初识RDD

简介

RDD全称是**Resilient Distributed Dataset**，弹性分布式数据集。

RDD是一个容错的、并行的、不可变的数据结构，可以显式地将数据存储到磁盘和内存中。

初识RDD

特性

分区**partition**的集合

提供处理每个分片的算子操作

RDD之间有着一系列的相互依赖

对于**<k,v>**对形式的**RDD**可指定一个分区器**partitioner**，告诉它如何分区

数据本地性

初识RDD

创建方式

集合并行化（**Parallelized Collections**）

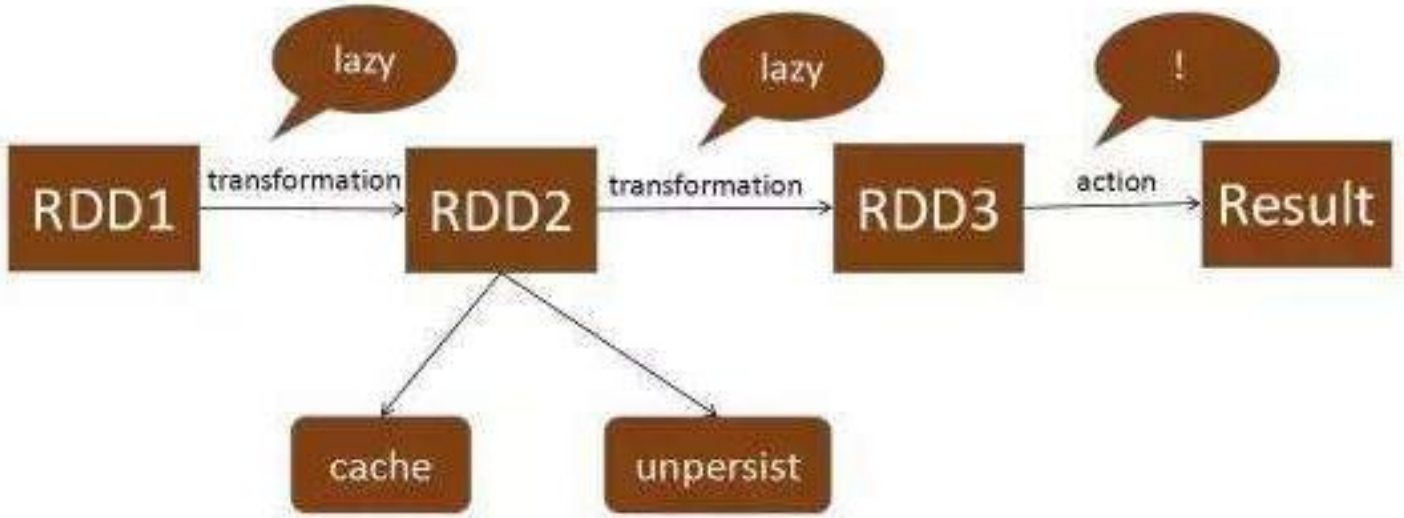
```
val data = Array(1, 2, 3, 4, 5)
val distData = sc.parallelize(data)
```

外部数据源（**External Datasets**）

```
val distFile = sc.textFile(“data.txt”)
```

RDD操作

算子


lazy
lazy
!
RDD1
transformation
RDD2
transformation
RDD3
action
Result
cache
unpersist

