# 1. Spark 内核概述

Spark内核泛指Spark的核心运行机制，包括Spark核心组件的运行机制、Spark任务调度机制、Spark内存管理机制、Spark核心功能的运行原理等，熟练掌握Spark内核原理，能够帮助我们更好地完成Spark代码设计，并能够帮助我们准确锁定项目运行过程中出现的问题的症结所在。

## 1.1 Spark核心组件回顾

### 1.1.1 Driver

Spark驱动器节点，用于执行Spark任务中的main方法，负责实际代码的执行工作。Driver在Spark作业执行时主要负责：

将用户程序转化为任务（job）；

在Executor之间调度任务（task）；

跟踪Executor的执行情况；

通过UI展示查询运行情况；

### 1.1.2 Executor

Spark Executor节点是一个JVM进程，负责在 Spark 作业中运行具体任务，任务彼此之间相互独立。Spark 应用启动时，Executor节点被同时启动，并且始终伴随着整个 Spark 应用的生命周期而存在。如果有Executor节点发生了故障或崩溃，Spark 应用也可以继续执行，会将出错节点上的任务调度到其他Executor节点上继续运行。

Executor有两个核心功能：

1. 负责运行组成Spark应用的任务，并将结果返回给Driver进程；

2. 它们通过自身的块管理器（Block Manager）为用户程序中要求缓存的 RDD 提供内存式存储。RDD 是直接缓存在Executor进程内的，因此任务可以在运行时充分利用缓存数据加速运算。

## 1.2 Spark通用运行流程概述

图1-1 Spark核心运行流程

图1-1为Spark通用运行流程，不论Spark以何种模式进行部署，任务提交后，都会先启动Driver进程，随后Driver进程向集群管理器注册应用程序，之后集群管理器根据此任务的配置文件分配Executor并启动，当Driver所需的资源全部满足后，Driver开始执行main函数，Spark查询为懒执行，当执行到action算子时开始反向推算，根据宽依赖进行stage的划分，随后每一个stage对应一个taskset，taskset中有多个task，根据本地化原则，task会被分发到指定的Executor去执行，在任务执行的过程中，Executor也会不断与Driver进行通信，报告任务运行情况。

# 2. Spark 部署模式

Spark支持3种集群管理器（Cluster Manager），分别为：

Standalone：独立模式，Spark原生的简单集群管理器，自带完整的服务，可单独部署到一个集群中，无需依赖任何其他资源管理系统，使用Standalone可以很方便地搭建一个集群；

Apache Mesos：一个强大的分布式资源管理框架，它允许多种不同的框架部署在其上，包括yarn；

Hadoop YARN：统一的资源管理机制，在上面可以运行多套计算框架，如map reduce、storm等，根据driver在集群中的位置不同，分为yarn client和yarn cluster。

实际上，除了上述这些通用的集群管理器外，Spark内部也提供了一些方便用户测试和学习的简单集群部署模式。由于在实际工厂环境下使用的绝大多数的集群管理器是Hadoop YARN，因此我们关注的重点是Hadoop YARN模式下的Spark集群部署。

Spark的运行模式取决于传递给SparkContext的MASTER环境变量的值，个别模式还需要辅助的程序接口来配合使用，目前支持的Master字符串及URL包括：

表2-1 Spark运行模式配置

|  |  |
| --- | --- |
| Master URL | Meaning |
| local | 在本地运行，只有一个工作进程，无并行计算能力。 |
| local[K] | 在本地运行，有K个工作进程，通常设置K为机器的CPU核心数量。 |
| local[\*] | 在本地运行，工作进程数量等于机器的CPU核心数量。 |
| spark://HOST:PORT | 以Standalone模式运行，这是Spark自身提供的集群运行模式，默认端口号: 7077。详细文档见:Spark standalone cluster。 |
| mesos://HOST:PORT | 在Mesos集群上运行，Driver进程和Worker进程运行在Mesos集群上，部署模式必须使用固定值:--deploy-mode cluster。详细文档见:MesosClusterDispatcher. |
| yarn-client | 在Yarn集群上运行，Driver进程在本地，Work进程在Yarn集群上，部署模式必须使用固定值:--deploy-mode client。Yarn集群地址必须在HADOOP\_CONF\_DIR or YARN\_CONF\_DIR变量里定义。 |
| yarn-cluster | 在Yarn集群上运行，Driver进程在Yarn集群上，Work进程也在Yarn集群上，部署模式必须使用固定值:--deploy-mode cluster。Yarn集群地址必须在HADOOP\_CONF\_DIR or YARN\_CONF\_DIR变量里定义。 |

