产环境中，全部都会使用 foreachPartition 算子完成数据库操作。

foreachPartition 算子存在一个问题，与 mapPartitions 算子类似，如果一个分区的数据量特别大，可能会造成 OOM，即内存溢出。

5. filter+coalesce/repartition(减少分区)

在 Spark 任务中我们经常会使用 filter 算子完成 RDD 中数据的过滤，在任务初始阶段，从各个分区中加载到的数据量是相近的，但是一旦进过 filter 过滤后，每个分区的数据量有可能会存在较大差异，如下图所示：



据过滤结果

根据上图我们可以发现两个问题：

1. 每个 partition 的数据量变小了，如果还按照之前与 partition 相等的 task 个数去处理当前数据，有点浪费 task 的计算资源；
2. 每个 partition 的数据量不一样，会导致后面的每个 task 处理每个 partition 数据的时候，每个 task 要处理的数据量不同，这很有可能导致数据倾斜问题。

如上图所示，第二个分区的数据过滤后只剩 100 条，而第三个分区的数据过滤后剩下 800 条，在相同的处理逻辑下，第二个分区对应的 task 处理的数据量与第三个分区对应的 task 处理的数据量差距达到了 8 倍，这也会导致运行速度可能存在数倍的差距，这也就是**数据倾斜问题**。

针对上述的两个问题，我们分别进行分析：

1. 针对第一个问题，既然分区的数据量变小了，我们希望对分区数据进行重新分配，比如将原来 4 个分区的数据转化到 2 个分区中，这样只需要用后面的两个 task 进行处理即可，避免了资源的浪费。
2. 针对第二个问题，解决方法和第一个问题的解决方法非常相似，对分区数据重新分配，让每个 partition 中的数据量差不多，这就避免了数据倾斜问题。

那么具体应该如何实现上面的解决思路？我们需要 coalesce 算子。

repartition 与 coalesce 都可以用来进行重分区，其中 repartition 只是 coalesce 接口中 shuffle 为 true 的简易实现，coalesce 默认情况下不进行 shuffle，但是可以通过参数进行设置。

假设我们希望将原本的分区个数 A 通过重新分区变为 B，那么有以下几种情况：

1. $A > B$ （多数分区合并为少数分区）

- A 与 B 相差值不大

此时使用 coalesce 即可，无需 shuffle 过程。

- A 与 B 相差值很大

此时可以使用 coalesce 并且不启用 shuffle 过程，但是会导致合并过程性能低下，所以推荐设置 coalesce 的第二个参数为 true，即启动 shuffle 过程。

2. $A < B$ （少数分区分解为多数分区）

此时使用 repartition 即可，如果使用 coalesce 需要将 shuffle 设置为 true，否则 coalesce 无效。

我们可以在 filter 操作之后，使用 coalesce 算子针对每个 partition 的数据量各不相同的情况，压缩 partition 的数量，而且让每个 partition 的数据量尽量均匀紧凑，以便于后面的 task 进行计算操作，在某种程度上能够在一定程度上提升性能。

注意：local 模式是进程内模拟集群运行，已经对并行度和分区数量有了一定的内部优化，因此不用去设置并行度和分区数量。

6. 并行度设置

Spark 作业中的并行度指各个 stage 的 task 的数量。

如果并行度设置不合理而导致并行度过低，会导致资源的极大浪费，例如，20 个 Executor，每个 Executor 分配 3 个 CPU core，而 Spark 作业有 40 个 task，这样每个 Executor 分配到的 task 个数是 2 个，这就使得每个 Executor 有一个 CPU core 空闲，导致资源的浪费。

理想的并行度设置，应该是让并行度与资源相匹配，简单来说就是在资源允许的前提下，并行度要设置的尽可能大，达到可以充分利用集群资源。合理的设置并行度，可以提升整个 Spark 作业的性能和运行速度。

Spark 官方推荐，task 数量应该设置为 Spark 作业总 CPU core 数量的 2~3 倍。之所以没有推荐 task 数量与 CPU core 总数相等，是因为 task 的执行时间不同，

有的 task 执行速度快而有的 task 执行速度慢，如果 task 数量与 CPU core 总数相等，那么执行快的 task 执行完成后，会出现 CPU core 空闲的情况。如果 task 数量设置为 CPU core 总数的 2~3 倍，那么一个 task 执行完毕后，CPU core 会立刻执行下一个 task，降低了资源的浪费，同时提升了 Spark 作业运行的效率。Spark 作业并行度的设置如下：

```
val conf = new SparkConf().set("spark.default.parallelism", "500")
```