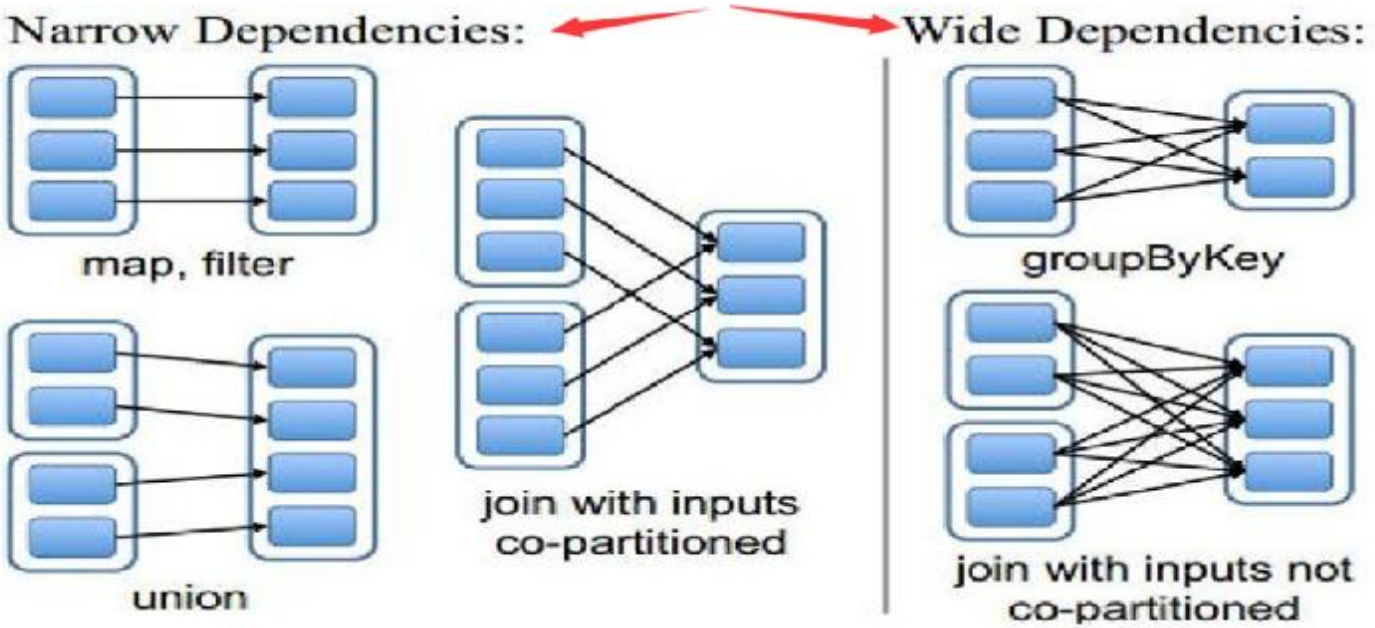
RDD操作

算子-transformation

操作	介绍
map	将RDD中的每个元素传入自定义函数，获取一个新的元素，然后用新的元素组成新的RDD
filter	对RDD中每个元素进行判断，如果返回true则保留，返回false则剔除。
flatMap	与map类似，但是对每个元素都可以返回一个或多个新元素。
gropuByKey	根据key进行分组，每个key对应一个Iterable<value>
reduceByKey	对每个key对应的value进行reduce操作。
sortByKey	对每个key对应的value进行排序操作。
Join	对两个包含<key,value>对的RDD进行join操作，每个key join上的pair，都会传入自定义函数进行处理。
cogroup	同join，但是每个key对应的Iterable<value>都会传入自定义函数进行处理。

RDD操作

算子-transformation依赖图解



RDD操作

窄依赖（**narrow dependencies**）

子**RDD**的每个分区依赖于常数个父分区
输入输出一对一，结果**RDD**的分区结构不变，主要是**map**、**flatMap**
输入输出一对一，但结果**RDD**的分区结构发生变化，如**union**、**coalesce**
从输入中选择部分元素的算子，如**filter**、**distinct**、**subtract**、**sample**

宽依赖（**wide dependencies**）

子**RDD**的每个分区依赖于所有父**RDD**分区
对单个**RDD**基于**key**进行重组和**reduce**，如**groupByKey**、**reduceByKey**
对两个**RDD**基于**key**进行**join**和重组，如**join**

RDD操作

算子-action

操作	介绍
reduce	将RDD中的所有元素进行聚合操作。第一个和第二个元素聚合，值与第三个元素聚合，值与第四个元素聚合，以此类推。
collect	将RDD中所有元素获取到本地客户端。
count	获取RDD元素总数。
take(n)	获取RDD中前n个元素。
saveAsTextFile	将RDD元素保存到文件中，对每个元素调用toString方法
countByKey	对每个key对应的值进行count计数。
foreach	遍历RDD中的每个元素。

RDD操作

算子-缓存概述

Spark速度非常快的原因之一，就是在不同操作中可以在内存中持久化或者缓存数据集。

当持久化某个**RDD**后，每一个节点都将把计算分区结果保存在内存中，在此**RDD**或衍生出的**RDD**进行的其他动作中重用。这使得后续的动作变得更加迅速。

缓存是**Spark**构建迭代式算法和快速交互式查询的关键。

RDD操作

算子-缓存方式

RDD通过**persist**方法或**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。

cache最终也是调用了**persist**方法，默认的存储级别是仅在内存存储一份。

```
/** Persist this RDD with the default storage level (`MEMORY_ONLY`). */
def persist(): this.type = persist(StorageLevel.MEMORY_ONLY)

/** Persist this RDD with the default storage level (`MEMORY_ONLY`). */
def cache(): this.type = persist()
```

RDD操作

算子-缓存方式

Spark的存储级别还有好多种，存储级别在**object StorageLevel**中定义的。

```
object StorageLevel {
  val NONE = new StorageLevel(false, false, false, false)
  val DISK_ONLY = new StorageLevel(true, false, false, false)
  val DISK_ONLY_2 = new StorageLevel(true, false, false, false, 2)
  val MEMORY_ONLY = new StorageLevel(false, true, false, true)
  val MEMORY_ONLY_2 = new StorageLevel(false, true, false, true, 2)
  val MEMORY_ONLY_SER = new StorageLevel(false, true, false, false)
  val MEMORY_ONLY_SER_2 = new StorageLevel(false, true, false, false, 2)
  val MEMORY_AND_DISK = new StorageLevel(true, true, false, true)
  val MEMORY_AND_DISK_2 = new StorageLevel(true, true, false, true, 2)
  val MEMORY_AND_DISK_SER = new StorageLevel(true, true, false, false)
  val MEMORY_AND_DISK_SER_2 = new StorageLevel(true, true, false, false, 2)
  val OFF_HEAP = new StorageLevel(false, false, true, false)
```

缓存有可能丢失，**RDD**的缓存容错机制保证即使缓存丢失也能保证计算正确执行。通过基于**RDD**的一系列转换，丢失的数据会被重算，由于**RDD**的各个**Partition**是相对独立的，因此只需要计算丢失的部分即可，并不需要重算全部**Partition**。