用户在提交任务给Spark处理时，以下两个参数共同决定了Spark的运行方式。

· –master MASTER\_URL ：决定了Spark任务提交给哪种集群处理。

· –deploy-mode DEPLOY\_MODE：决定了Driver的运行方式，可选值为Client或者Cluster。

## 2.1 Standalone模式运行机制

Standalone集群有四个重要组成部分，分别是：

1) Driver：是一个进程，我们编写的Spark应用程序就运行在Driver上，由Driver进程执行；

2) Master：是一个进程，主要负责资源的调度和分配，并进行集群的监控等职责；

3) Worker：是一个进程，一个Worker运行在集群中的一台服务器上，主要负责两个职责，一个是用自己的内存存储RDD的某个或某些partition；另一个是启动其他进程和线程（Executor），对RDD上的partition进行并行的处理和计算。

4) Executor：是一个进程，一个Worker上可以运行多个Executor，Executor通过启动多个线程（task）来执行对RDD的partition进行并行计算，也就是执行我们对RDD定义的例如map、flatMap、reduce等算子操作。

### 2.1.1 Standalone Client模式

图2-1 Standalone Client模式

在Standalone Client模式下，Driver在任务提交的本地机器上运行，Driver启动后向Master注册应用程序，Master根据submit脚本的资源需求找到内部资源至少可以启动一个Executor的所有Worker，然后在这些Worker之间分配Executor，Worker上的Executor启动后会向Driver反向注册，所有的Executor注册完成后，Driver开始执行main函数，之后执行到Action算子时，开始划分stage，每个stage生成对应的taskSet，之后将task分发到各个Executor上执行。

2.1.2 Standalone Cluster模式

图2-2 Standalone Cluster模式

在Standalone Cluster模式下，任务提交后，Master会找到一个Worker启动Driver进程， Driver启动后向Master注册应用程序，Master根据submit脚本的资源需求找到内部资源至少可以启动一个Executor的所有Worker，然后在这些Worker之间分配Executor，Worker上的Executor启动后会向Driver反向注册，所有的Executor注册完成后，Driver开始执行main函数，之后执行到Action算子时，开始划分stage，每个stage生成对应的taskSet，之后将task分发到各个Executor上执行。

注意，Standalone的两种模式下（client/Cluster），Master在接到Driver注册Spark应用程序的请求后，会获取其所管理的剩余资源能够启动一个Executor的所有Worker，然后在这些Worker之间分发Executor，此时的分发只考虑Worker上的资源是否足够使用，直到当前应用程序所需的所有Executor都分配完毕，Executor反向注册完毕后，Driver开始执行main程序。

## 2.2 YARN模式运行机制

### 2.2.1 YARN Client模式

图2-3 YARN Client模式

在YARN Client模式下，Driver在任务提交的本地机器上运行，Driver启动后会和ResourceManager通讯申请启动ApplicationMaster，随后ResourceManager分配container，在合适的NodeManager上启动ApplicationMaster，此时的ApplicationMaster的功能相当于一个ExecutorLaucher，只负责向ResourceManager申请Executor内存。

ResourceManager接到ApplicationMaster的资源申请后会分配container，然后ApplicationMaster在资源分配指定的NodeManager上启动Executor进程，Executor进程启动后会向Driver反向注册，Executor全部注册完成后Driver开始执行main函数，之后执行到Action算子时，触发一个job，并根据宽依赖开始划分stage，每个stage生成对应的taskSet，之后将task分发到各个Executor上执行。