原则：让 cpu 的 Core（cpu 核心数）充分利用起来，如有 100 个 Core, 那么并行度可以设置为 200~300。

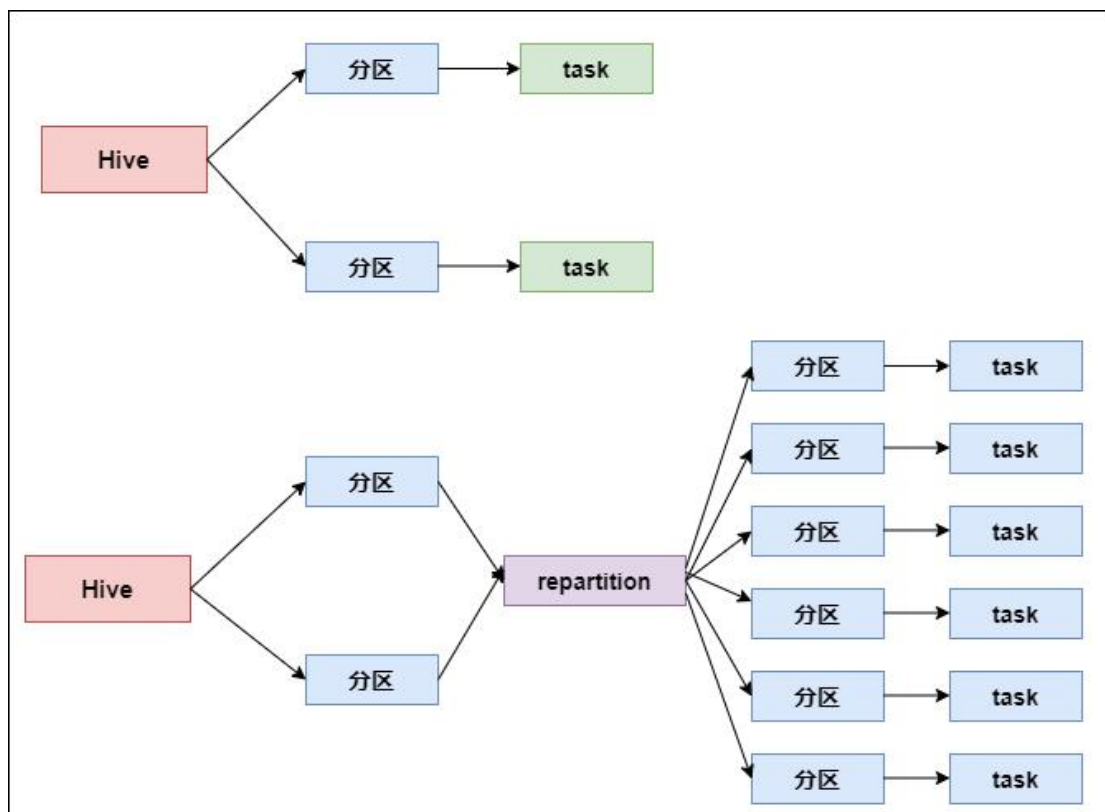
7. repartition/coalesce 调节并行度

我们知道 Spark 中有并行度的调节策略，但是，并行度的设置对于 Spark SQL 是不生效的，用户设置的并行度只对于 Spark SQL 以外的所有 Spark 的 stage 生效。Spark SQL 的并行度不允许用户自己指定，Spark SQL 自己会默认根据 hive 表对应的 HDFS 文件的 split 个数自动设置 Spark SQL 所在的那个 stage 的并行度，用户自己通 `spark.default.parallelism` 参数指定的并行度，只会在没 Spark SQL 的 stage 中生效。

由于 Spark SQL 所在 stage 的并行度无法手动设置，如果数据量较大，并且此 stage 中后续的 transformation 操作有着复杂的业务逻辑，而 Spark SQL 自动设置的 task 数量很少，这就意味着每个 task 要处理为数不少的数据量，然后还要执行非常复杂的处理逻辑，这就可能表现为第一个有 Spark SQL 的 stage 速度很慢，而后续的没有 Spark SQL 的 stage 运行速度非常快。

为了解决 Spark SQL 无法设置并行度和 task 数量的问题，我们可以使用 repartition 算子。

repartition 算子使用前后对比图如下：



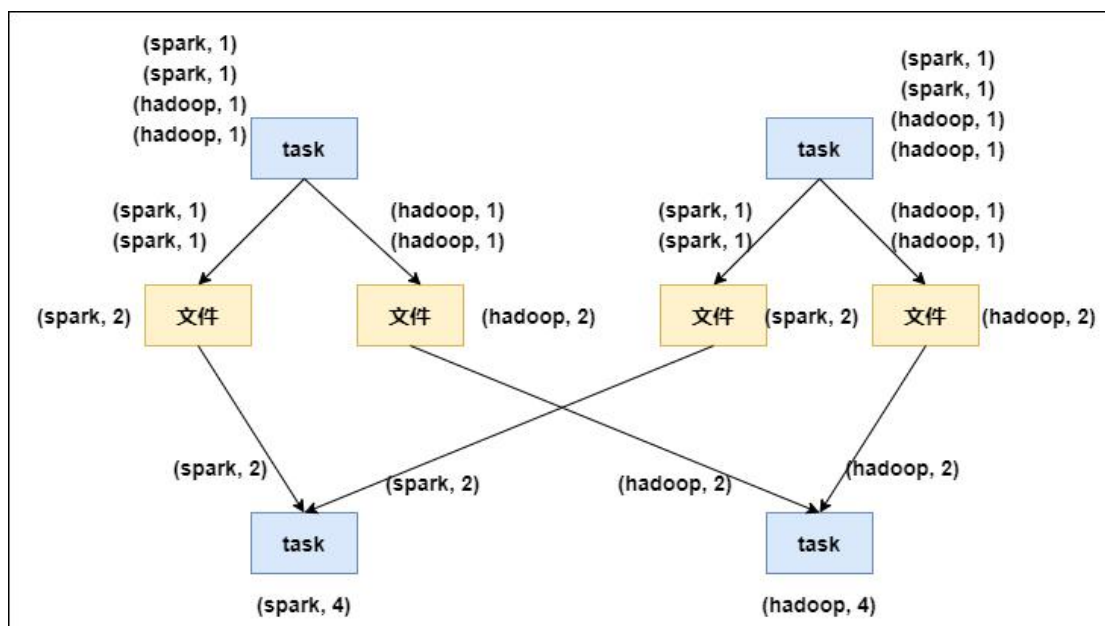
repartition 算子使用前后对比图

Spark SQL 这一步的并行度和 task 数量肯定是没有办法去改变了,但是,对于 Spark SQL 查询出来的 RDD,立即使用 repartition 算子,去重新进行分区,这样可以重新分区为多个 partition,从 repartition 之后的 RDD 操作,由于不再涉及 Spark SQL,因此 stage 的并行度就会等于你手动设置的值,这样就避免了 Spark SQL 所在的 stage 只能用少量的 task 去处理大量数据并执行复杂的算法逻辑。使用 repartition 算子的前后对比如上图所示。

8. reduceByKey 本地预聚合

reduceByKey 相较于普通的 shuffle 操作一个显著的特点就是会进行 map 端的本地聚合, map 端会先对本地的数据进行 combine 操作,然后将数据写入给下个 stage 的每个 task 创建的文件中,也就是在 map 端,对每一个 key 对应的 value,执行 reduceByKey 算子函数。

reduceByKey 算子的执行过程如下图所示:



reduceByKey 算子执行过程

使用 reduceByKey 对性能的提升如下：

1. 本地聚合后，在 map 端的数据量变少，减少了磁盘 IO，也减少了对磁盘空间的占用；
2. 本地聚合后，下一个 stage 拉取的数据量变少，减少了网络传输的数据量；
3. 本地聚合后，在 reduce 端进行数据缓存的内存占用减少；
4. 本地聚合后，在 reduce 端进行聚合的数据量减少。