容错机制

容错概述

容错 指的是一个系统在部分模块出现故障时还能否持续的对外提供服务。 **Spark** 使用容错机制来提高整个系统的可用性。

容错机制：使用记录数据的更新（血统**Lineage**）和数据检查点（**Checkpoint**）

容错机制

Lineage机制

RDD的**Lineage**记录的是粗粒度的特定数据**Transformation**操作行为。当**RDD**的部分分区数据丢失时，可以通过**Lineage**来重新运算和恢复丢失的数据分区。这种粗颗粒的数据模型，限制了**Spark**的运用场合，所以**Spark**并不适用于所有高性能要求的场景，但同时相比细颗粒度的数据模型，也带来了性能的提升。

Spark Lineage机制是通过**RDD**的依赖关系来执行的
1.窄依赖可以在某个计算节点上直接通过计算父**RDD**的某块数据计算得到子**RDD**对应的某块数据。
2.宽依赖则要等到父**RDD**所有数据都计算完成后，将父**RDD**的计算结果进行**hash**并传到对应节点上之后才能计算子**RDD**。宽依赖要将祖先**RDD**中的所有数据块全部重新计算，所以在长“血统”链特别是有宽依赖的时候，需要在适当的时机设置数据检查点。

容错机制

Checkpoint机制

两种实现方式
1.LocalRDDCheckpointData：临时存储在本地**executor**的磁盘和内存上。该实现的特点是比较快，适合**lineage**信息需要经常被删除的场景（如**GraphX**），可容忍**executor**挂掉。

2.ReliableRDDCheckpointData：存储在外部可靠存储（如**hdfs**），可以达到容忍**driver** 挂掉情况。虽然效率没有存储本地高，但是容错级别最好。

注：如果代码中没有设置**checkpoint**，则使用**local**的**checkpoint**模式，如果设置路径，则使用**reliable**的**checkpoint**模式。

当**RDD**的**action**算子触发计算结束后会执行**checkpoint**；**Task**计算失败的时候会从**checkpoint**读取数据进行计算。

共享变量

共享变量

Spark是集群部署的，具有很多节点，节点之间的运算是相互独立的，**Spark**会自动把函数中所有引用到的变量发送到每个工作节点上。虽然很方便，但有时也很低效，比如你可能会在多个并行操作中使用同一个变量，而**Spark**每次都要把它分别发送给每个节点。所以共享变量的存在是很有必要的。

Spark提供了两种有限类型的共享变量，广播变量和累加器。

共享变量

广播变量（**broadcast**）

假如多个并行操作会用到同一个变量，而**Spark每次都将这个变量自动分发到每个节点**，如果变量很大，那么会很低效。可以引入一个广播变量，让程序高效的给所有工作节点发送一个较大的可读值，而不是每个任务经过网络传输保存一份拷贝。这样该变量不会多次发送到各节点，提高了效率。

使用方法：使用**sparkContext**的**broadcast()**创建广播变量。使用**value**属性访问广播值。使用**unpersist()**清除广播变量。

共享变量

累加器（**accumulator**）

大数据操作几乎都是并行、分节点、分区的。但集群中每个节点的运算是独立的，每个运行的任务都会得到该变量的一份新的副本，更新这些副本的值不会影响驱动器中的对应变量。所以需要一个共享变量：累加器。累加器的作用就是多个节点之间共享一个变量。它将工作节点的值聚合到**Driver**端。

使用方法：在**Driver**中调用 **SparkContext**中的**Accumulato**相关方法创建累加器，并给它定义**name**，方便在**Web UI**中查看。