### 2.2.2 YARN Cluster模式

图2-4 YARN Cluster模式

在YARN Cluster模式下，任务提交后会和ResourceManager通讯申请启动ApplicationMaster，随后ResourceManager分配container，在合适的NodeManager上启动ApplicationMaster，此时的ApplicationMaster就是Driver。

Driver启动后向ResourceManager申请Executor内存，ResourceManager接到ApplicationMaster的资源申请后会分配container，然后在合适的NodeManager上启动Executor进程，Executor进程启动后会向Driver反向注册，Executor全部注册完成后Driver开始执行main函数，之后执行到Action算子时，触发一个job，并根据宽依赖开始划分stage，每个stage生成对应的taskSet，之后将task分发到各个Executor上执行。

# 3. Spark 通讯架构

## 3.1 Spark通信架构概述

Spark2.x版本使用Netty通讯框架作为内部通讯组件。spark 基于Netty新的rpc框架借鉴了Akka中的设计，它是基于Actor模型，如下图所示：

图4-1 Actor模型

Spark通讯框架中各个组件（Client/Master/Worker）可以认为是一个个独立的实体，各个实体之间通过消息来进行通信。具体各个组件之间的关系图如下：

图4-2 Spark通讯架构

Endpoint（Client/Master/Worker）有1个InBox和N个OutBox（N>=1，N取决于当前Endpoint与多少其他的Endpoint进行通信，一个与其通讯的其他Endpoint对应一个OutBox），Endpoint接收到的消息被写入InBox，发送出去的消息写入OutBox并被发送到其他Endpoint的InBox中。

## 3.2 Spark通讯架构解析

Spark通信架构如下图所示：

图 4-3 Spark通讯架构

1) RpcEndpoint：RPC端点，Spark针对每个节点（Client/Master/Worker）都称之为一个Rpc端点，且都实现RpcEndpoint接口，内部根据不同端点的需求，设计不同的消息和不同的业务处理，如果需要发送（询问）则调用Dispatcher；

2) RpcEnv：RPC上下文环境，每个RPC端点运行时依赖的上下文环境称为RpcEnv；

3) Dispatcher：消息分发器，针对于RPC端点需要发送消息或者从远程RPC接收到的消息，分发至对应的指令收件箱/发件箱。如果指令接收方是自己则存入收件箱，如果指令接收方不是自己，则放入发件箱；

4) Inbox：指令消息收件箱，一个本地RpcEndpoint对应一个收件箱，Dispatcher在每次向Inbox存入消息时，都将对应EndpointData加入内部ReceiverQueue中，另外Dispatcher创建时会启动一个单独线程进行轮询ReceiverQueue，进行收件箱消息消费；

5) RpcEndpointRef：RpcEndpointRef是对远程RpcEndpoint的一个引用。当我们需要向一个具体的RpcEndpoint发送消息时，一般我们需要获取到该RpcEndpoint的引用，然后通过该应用发送消息。

6) OutBox：指令消息发件箱，对于当前RpcEndpoint来说，一个目标RpcEndpoint对应一个发件箱，如果向多个目标RpcEndpoint发送信息，则有多个OutBox。当消息放入Outbox后，紧接着通过TransportClient将消息发送出去。消息放入发件箱以及发送过程是在同一个线程中进行；

7) RpcAddress：表示远程的RpcEndpointRef的地址，Host + Port。

8) TransportClient：Netty通信客户端，一个OutBox对应一个TransportClient，TransportClient不断轮询OutBox，根据OutBox消息的receiver信息，请求对应的远程TransportServer；

9) TransportServer：Netty通信服务端，一个RpcEndpoint对应一个TransportServer，接受远程消息后调用Dispatcher分发消息至对应收发件箱；

根据上面的分析，Spark通信架构的高层视图如下图所示：

图4-4 Spark通信框架高层视图

# 4. SparkContext解析

在Spark中由SparkContext负责与集群进行通讯、资源的申请以及任务的分配和监控等。当Worker节点中的Executor运行完毕Task后，Driver同时负责将SparkContext关闭。通常也可以使用SparkContext来代表驱动程序（Driver）。