基于 reduceByKey 的本地聚合特征，我们应该考虑使用 reduceByKey 代替其他的 shuffle 算子，例如 groupByKey。

groupByKey 与 reduceByKey 的运行原理如下图 1 和图 2 所示：

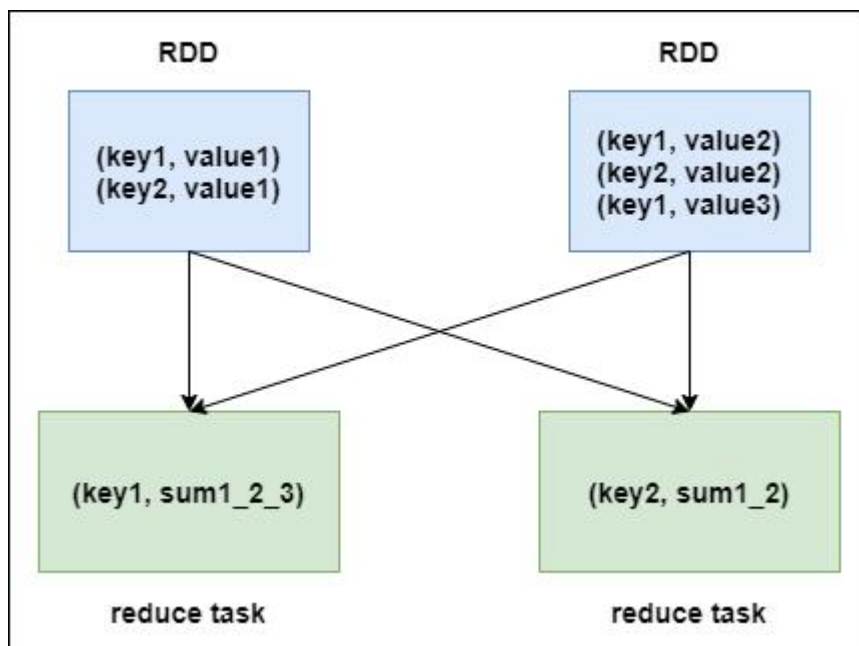


图 1: groupByKey

原理

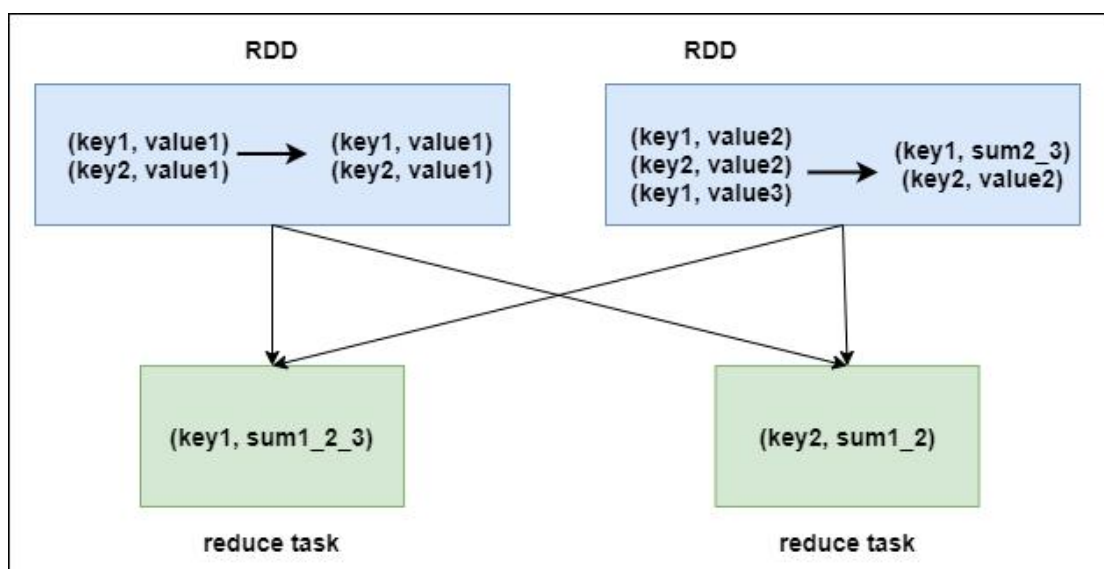


图 2: reduceByKey 原理

根据上图可知，groupByKey 不会进行 map 端的聚合，而是将所有 map 端的数据 shuffle 到 reduce 端，然后在 reduce 端进行数据的聚合操作。由于 reduceByKey 有 map 端聚合的特性，使得网络传输的数据量减小，因此效率要明显高于 groupByKey。

9. 使用持久化+checkpoint

Spark 持久化在大部分情况下是没有问题的，但是有时数据可能会丢失，如果数据一旦丢失，就需要对丢失的数据重新进行计算，计算完后再缓存和使用，为了

避免数据的丢失，可以选择对这个 RDD 进行 checkpoint，也就是**将数据持久化一份到容错的文件系统上（比如 HDFS）**。

一个 RDD 缓存并 checkpoint 后，如果一旦发现缓存丢失，就会优先查看 checkpoint 数据存不存在，如果有，就会使用 checkpoint 数据，而不用重新计算。也即是说，checkpoint 可以视为 cache 的保障机制，如果 cache 失败，就使用 checkpoint 的数据。

使用 checkpoint 的**优点在于提高了 Spark 作业的可靠性，一旦缓存出现问题，不需要重新计算数据，缺点在于，checkpoint 时需要将数据写入 HDFS 等文件系统，对性能消耗较大。**

持久化设置如下：

```
sc.setCheckpointDir('HDFS')
rdd.cache/persist(memory_and_disk)
rdd.checkpoint
```