常用实例

排序

去重

平均值

最大最小值

每年最高温度

TopN

分组**TopN**

二次排序

常用实例

app用户行为分析

PV/UV统计

session粒度进行数据聚合

统计**Session**时长步长

各个范围的**session**占比

top10热门品类

top10活跃**session**

Shuffle

Shuffle概述

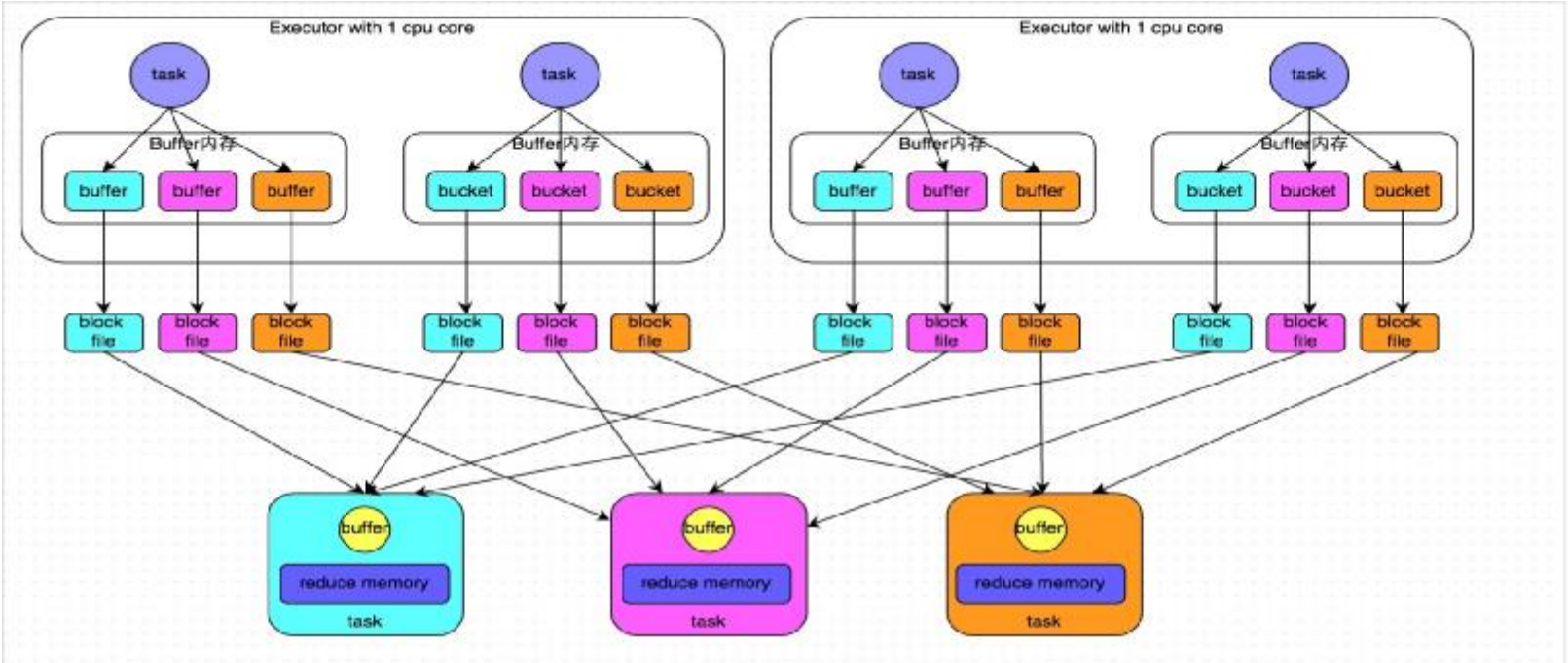
在**Spark**中，负责**shuffle**过程的执行、计算和处理的组件主要就是**ShuffleManager**，即**shuffle**管理器。而随着**Spark**的版本的发展，**ShuffleManager**也在不断迭代，变得越来越先进。

在**Spark 1.2**以前，默认的**shuffle**计算引擎是**HashShuffleManager**。而**HashShuffleManager**有着一个非常严重的弊端，就是会产生大量的中间磁盘文件，进而由大量的磁盘**IO**操作影响了性能。

在**Spark 1.2**以后的版本中，默认的**ShuffleManager**改成了**SortShuffleManager**。**SortShuffleManager**相较于**HashShuffleManager**来说，有了一定的改进。主要就在于，每个**Task**在进行**shuffle**操作时，虽然也会产生较多的临时磁盘文件，但是最后会将所有的临时文件合并（**merge**）成一个磁盘文件，因此每个**Task**就只有一个磁盘文件。在下一个**stage**的**shuffle read task**拉取自己的数据时，只要根据索引读取每个磁盘文件中的部分数据即可。