图4-1 SparkContext与集群交互

SparkContext是用户通往Spark集群的唯一入口，可以用来在Spark集群中创建RDD、累加器和广播变量。SparkContext也是整个Spark应用程序中至关重要的一个对象，可以说是整个Application运行调度的核心（不包括资源调度）。

SparkContext的核心作用是初始化Spark应用程序运行所需的核心组件，包括高层调度器（DAGScheduler）、底层调度器（TaskScheduler）和调度器的通信终端（SchedulerBackend），同时还会负责Spark程序向Cluster Manager的注册等。

图4-2 SparkContext初始化组件

在实际的编码过程中，我们会先创建SparkConf实例，并对SparkConf的属性进行自定义设置，随后，将SparkConf作为SparkContext类的唯一构造参数传入来完成SparkContext实例对象的创建。

SparkContext在实例化的过程中会初始化DAGScheduler、TaskScheduler和SchedulerBackend，当RDD的action算子触发了作业（Job）后，SparkContext会调用DAGScheduler根据宽窄依赖将Job划分成几个小的阶段（Stage），TaskScheduler会调度每个Stage的任务（Task），另外，SchedulerBackend负责申请和管理集群为当前Application分配的计算资源（即Executor）。

如果我们将Spark Application比作汽车，那么SparkContext就是汽车的引擎，而SparkConf就是引擎的配置参数。

下图描述了Spark-On-Yarn模式下在任务调度期间，ApplicationMaster、Driver以及Executor内部模块的交互过程：

图4-2 Spark组件交互过程

Driver初始化SparkContext过程中，会分别初始化DAGScheduler、TaskScheduler、SchedulerBackend以及HeartbeatReceiver，并启动SchedulerBackend以及HeartbeatReceiver。SchedulerBackend通过ApplicationMaster申请资源，并不断从TaskScheduler中拿到合适的Task分发到Executor执行。HeartbeatReceiver负责接收Executor的心跳信息，监控Executor的存活状况，并通知到TaskScheduler。