10. 使用广播变量

默认情况下，task 中的算子中如果使用了外部的变量，每个 task 都会获取一份变量的复本，这就造成了内存的极大消耗。一方面，如果后续对 RDD 进行持久化，可能就无法将 RDD 数据存入内存，只能写入磁盘，磁盘 IO 将会严重消耗性能；另一方面，task 在创建对象的时候，也许会发现堆内存无法存放新创建的对象，这就会导致频繁的 GC，GC 会导致工作线程停止，进而导致 Spark 暂停工作一段时间，严重影响 Spark 性能。

假设当前任务配置了 20 个 Executor，指定 500 个 task，有一个 20M 的变量被所有 task 共用，此时会在 500 个 task 中产生 500 个副本，耗费集群 10G 的内存，如果使用了广播变量，那么每个 Executor 保存一个副本，一共消耗 400M 内存，内存消耗减少了 5 倍。

广播变量在每个 Executor 保存一个副本，此 Executor 的所有 task 共用此广播变量，这让变量产生的副本数量大大减少。

在初始阶段，广播变量只在 Driver 中有一份副本。task 在运行的时候，想要使用广播变量中的数据，此时首先会在自己本地的 Executor 对应的 BlockManager 中尝试获取变量，如果本地没有，BlockManager 就会从 Driver 或者其他节点的 BlockManager 上远程拉取变量的复本，并由本地的 BlockManager 进行管理；之后此 Executor 的所有 task 都会直接从本地的 BlockManager 中获取变量。

对于多个 Task 可能会共用的数据可以广播到每个 Executor 上：


```
val 广播变量名= sc.broadcast(会被各个 Task 用到的变量,即需要广播的变量)
```

```
广播变量名.value//获取广播变量
```

11. 使用 Kryo 序列化

默认情况下，Spark 使用 Java 的序列化机制。Java 的序列化机制使用方便，不需要额外的配置，在算子中使用的变量实现 Serializable 接口即可，但是，Java 序列化机制的效率不高，序列化速度慢并且序列化后的数据所占用的空间依然较大。

Spark 官方宣称 Kryo 序列化机制比 Java 序列化机制性能提高 10 倍左右，Spark 之所以没有默认使用 Kryo 作为序列化类库，是因为它不支持所有对象的序列化，同时 Kryo 需要用户在使用前注册需要序列化的类型，不够方便，但从 Spark 2.0.0 版本开始，简单类型、简单类型数组、字符串类型的 Shuffling RDDs 已经默认使用 Kryo 序列化方式了。

Kryo 序列化注册方式的代码如下：

```
public class MyKryoRegistrar implements KryoRegistrar{
    @Override
    public void registerClasses(Kryo kryo){
        kryo.register(StartupReportLogs.class);
    }
}
```

配置 Kryo 序列化方式的代码如下：

```
//创建 SparkConf 对象
val conf = new SparkConf().setMaster(...).setAppName(...)
//使用 Kryo 序列化库
conf.set("spark.serializer", "org.apache.spark.serializer.KryoSerializer");
//在 Kryo 序列化库中注册自定义的类集合
conf.set("spark.kryo.registrator", "bigdata.com.MyKryoRegistrar");
```

本文档首发于公众号：五分钟学大数据，回复【666】即可获得全套大数据笔面试题

Spark 调优之 Shuffle 调优

1. map 和 reduce 端缓冲区大小

在 Spark 任务运行过程中，如果 shuffle 的 map 端处理的数据量比较大，但是 map 端缓冲的大小是固定的，可能会出现 map 端缓冲数据频繁 spill 溢写到磁盘文件中的情况，使得性能非常低下，通过调节 map 端缓冲的大小，可以避免频繁的磁盘 IO 操作，进而提升 Spark 任务的整体性能。

map 端缓冲的默认配置是 32KB，如果每个 task 处理 640KB 的数据，那么会发生 $640/32 = 20$ 次溢写，如果每个 task 处理 64000KB 的数据，即会发生 $64000/32=2000$ 次溢写，这对于性能的影响是非常严重的。

map 端缓冲的配置方法：

```
val conf = new SparkConf()
    .set("spark.shuffle.file.buffer", "64")
```

Spark Shuffle 过程中，shuffle reduce task 的 buffer 缓冲区大小决定了 reduce task 每次能够缓冲的数据量，也就是每次能够拉取的数据量，如果内存资源较为充足，适当增加拉取数据缓冲区的大小，可以减少拉取数据的次数，也就可以减少网络传输的次数，进而提升性能。

reduce 端数据拉取缓冲区的大小可以通过 `spark.reducer.maxSizeInFlight` 参数进行设置，默认为 48MB。该参数的设置方法如下：

reduce 端数据拉取缓冲区配置：

```
val conf = new SparkConf()
    .set("spark.reducer.maxSizeInFlight", "96")
```

2. reduce 端重试次数和等待时间间隔

Spark Shuffle 过程中，reduce task 拉取属于自己的数据时，如果因为网络异常等原因导致失败会自动进行重试。对于那些包含了特别耗时的 shuffle 操作的作业，建议增加重试最大次数（比如 60 次），以避免由于 JVM 的 full gc 或者网络不稳定等因素导致的数据拉取失败。在实践中发现，对于针对超大数据量（数十亿~上百亿）的 shuffle 过程，调节该参数可以大幅度提升稳定性。

reduce 端拉取数据重试次数可以通过 `spark.shuffle.io.maxRetries` 参数进行设置，该参数就代表了可以重试的最大次数。如果在指定次数之内拉取还是没有成功，就可能会导致作业执行失败，默认为 3，该参数的设置方法如下：

reduce 端拉取数据重试次数配置：

```
val conf = new SparkConf()
    .set("spark.shuffle.io.maxRetries", "6")
```

Spark Shuffle 过程中，reduce task 拉取属于自己的数据时，如果因为网络异常等原因导致失败会自动进行重试，在一次失败后，会等待一定的时间间隔再进行重试，**可以通过加大间隔时长（比如 60s），以增加 shuffle 操作的稳定性。**

reduce 端拉取数据等待间隔可以通过 `spark.shuffle.io.retryWait` 参数进行设置，默认值为 5s，该参数的设置方法如下：

reduce 端拉取数据等待间隔配置：

```
val conf = new SparkConf()
    .set("spark.shuffle.io.retryWait", "60s")
```