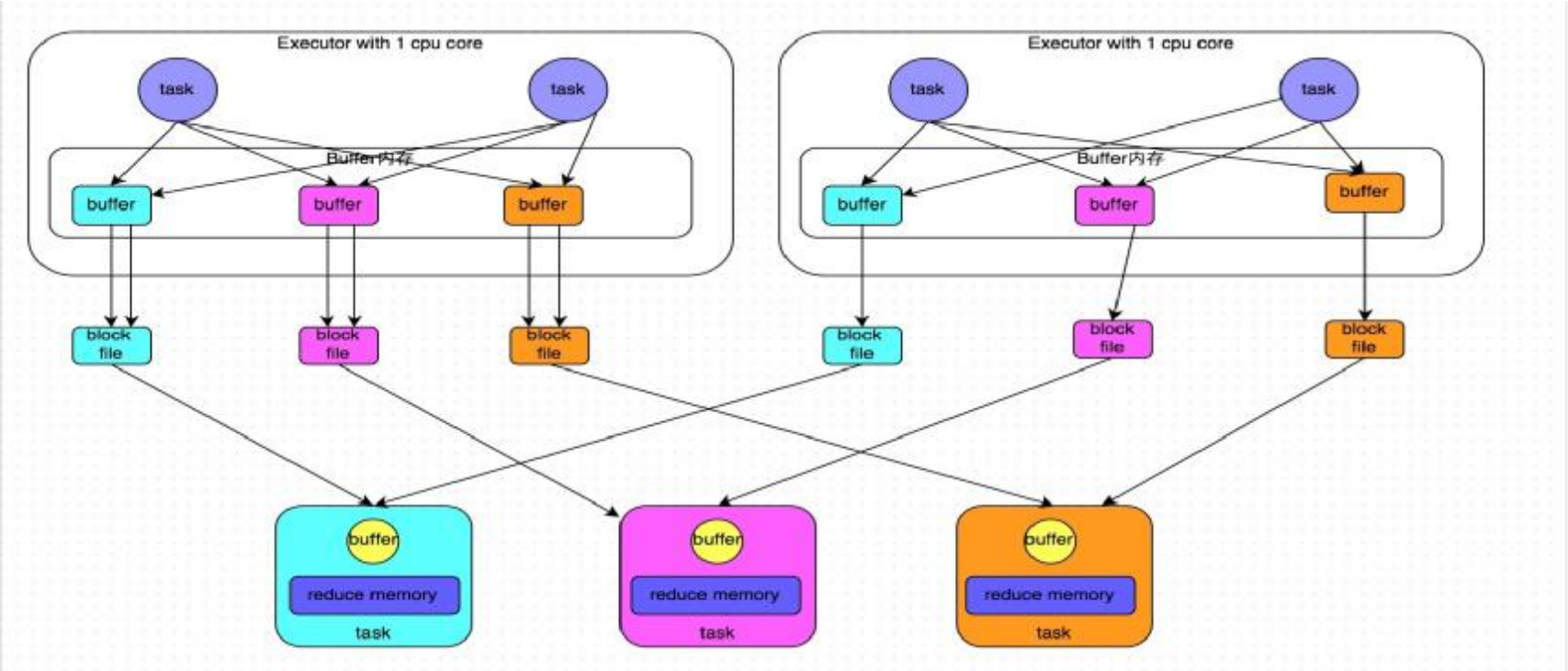
Shuffle

未经优化的**HashShuffleManager**



Shuffle

优化后的**HashShuffleManager**



Shuffle

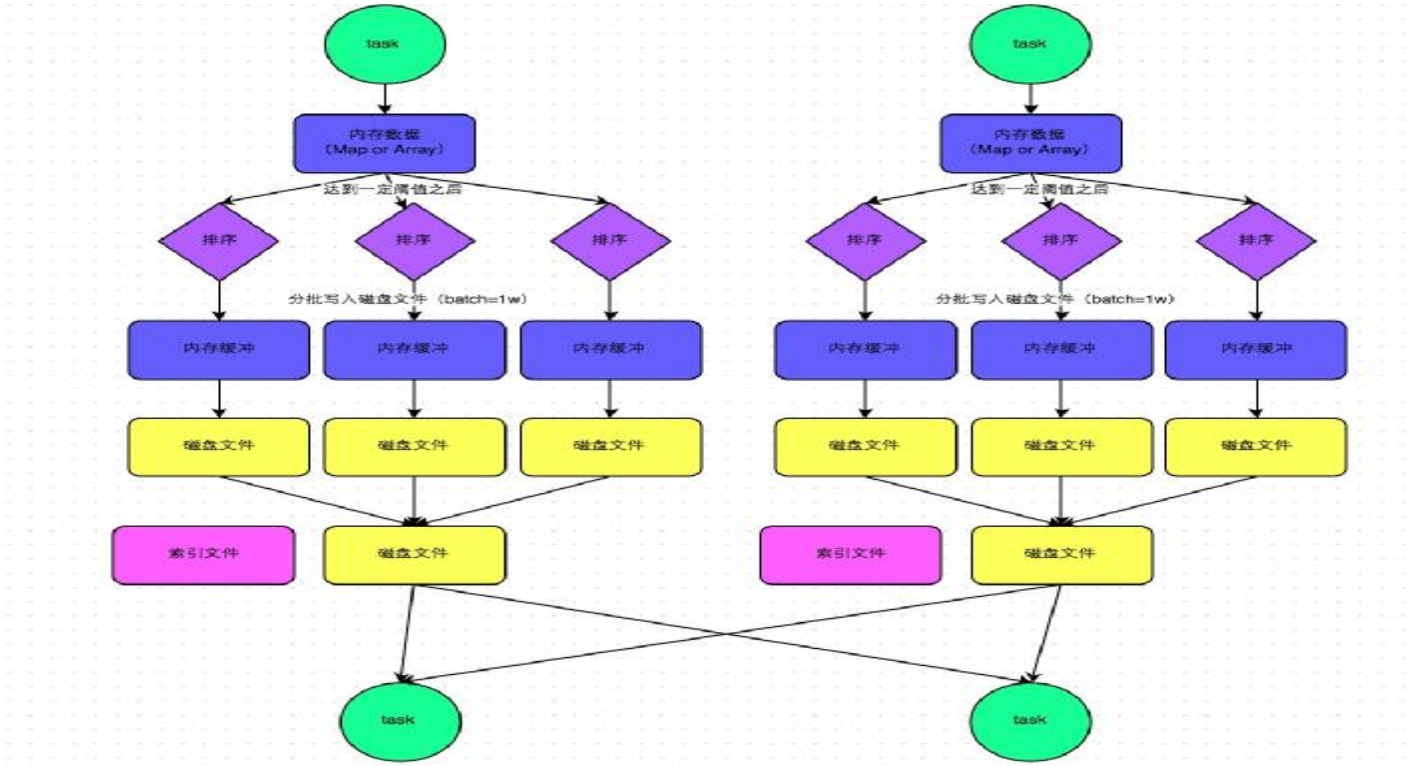
优化后的**HashShuffleManager**

SortShuffleManager的运行机制主要分成两种，一种是普通运行机制，另一种是**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。