# 5. Spark 任务调度机制

在工厂环境下，Spark集群的部署方式一般为YARN-Cluster模式，之后的内核分析内容中我们默认集群的部署方式为YARN-Cluster模式。

## 5.1 Spark任务提交流程

在上一章中我们讲解了Spark YARN-Cluster模式下的任务提交流程，如下图所示：

图5-1 YARN-Cluster任务提交流程

下面的时序图清晰地说明了一个Spark应用程序从提交到运行的完整流程：

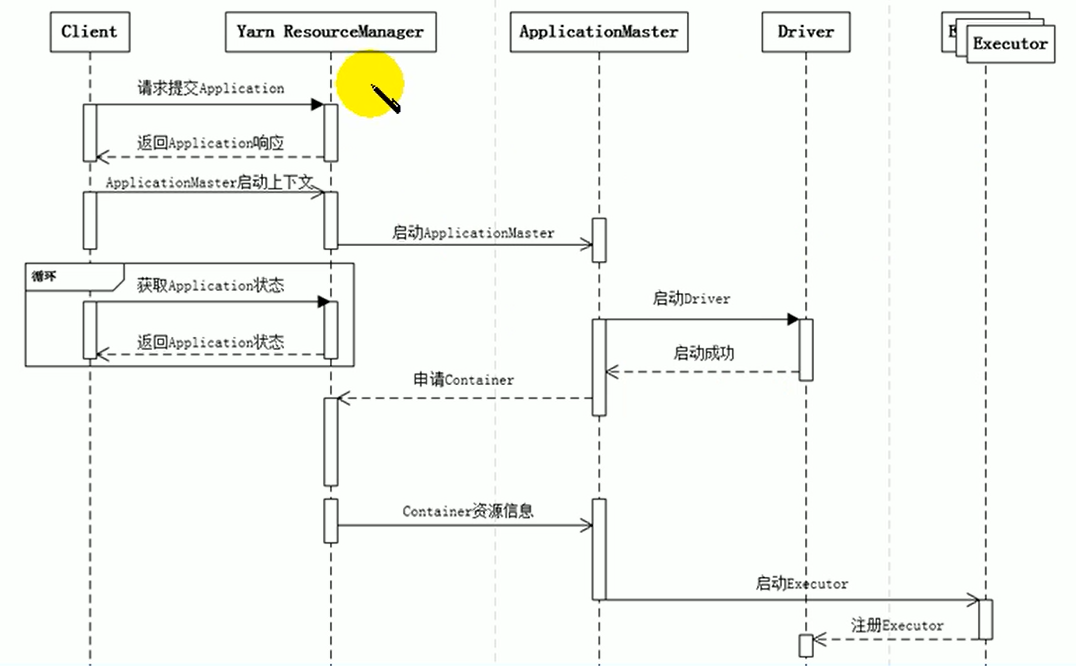


图5-2 Spark任务提交时序图

提交一个Spark应用程序，首先通过Client向ResourceManager请求启动一个Application，同时检查是否有足够的资源满足Application的需求，如果资源条件满足，则准备ApplicationMaster的启动上下文，交给ResourceManager，并循环监控Application状态。

当提交的资源队列中有资源时，ResourceManager会在某个NodeManager上启动ApplicationMaster进程，ApplicationMaster会单独启动Driver后台线程，当Driver启动后，ApplicationMaster会通过本地的RPC连接Driver，并开始向ResourceManager申请Container资源运行Executor进程（一个Executor对应与一个Container），当ResourceManager返回Container资源，ApplicationMaster则在对应的Container上启动Executor。

Driver线程主要是初始化SparkContext对象，准备运行所需的上下文，然后一方面保持与ApplicationMaster的RPC连接，通过ApplicationMaster申请资源，另一方面根据用户业务逻辑开始调度任务，将任务下发到已有的空闲Executor上。

当ResourceManager向ApplicationMaster返回Container资源时，ApplicationMaster就尝试在对应的Container上启动Executor进程，Executor进程起来后，会向Driver反向注册，注册成功后保持与Driver的心跳，同时等待Driver分发任务，当分发的任务执行完毕后，将任务状态上报给Driver。

从上述时序图可知，Client只负责提交Application并监控Application的状态。对于Spark的任务调度主要是集中在两个方面: 资源申请和任务分发，其主要是通过ApplicationMaster、Driver以及Executor之间来完成。

## 5.2 Spark任务调度概述

当Driver起来后，Driver则会根据用户程序逻辑准备任务，并根据Executor资源情况逐步分发任务。在详细阐述任务调度前，首先说明下Spark里的几个概念。一个Spark应用程序包括Job、Stage以及Task三个概念：

l Job是以Action方法为界，遇到一个Action方法则触发一个Job；

l Stage是Job的子集，以RDD宽依赖(即Shuffle)为界，遇到Shuffle做一次划分；

l Task是Stage的子集，以并行度(分区数)来衡量，分区数是多少，则有多少个task。

Spark的任务调度总体来说分两路进行，一路是Stage级的调度，一路是Task级的调度，总体调度流程如下图所示：

图5-3 Spark任务调度概览

Spark RDD通过其Transactions操作，形成了RDD血缘关系图，即DAG，最后通过Action的调用，触发Job并调度执行。DAGScheduler负责Stage级的调度，主要是将job切分成若干Stages，并将每个Stage打包成TaskSet交给TaskScheduler调度。TaskScheduler负责Task级的调度，将DAGScheduler给过来的TaskSet按照指定的调度策略分发到Executor上执行，调度过程中SchedulerBackend负责提供可用资源，其中SchedulerBackend有多种实现，分别对接不同的资源管理系统。

## 5.3 Spark Stage级调度

Spark的任务调度是从DAG切割开始，主要是由DAGScheduler来完成。当遇到一个Action操作后就会触发一个Job的计算，并交给DAGScheduler来提交，下图是涉及到Job提交的相关方法调用流程图。