3. bypass 机制开启阈值

对于 SortShuffleManager，如果 shuffle reduce task 的数量小于某一阈值则 shuffle write 过程中不会进行排序操作，而是直接按照未经优化的 HashShuffleManager 的方式去写数据，但是最后会将每个 task 产生的所有临时磁盘文件都合并成一个文件，并会创建单独的索引文件。

当你使用 SortShuffleManager 时，如果的确不需要排序操作，那么建议将这个参数调大一些，大于 shuffle read task 的数量，那么此时 map-side 就不会进行排序了，减少了排序的性能开销，但是这种方式下，依然会产生大量的磁盘文件，因此 shuffle write 性能有待提高。

SortShuffleManager 排序操作阈值的设置可以通过

`spark.shuffle.sort.bypassMergeThreshold` 这一参数进行设置，默认值为 200，该参数的设置方法如下：

reduce 端拉取数据等待间隔配置：

```
val conf = new SparkConf()
    .set("spark.shuffle.sort.bypassMergeThreshold", "400")
```

十、故障排除

1. 避免 OOM-out of memory

在 Shuffle 过程，reduce 端 task 并不是等到 map 端 task 将其数据全部写入磁盘后再去拉取，而是 map 端写一点数据，reduce 端 task 就会拉取一小部分数据，然后立即进行后面的聚合、算子函数的使用等操作。

reduce 端 task 能够拉取多少数据,由 reduce 拉取数据的缓冲区 buffer 来决定,因为拉取过来的数据都是先放在 buffer 中,然后再进行后续的处理,buffer 的默认大小为 48MB。

reduce 端 task 会一边拉取一边计算,不一定每次都会拉满 48MB 的数据,可能大多数时候拉取一部分数据就处理掉了。

虽然说增大 reduce 端缓冲区大小可以减少拉取次数,提升 Shuffle 性能,但是有时 map 端的数据量非常大,写出的速度非常快,此时 reduce 端的所有 task 在拉取的时候,有可能全部达到自己缓冲的最大极限值,即 48MB,此时,再加上 reduce 端执行的聚合函数的代码,可能会创建大量的对象,这可能会导致内存溢出,即 OOM。

如果一旦出现 reduce 端内存溢出的问题,我们可以考虑减小 reduce 端拉取数据缓冲区的大小,例如减少为 12MB。

在实际生产环境中是出现过这种问题的,这是典型的以性能换执行的原理。

reduce 端拉取数据的缓冲区减小,不容易导致 OOM,但是相应的,reduce 端的拉取次数增加,造成更多的网络传输开销,造成性能的下降。

注意,要保证任务能够运行,再考虑性能的优化。

2. 避免 GC 导致的 shuffle 文件拉取失败

在 Spark 作业中,有时会出现 shuffle file not found 的错误,这是非常常见的一个报错,有时出现这种错误以后,选择重新执行一遍,就不再报出这种错误。

出现上述问题可能的原因是 Shuffle 操作中,后面 stage 的 task 想要去上一个 stage 的 task 所在的 Executor 拉取数据,结果对方正在执行 GC,执行 GC 会导致 Executor 内所有的工作现场全部停止,比如 BlockManager、基于 netty 的网络通信等,这就会导致后面的 task 拉取数据拉取了半天都没有拉取到,就会报出 shuffle file not found 的错误,而第二次再次执行就不会再出现这种错误。

可以通过调整 reduce 端拉取数据重试次数和 reduce 端拉取数据时间间隔这两个参数来对 Shuffle 性能进行调整,增大参数值,使得 reduce 端拉取数据的重试次数增加,并且每次失败后等待的时间间隔加长。

JVM GC 导致的 shuffle 文件拉取失败调整数据重试次数和 reduce 端拉取数据时间间隔:

```
val conf = new SparkConf()
    .set("spark.shuffle.io.maxRetries", "6")
    .set("spark.shuffle.io.retryWait", "60s")
```

3. YARN-CLIENT 模式导致的网卡流量激增问题

在 YARN-client 模式下，Driver 启动在本地机器上，而 Driver 负责所有的任务调度，需要与 YARN 集群上的多个 Executor 进行频繁的通信。

假设有 100 个 Executor，1000 个 task，那么每个 Executor 分配到 10 个 task，之后，Driver 要频繁地跟 Executor 上运行的 1000 个 task 进行通信，通信数据非常多，并且通信品类特别高。这就导致有可能在 Spark 任务运行过程中，由于频繁大量的网络通讯，本地机器的网卡流量会激增。

注意，YARN-client 模式只会在测试环境中使用，而之所以使用 YARN-client 模式，是由于可以看到详细全面的 log 信息，通过查看 log，可以锁定程序中存在的问题，避免在生产环境下发生故障。

在生产环境下，使用的一定是 YARN-cluster 模式。在 YARN-cluster 模式下，就不会造成本地机器网卡流量激增问题，如果 YARN-cluster 模式下存在网络通信的问题，需要运维团队进行解决。

4. YARN-CLUSTER 模式的 JVM 栈内存溢出无法执行问题

当 Spark 作业中包含 SparkSQL 的内容时，可能会碰到 YARN-client 模式下可以运行，但是 YARN-cluster 模式下无法提交运行（报出 OOM 错误）的情况。

YARN-client 模式下，Driver 是运行在本地机器上的，Spark 使用的 JVM 的 PermGen 的配置，是本地机器上的 spark-class 文件，JVM 永久代的大小是 128MB，这个是没有问题的，但是在 YARN-cluster 模式下，Driver 运行在 YARN 集群的某个节点上，使用的是没有经过配置的默认设置，PermGen 永久代大小为 82MB。

SparkSQL 的内部要进行很复杂的 SQL 的语义解析、语法树转换等等，非常复杂，如果 sql 语句本身就非常复杂，那么很有可能会导致性能的损耗和内存的占用，特别是对 PermGen 的占用会比较大。