Shuffle

普通运行机制



Shuffle

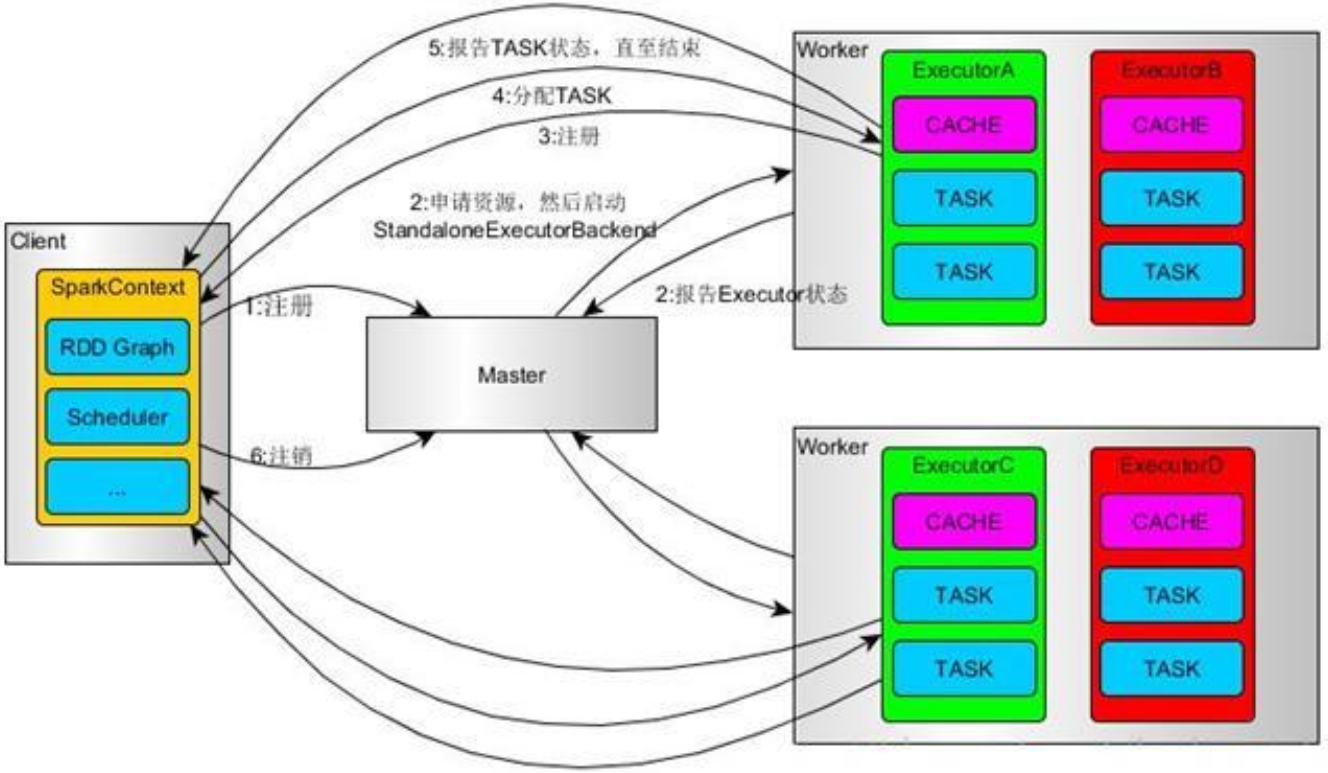
bypass运行机制

shuffle map task数量小于**spark.shuffle.sort.bypassMergeThreshold**参数的值

不是聚合类的**shuffle**算子（比如**reduceByKey**）

执行流程

Spark on Standalone运行流程



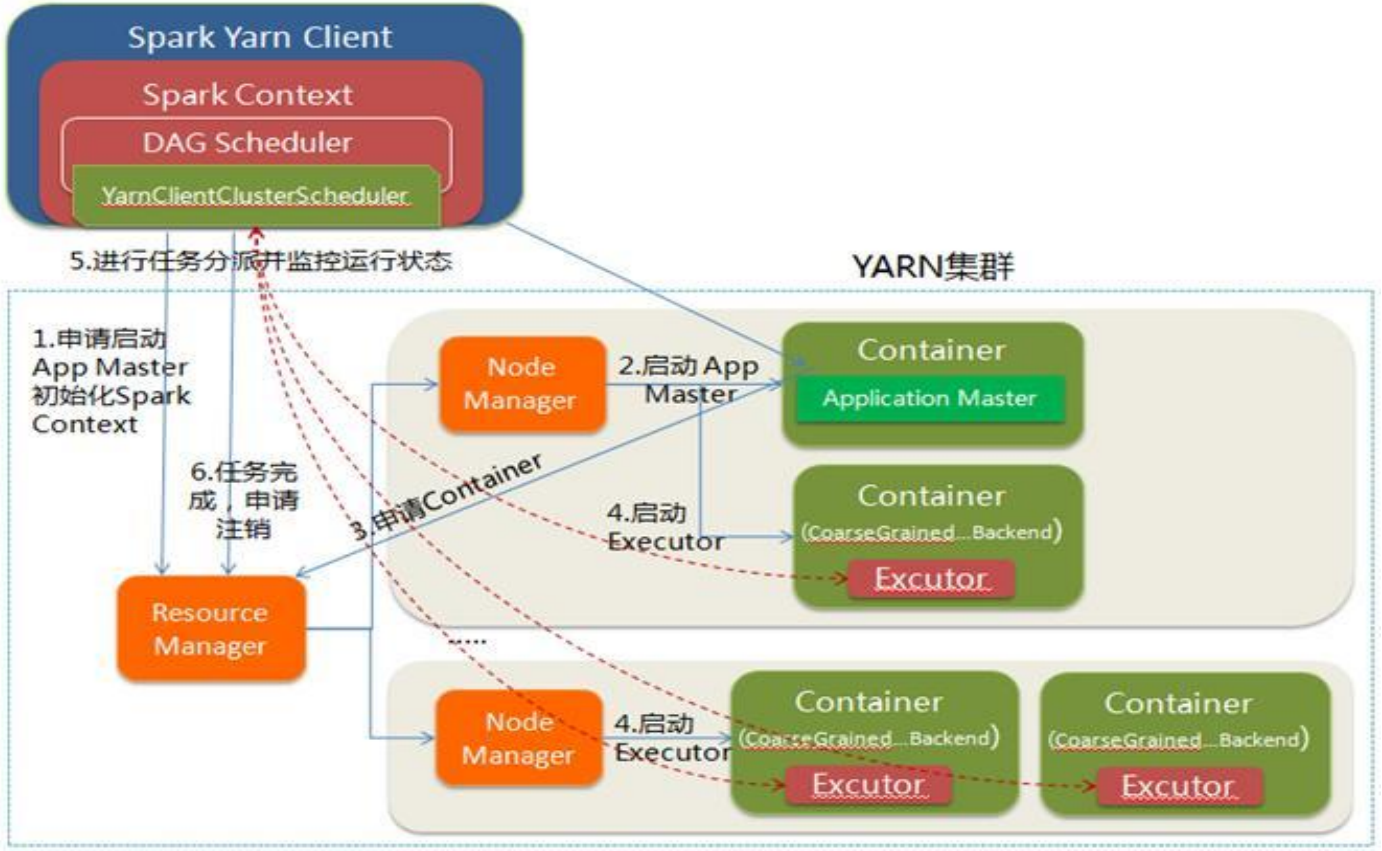
执行流程

Spark on Standalone运行流程

1.SparkContext连接到**Master**，向**Master**注册并申请资源（**CPU Core** 和**Memory**）
2.Master根据**SparkContext**的资源申请要求和**Worker**心跳周期内报告的信息决定在哪个**Worker**上分配资源，然后在该**Worker**上获取资源，然后启动**StandaloneExecutorBackend**
3.StandaloneExecutorBackend向**SparkContext**注册
4. SparkContext将**Applicaiton**代码发送给**StandaloneExecutorBackend**；并且**SparkContext**解 析**Applicaiton**代码，构建**DAG**图，并提交给**DAG Scheduler**分解成**Stage**。然后将**Stage**提交给**Task Scheduler**，**Task Scheduler**负责将**Task**分配到相应的**Worker**，最后提交给**StandaloneExecutorBackend**执行
5.StandaloneExecutorBackend会建立**Executor**线程池，开始执行**Task**，并向**SparkContext**报告，直至**Task**完成
6.所有**Task**完成后，**SparkContext**向**Master**注销，释放资源