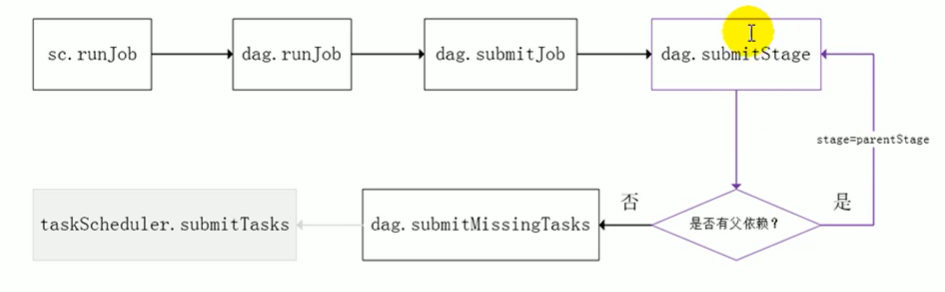


图5-5 Job提交调用栈

Job由最终的RDD和Action方法封装而成，SparkContext将Job交给DAGScheduler提交，它会根据RDD的血缘关系构成的DAG进行切分，将一个Job划分为若干Stages，具体划分策略是，由最终的RDD不断通过依赖回溯判断父依赖是否是宽依赖，即以Shuffle为界，划分Stage，窄依赖的RDD之间被划分到同一个Stage中，可以进行pipeline式的计算，如上图紫色流程部分。划分的Stages分两类，一类叫做ResultStage，为DAG最下游的Stage，由Action方法决定，另一类叫做ShuffleMapStage，为下游Stage准备数据，下面看一个简单的例子WordCount。

图5-6 WordCount实例

Job由saveAsTextFile触发，该Job由RDD-3和saveAsTextFile方法组成，根据RDD之间的依赖关系从RDD-3开始回溯搜索，直到没有依赖的RDD-0，在回溯搜索过程中，RDD-3依赖RDD-2，并且是宽依赖，所以在RDD-2和RDD-3之间划分Stage，RDD-3被划到最后一个Stage，即ResultStage中，RDD-2依赖RDD-1，RDD-1依赖RDD-0，这些依赖都是窄依赖，所以将RDD-0、RDD-1和RDD-2划分到同一个Stage，即ShuffleMapStage中，实际执行的时候，数据记录会一气呵成地执行RDD-0到RDD-2的转化。不难看出，其本质上是一个深度优先搜索算法。

一个Stage是否被提交，需要判断它的父Stage是否执行，只有在父Stage执行完毕才能提交当前Stage，如果一个Stage没有父Stage，那么从该Stage开始提交。Stage提交时会将Task信息（分区信息以及方法等）序列化并被打包成TaskSet交给TaskScheduler，一个Partition对应一个Task，另一方面TaskScheduler会监控Stage的运行状态，只有Executor丢失或者Task由于Fetch失败才需要重新提交失败的Stage以调度运行失败的任务，其他类型的Task失败会在TaskScheduler的调度过程中重试。

相对来说DAGScheduler做的事情较为简单，仅仅是在Stage层面上划分DAG，提交Stage并监控相关状态信息。TaskScheduler则相对较为复杂，下面详细阐述其细节。

## 5.4 Spark Task级调度

Spark Task的调度是由TaskScheduler来完成，由前文可知，DAGScheduler将Stage打包到TaskSet交给TaskScheduler，TaskScheduler会将TaskSet封装为TaskSetManager加入到调度队列中，TaskSetManager结构如下图所示。

图5-7 TaskManager结构

TaskSetManager负责监控管理同一个Stage中的Tasks，TaskScheduler就是以TaskSetManager为单元来调度任务。

TaskScheduler初始化后会启动SchedulerBackend，它负责跟外界打交道，接收Executor的注册信息，并维护Executor的状态，所以说SchedulerBackend是管“粮食”的，同时它在启动后会定期地去“询问”TaskScheduler有没有任务要运行，也就是说，它会定期地“问”TaskScheduler“我有这么余量，你要不要啊”，TaskScheduler在SchedulerBackend“问”它的时候，会从调度队列中按照指定的调度策略选择TaskSetManager去调度运行，大致方法调用流程如下图所示：