所以，此时如果 PermGen 占用好过了 82MB，但是又小于 128MB，就会出现 YARN-client 模式下可以运行，YARN-cluster 模式下无法运行的情况。

解决上述问题的方法是增加 PermGen(永久代)的容量，需要在 spark-submit 脚本中对相关参数进行设置，设置方法如下：

```
--conf spark.driver.extraJavaOptions="-XX:PermSize=128M -XX:MaxPermSize=256M"
```

通过上述方法就设置了 Driver 永久代的大小，默认为 128MB，最大 256MB，这样就可以避免上面所说的问题。

5. 避免 SparkSQL JVM 栈内存溢出

当 SparkSQL 的 sql 语句有成百上千的 or 关键字时，就可能会出现 Driver 端的 JVM 栈内存溢出。

JVM 栈内存溢出基本上就是由于调用的方法层级过多，产生了大量的，非常深的，超出了 JVM 栈深度限制的递归。（我们猜测 SparkSQL 有大量 or 语句的时候，在解析 SQL 时，例如转换为语法树或者进行执行计划的生成的时候，对于 or 的处理是递归，or 非常多时，会发生大量的递归）

此时，建议将一条 sql 语句拆分为多条 sql 语句来执行，每条 sql 语句尽量保证 100 个以内的子句。根据实际的生产环境试验，一条 sql 语句的 or 关键字控制在 100 个以内，通常不会导致 JVM 栈内存溢出。

更多大数据好文，欢迎关注公众号【五分钟学大数据】

十一、Spark 大厂面试真题

1. 通常来说，Spark 与 MapReduce 相比，Spark 运行效率更高。请说明效率更高来源于 Spark 内置的哪些机制？

spark 是借鉴了 Mapreduce, 并在其基础上发展起来的, 继承了其分布式计算的优点并进行了改进, spark 生态更为丰富, 功能更为强大, 性能更加适用范围广, mapreduce 更简单, 稳定性好。主要区别:

1. spark 把运算的中间数据(shuffle 阶段产生的数据)存放在内存, 迭代计算效率更高, mapreduce 的中间结果需要落地, 保存到磁盘;
2. Spark 容错性高, 它通过弹性分布式数据集 RDD 来实现高效容错, RDD 是一组分布式的存储在 节点内存中的只读性的数据集, 这些集合是弹性的, 某一部分丢失或者出错, 可以通过整个数据集的计算流程的血缘关系来实现重建, mapreduce 的容错只能重新计算;
3. Spark 更通用, 提供了 transformation 和 action 这两大类的多功能 api, 另外还有流式处理 sparkstreaming 模块、图计算等等, mapreduce 只提供了 map 和 reduce 两种操作, 流计算及其他模块支持比较缺乏;
4. Spark 框架和生态更为复杂, 有 RDD, 血缘 lineage、执行时的有向无环图 DAG, stage 划分等, 很多时候 spark 作业都需要根据不同业务场景的需求

要进行调优以达到性能要求，mapreduce 框架及其生态相对较为简单，对性能的要求也相对较弱，运行较为稳定，适合长期后台运行；

5. Spark 计算框架对内存的利用和运行的并行度比 mapreduce 高，Spark 运行容器为 executor，内部 ThreadPool 中线程运行一个 Task，mapreduce 在线程内部运行 container，container 容器分类为 MapTask 和 ReduceTask，Spark 程序运行并行度更高；
6. Spark 对于 executor 的优化，在 JVM 虚拟机的基础上对内存弹性利用：storage memory 与 Execution memory 的弹性扩容，使得内存利用效率更高。

2. hadoop 和 spark 使用场景？

Hadoop/MapReduce 和 Spark 最适合的都是做离线型的数据分析，但 Hadoop 特别适合是单次分析的数据量“很大”的情景，而 Spark 则适用于数据量不是很大的情景。

1. 一般情况下，对于中小互联网和企业级的大数据应用而言，单次分析的数量都不会“很大”，因此可以优先考虑使用 Spark。
2. 业务通常认为 Spark 更适用于机器学习之类的“迭代式”应用，80GB 的压缩数据（解压后超过 200GB），10 个节点的集群规模，跑类似“sum+group-by”的应用，MapReduce 花了 5 分钟，而 spark 只需要 2 分钟。

3. spark 如何保证宕机迅速恢复？

1. 适当增加 spark standby master
2. 编写 shell 脚本，定期检测 master 状态，出现宕机后对 master 进行重启操作

4. hadoop 和 spark 的相同点和不同点？

Hadoop 底层使用 MapReduce 计算架构，只有 map 和 reduce 两种操作，表达能力比较欠缺，而且在 MR 过程中会重复的读写 hdfs，造成大量的磁盘 io 读写操作，所以适合高时延环境下批处理计算的应用；

Spark 是基于内存的分布式计算架构，提供更加丰富的数据集操作类型，主要分成转化操作和行动操作，包括 map、reduce、filter、flatMap、groupByKey、reduceByKey、union 和 join 等，数据分析更加快速，所以适合低时延环境下计算的应用；

spark 与 hadoop 最大的区别在于迭代式计算模型。基于 mapreduce 框架的 Hadoop 主要分为 map 和 reduce 两个阶段，两个阶段完了就结束了，所以在一个 job 里面能做的处理很有限；spark 计算模型是基于内存的迭代式计算模型，可以分为 n 个阶段，根据用户编写的 RDD 算子和程序，在处理完一个阶段后可以继续往下处理很多个阶段，而不只是两个阶段。所以 spark 相较于 mapreduce，计算模型更加灵活，可以提供更强大的功能。

但是 spark 也有劣势，由于 spark 基于内存进行计算，虽然开发容易，但是真正面对大数据的时候，在没有进行调优的情况下，可能会出现各种各样的问题，比如 OOM 内存溢出等情况，导致 spark 程序可能无法运行起来，而 mapreduce 虽然运行缓慢，但是至少可以慢慢运行完。

5. RDD 持久化原理？

spark 非常重要的一个功能特性就是可以将 RDD 持久化在内存中。

调用 `cache()` 和 `persist()` 方法即可。`cache()` 和 `persist()` 的区别在于，`cache()` 是 `persist()` 的一种简化方式，`cache()` 的底层就是调用 `persist()` 的无参版本 `persist(MEMORY_ONLY)`，将数据持久化到内存中。