执行流程

Spark on YARN（YARN-Client）运行流程



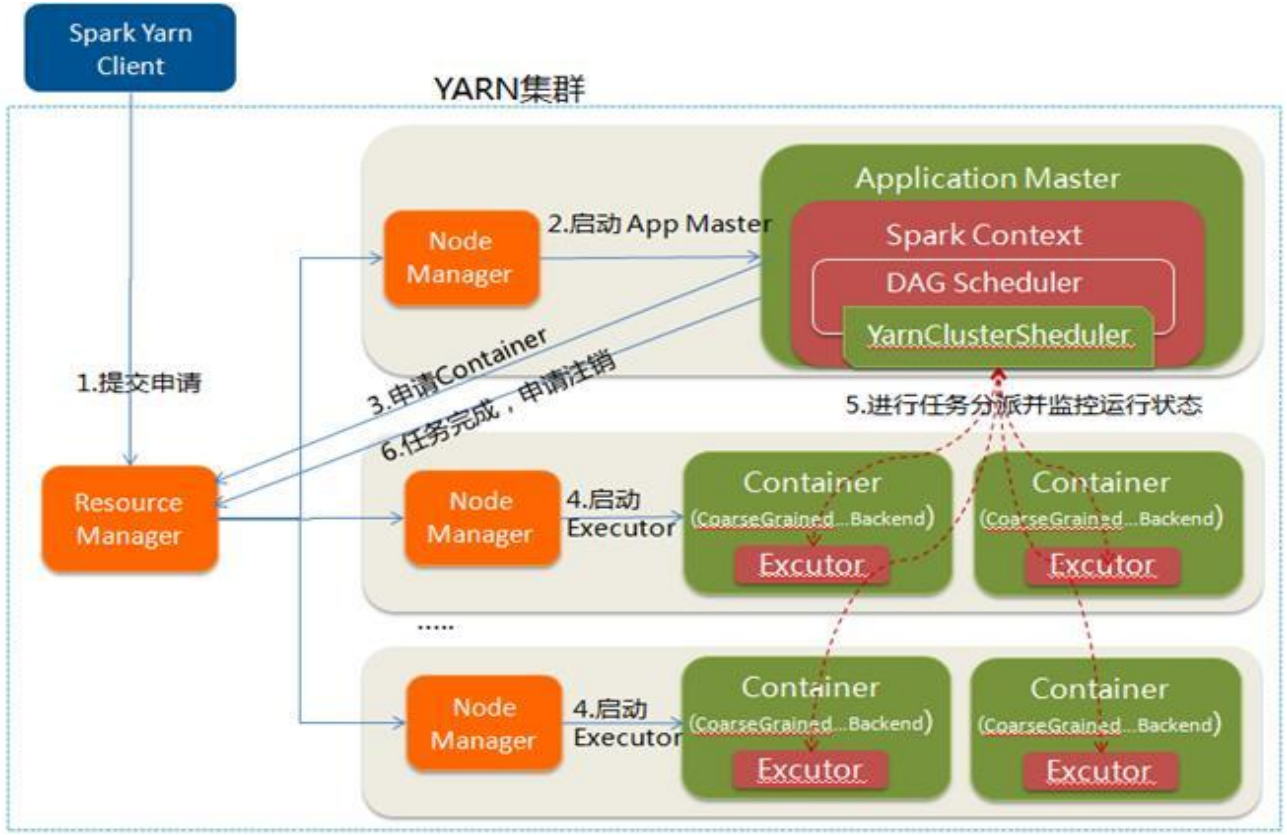
执行流程

Spark on YARN（YARN-Client）运行流程

1.Yarn Client向YARN的RM申请启动AM。同时在SparkContent初始化中将创建DAGScheduler和TASKScheduler等，由于我们选择的是Yarn- Client模式，程序会选择YarnClientClusterScheduler和YarnClientSchedulerBackend
2.RM收到请求后，在集群中选择一个NM为启动AM，与YARN-Cluster区别的是在该AM不运行SparkContext，只与SparkContext进行联系进行资源的分派
3.Client中的SparkContext初始化完毕后，与AM建立通讯，向RM注册，根据任务信息向RM申请资源
4.AM申请到资源Container后，与NM通信，要求在获得的 Container中启动CoarseGrainedExecutorBackend，CoarseGrainedExecutorBackend启 动后会向Client中的SparkContext注册并申请Task
5.Client中的SparkContext分配Task给CoarseGrainedExecutorBackend，CoarseGrainedExecutorBackend运行Task并向Driver汇报运行的状态和进度。Client可以在任务失败时重新启动任务
6.应用程序运行完成后，Client的SparkContext向RM申请注销并关闭自己

执行流程

Spark on YARN（YARN-Cluster）运行流程



执行流程

Spark on YARN（YARN-Cluster）运行流程

1.Spark Yarn Client向YARN提交应用程序，包括AM程序、启动AM的命令、需要运行的程序等
2.RM收到请求后，在集群中选择一个NM为该应用程序启动AM进行SparkContext的初始 化
3.AM向RM注册，这样可以直接通过RM查看应用程序的运行状态，然后它将采用轮询的方式通过RPC协议为各个任务申请资源，并监控运行状态直到运行结束
4.AM申请到资源后，便与对应的NM通信，在获得的container中启动CoarseGrainedExecutorBackend，CoarseGrainedExecutorBackend启 动后会向AM中的SparkContext注册并申请Task。SparkContext在Spark Application中初始化时，使用CoarseGrainedSchedulerBackend配合YarnClusterScheduler进行 任务的调度，其中YarnClusterScheduler只是对TaskSchedulerImpl的一个简单包装，增加了对Executor的等待逻辑等
5.AM中的SparkContext分配Task给CoarseGrainedExecutorBackend执 行，CoarseGrainedExecutorBackend运行Task并向AM汇报运行的状态和进度，以让AM可以在任务失败时重新启动任务
6.应用程序运行完成后，AM向RM申请注销并关闭自己

性能调优

开发调优

a.避免创建重复的**RDD**
b.尽可能复用同一个**RDD**
c.对多次使用的**RDD**进行持久化
d.尽量避免使用**shuffle**类算子
e.使用**map-side**预聚合的**shuffle**操作
f.使用高性能的算子
g.广播大变量
h.使用**Kryo**序列化
i.优化数据结构

性能调优

资源参数调优

num-executors
executor-memory
executor-cores
driver-memory
spark.default.parallelism
spark.storage.memoryFraction
spark.shuffle.memoryFraction

性能调优

数据倾斜调优

过滤少数导致倾斜的**key**

提高**shuffle**操作的并行度

两阶段聚合（局部聚合+全局聚合）

将**reduce join**转为**map join**

采样倾斜**key**并分拆**join**操作

使用随机前缀和扩容**RDD**进行**join**