如果需要从内存中清除缓存，可以使用 `unpersist()` 方法。RDD 持久化是可以手动选择不同的策略的。在调用 `persist()` 时传入对应的 `StorageLevel` 即可。

6. checkpoint 检查点机制？

应用场景：当 spark 应用程序特别复杂，从初始的 RDD 开始到最后整个应用程序完成有很多的步骤，而且整个应用运行时间特别长，这种情况下就比较适合使用 checkpoint 功能。

原因：对于特别复杂的 Spark 应用，会出现某个反复使用的 RDD，即使之前持久化过但由于节点的故障导致数据丢失了，没有容错机制，所以需要重新计算一次数据。

Checkpoint 首先会调用 SparkContext 的 `setCheckpointDir()` 方法，设置一个容错的文件系统的目录，比如说 HDFS；然后对 RDD 调用 `checkpoint()` 方法。之后在 RDD 所处的 job 运行结束之后，会启动一个单独的 job，来将 checkpoint 过的 RDD 数据写入之前设置的文件系统，进行高可用、容错的类持久化操作。

检查点机制是我们在 spark streaming 中用来保障容错性的主要机制，它可以使 spark streaming 阶段性的把应用数据存储到诸如 HDFS 等可靠存储系统中，以供恢复时使用。具体来说基于以下两个目的服务：

1. 控制发生失败时需要重算的状态数。Spark streaming 可以通过转化图的谱系图来重算状态，检查点机制则可以控制需要在转化图中回溯多远。
2. 提供驱动器程序容错。如果流计算应用中的驱动器程序崩溃了，你可以重启驱动器程序并让驱动器程序从检查点恢复，这样 spark streaming 就可以读取之前运行的程序处理数据的进度，并从那里继续。

7. checkpoint 和持久化机制的区别？

最主要的区别在于持久化只是将数据保存在 BlockManager 中，但是 RDD 的 lineage(血缘关系，依赖关系)是不变的。但是 checkpoint 执行完之后，rdd 已经没有之前所谓的依赖 rdd 了，而只有一个强行为其设置的 checkpointRDD，checkpoint 之后 rdd 的 lineage 就改变了。

持久化的数据丢失的可能性更大，因为节点的故障会导致磁盘、内存的数据丢失。但是 checkpoint 的数据通常是保存在高可用的文件系统中，比如 HDFS 中，所以数据丢失可能性比较低

8. RDD 机制理解吗？

rdd 分布式弹性数据集，简单的理解成一种数据结构，是 spark 框架上的通用货币。所有算子都是基于 rdd 来执行的，不同的场景会有不同的 rdd 实现类，但是都可以进行互相转换。rdd 执行过程中会形成 dag 图，然后形成 lineage 保证容错性等。从物理的角度来看 rdd 存储的是 block 和 node 之间的映射。

RDD 是 spark 提供的核心抽象，全称为弹性分布式数据集。

RDD 在逻辑上是一个 hdfs 文件，在抽象上是一种元素集合，包含了数据。它是由分区组成的，分为多个分区，每个分区分布在集群中的不同节点上，从而让 RDD 中的数据可以被并行操作（分布式数据集）

比如有个 RDD 有 90W 数据，3 个 partition，则每个分区上有 30W 数据。RDD 通常通过 Hadoop 上的文件，即 HDFS 或者 HIVE 表来创建，还可以通过应用程序中的集合来创建；RDD 最重要的特性就是容错性，可以自动从节点失败中恢复过来。即如果某个节点上的 RDD partition 因为节点故障，导致数据丢失，那么 RDD 可以通过自己的数据来源重新计算该 partition。这一切对使用者都是透明的。RDD 的数据默认存放在内存中，但是当内存资源不足时，spark 会自动将 RDD 数据写入磁盘。比如某节点内存只能处理 20W 数据，那么这 20W 数据就会放入内存中计算，剩下 10W 放到磁盘中。RDD 的弹性体现在于 RDD 上自动进行内存和磁盘之间权衡和切换的机制。

9. Spark streaming 以及基本工作原理？

Spark streaming 是 spark core API 的一种扩展，可以用于进行大规模、高吞吐量、容错的实时数据流的处理。

它支持从多种数据源读取数据，比如 Kafka、Flume、Twitter 和 TCP Socket，并且能够使用算子比如 map、reduce、join 和 window 等来处理数据，处理后的数据可以保存到文件系统、数据库等存储中。

Spark streaming 内部的基本工作原理是：接受实时输入数据流，然后将数据拆分成 batch，比如每收集一秒的数据封装成一个 batch，然后将每个 batch 交给 spark 的计算引擎进行处理，最后会生产出一个结果数据流，其中的数据也是一个一个的 batch 组成的。

10. DStream 以及基本工作原理？

DStream 是 spark streaming 提供了一种高级抽象，代表了一个持续不断的数据流。

DStream 可以通过输入数据源来创建，比如 Kafka、flume 等，也可以通过其他 DStream 的高阶函数来创建，比如 map、reduce、join 和 window 等。

DStream 内部其实不断产生 RDD，每个 RDD 包含了一个时间段的数据。
Spark streaming 一定是有一个输入的 DStream 接收数据，按照时间划分成一个一个的 batch，并转化为一个 RDD，RDD 的数据是分散在各个子节点的 partition 中。

11. spark 有哪些组件？

1. master：管理集群和节点，不参与计算。
2. worker：计算节点，进程本身不参与计算，和 master 汇报。
3. Driver：运行程序的 main 方法，创建 spark context 对象。
4. spark context：控制整个 application 的生命周期，包括 dagscheduler 和 task scheduler 等组件。
5. client：用户提交程序的入口。

12. spark 工作机制？

用户在 client 端提交作业后，会由 Driver 运行 main 方法并创建 spark context 上下文。执行 add 算子，形成 dag 图输入 dagscheduler，按照 add 之间的依赖关系划分 stage 输入 task scheduler。task scheduler 会将 stage 划分为 task set 分发到各个节点的 executor 中执行。

13. 说下宽依赖和窄依赖

- 宽依赖：
本质就是 shuffle。父 RDD 的每一个 partition 中的数据，都可能会传输一部分到下一个子 RDD 的每一个 partition 中，此时会出现父 RDD 和子 RDD 的 partition 之间具有交互错综复杂的关系，这种情况就叫做两个 RDD 之间是宽依赖。
- 窄依赖：
父 RDD 和子 RDD 的 partition 之间的对应关系是一对一的。

14. Spark 主备切换机制原理知道吗？

Master 实际上可以配置两个，Spark 原生的 standalone 模式是支持 Master 主备切换的。当 Active Master 节点挂掉以后，我们可以将 Standby Master 切换为 Active Master。

Spark Master 主备切换可以基于两种机制，一种是基于文件系统的，一种是基于 ZooKeeper 的。

基于文件系统的主备切换机制，需要在 Active Master 挂掉之后手动切换到 Standby Master 上；

而基于 Zookeeper 的主备切换机制，可以实现自动切换 Master。

15. spark 解决了 hadoop 的哪些问题？

1. **MR**: 抽象层次低，需要使用手工代码来完成程序编写，使用上难以上手；

Spark: Spark 采用 RDD 计算模型，简单容易上手。

2. **MR**: 只提供 map 和 reduce 两个操作，表达能力欠缺；

Spark: Spark 采用更加丰富的算子模型，包括 map、flatMap、groupByKey、reduceByKey 等；

3. **MR**: 一个 job 只能包含 map 和 reduce 两个阶段，复杂的任务需要包含很多个 job，这些 job 之间的管理以来需要开发者自己进行管理；

Spark: Spark 中一个 job 可以包含多个转换操作，在调度时可以生成多个 stage，而且如果多个 map 操作的分区不变，是可以放在同一个 task 里面去执行；

4. **MR**: 中间结果存放在 hdfs 中；

Spark: Spark 的中间结果一般存在内存中，只有当内存不够了，才会存入本地磁盘，而不是 hdfs；

5. **MR**: 只有等到所有的 map task 执行完毕后才能执行 reduce task；

Spark: Spark 中分区相同的转换构成流水线在一个 task 中执行，分区不同的需要进行 shuffle 操作，被划分成不同的 stage 需要等待前面的 stage 执行完才能执行。

6. **MR**: 只适合 batch 批处理，时延高，对于交互式处理和实时处理支持不够；

Spark: Spark streaming 可以将流拆成时间间隔的 batch 进行处理，实时计算。

16. 数据倾斜的产生和解决办法？

数据倾斜以为着某一个或者某几个 partition 的数据特别大，导致这几个 partition 上的计算需要耗费相当长的时间。

在 spark 中同一个应用程序划分成多个 stage，这些 stage 之间是串行执行的，而一个 stage 里面的多个 task 是可以并行执行，task 数目由 partition 数目决定，如果一个 partition 的数目特别大，那么导致这个 task 执行时间很长，导致接下来的 stage 无法执行，从而导致整个 job 执行变慢。

避免数据倾斜，一般是要选用合适的 key，或者自己定义相关的 partitioner，通过加盐或者哈希值来拆分这些 key，从而将这些数据分散到不同的 partition 去执行。

如下算子会导致 shuffle 操作，是导致数据倾斜可能发生的关键点所在：

groupByKey; reduceByKey; aggregateByKey; join; cogroup;

17. 你用 sparksql 处理的时候，处理过程中用的 dataframe 还是直接写的 sql？为什么？

这个问题的宗旨是问你 spark sql 中 dataframe 和 sql 的区别，从执行原理、操作方便程度和自定义程度来分析 这个问题。

18. RDD 中 reduceByKey 与 groupByKey 哪个性能好，为什么

reduceByKey: reduceByKey 会在结果发送至 reducer 之前会对每个 mapper 在本地进行 merge，有点类似于在 MapReduce 中的 combiner。这样做的好处在于，在 map 端进行一次 reduce 之后，数据量会大幅度减小，从而减小传输，保证 reduce 端能够更快的进行结果计算。

groupByKey: groupByKey 会对每一个 RDD 中的 value 值进行聚合形成一个序列 (Iterator)，此操作发生在 reduce 端，所以势必会将所有的数据通过网络进行传输，造成不必要的浪费。同时如果数据量十分大，可能还会造成 OutOfMemoryError。

所以在进行大量数据的 reduce 操作时候建议使用 reduceByKey。不仅可以提高速度，还可以防止使用 groupByKey 造成的内存溢出问题。

19. Spark master HA 主从切换过程不会影响到集群已有作业的运行，为什么

不会的。

因为程序在运行之前，已经申请过资源了，driver 和 Executors 通讯，不需要和 master 进行通讯的。

20. spark master 使用 zookeeper 进行 ha，有哪些源数据保存到 Zookeeper 里面

spark 通过这个参数 `spark.deploy.zookeeper.dir` 指定 master 元数据在 zookeeper 中保存的位置，包括 Worker, Driver 和 Application 以及 Executors。standby 节点要从 zk 中，获得元数据信息，恢复集群运行状态，才能对外继续提供服务，作业提交资源申请等，在恢复前是不能接受请求的。

注：Master 切换需要注意 2 点：

- 1、在 Master 切换的过程中，所有的已经在运行的程序皆正常运行！因为 Spark Application 在运行前就已经通过 Cluster Manager 获得了计算资源，所以在运行时 Job 本身的调度和处理和 Master 是没有任何关系。
- 2、在 Master 的切换过程中唯一的影响是不能提交新的 Job：一方面不能够提交新的应用程序给集群，因为只有 Active Master 才能接受新的程序的提交请求；另外一方面，已经运行的程序中也不能够因 Action 操作触发新的 Job 的提交请求。

最后

第一时间获取最新大数据技术，尽在公众号：[五分钟学大数据](#)

搜索公众号：[五分钟学大数据](#)，学更多大数据技术！

可直接扫码关注



微信搜一搜



五分钟学大数据