图5-9 task调度流程

图3-7中，将TaskSetManager加入rootPool调度池中之后，调用SchedulerBackend的riviveOffers方法给driverEndpoint发送ReviveOffer消息；driverEndpoint收到ReviveOffer消息后调用makeOffers方法，过滤出活跃状态的Executor（这些Executor都是任务启动时反向注册到Driver的Executor），然后将Executor封装成WorkerOffer对象；准备好计算资源（WorkerOffer）后，taskScheduler基于这些资源调用resourceOffer在Executor上分配task。

### 5.4.1 调度策略

前面讲到，TaskScheduler会先把DAGScheduler给过来的TaskSet封装成TaskSetManager扔到任务队列里，然后再从任务队列里按照一定的规则把它们取出来在SchedulerBackend给过来的Executor上运行。这个调度过程实际上还是比较粗粒度的，是面向TaskSetManager的。

TaskScheduler是以树的方式来管理任务队列，树中的节点类型为Schdulable，叶子节点为TaskSetManager，非叶子节点为Pool，下图是它们之间的继承关系。

图5-10 任务队列继承关系

调度队列的层次结构如下图所示：

图5-11 FIFO调度策略内存结构

TaskScheduler支持两种调度策略，一种是FIFO，也是默认的调度策略，另一种是FAIR。在TaskScheduler初始化过程中会实例化rootPool，表示树的根节点，是Pool类型。

FIFO调度策略

FIFO调度策略执行步骤如下：

1) 对s1和s2两个Schedulable的优先级（Schedulable类的一个属性，记为priority，值越小，优先级越高）；

2) 如果两个Schedulable的优先级相同，则对s1，s2所属的Stage的身份进行标识进行比较（Schedulable类的一个属性，记为priority，值越小，优先级越高）；

3) 如果比较的结果小于0，则优先调度s1，否则优先调度s2。

图5-12 FIFO调度策略内存结构

FAIR调度策略

FAIR调度策略的树结构如下图所示：

图5-13 FAIR调度策略内存结构

FAIR模式中有一个rootPool和多个子Pool，各个子Pool中存储着所有待分配的TaskSetMagager。

可以通过在Properties中指定spark.scheduler.pool属性，指定调度池中的某个调度池作为TaskSetManager的父调度池，如果根调度池不存在此属性值对应的调度池，会创建以此属性值为名称的调度池作为TaskSetManager的父调度池，并将此调度池作为根调度池的子调度池。

在FAIR模式中，需要先对子Pool进行排序，再对子Pool里面的TaskSetMagager进行排序，因为Pool和TaskSetMagager都继承了Schedulable特质，因此使用相同的排序算法。

排序过程的比较是基于Fair-share来比较的，每个要排序的对象包含三个属性: runningTasks值（正在运行的Task数）、minShare值、weight值，比较时会综合考量runningTasks值，minShare值以及weight值。

注意，minShare、weight的值均在公平调度配置文件fairscheduler.xml中被指定，调度池在构建阶段会读取此文件的相关配置。

1) 如果A对象的runningTasks大于它的minShare，B对象的runningTasks小于它的minShare，那么B排在A前面；（runningTasks比minShare小的先执行）

2) 如果A、B对象的runningTasks都小于它们的minShare，那么就比较runningTasks与minShare的比值（minShare使用率），谁小谁排前面；（minShare使用率低的先执行）

3) 如果A、B对象的runningTasks都大于它们的minShare，那么就比较runningTasks与weight的比值（权重使用率），谁小谁排前面。（权重使用率低的先执行）

4) 如果上述比较均相等，则比较名字。

整体上来说就是通过minShare和weight这两个参数控制比较过程，可以做到让minShare使用率和权重使用率少（实际运行task比例较少）的先运行。

FAIR模式排序完成后，所有的TaskSetManager被放入一个ArrayBuffer里，之后依次被取出并发送给Executor执行。

从调度队列中拿到TaskSetManager后，由于TaskSetManager封装了一个Stage的所有Task，并负责管理调度这些Task，那么接下来的工作就是TaskSetManager按照一定的规则一个个取出Task给TaskScheduler，TaskScheduler再交给SchedulerBackend去发到Executor上执行。

### 5.4.2 本地化调度

DAGScheduler切割Job，划分Stage, 通过调用submitStage来提交一个Stage对应的tasks，submitStage会调用submitMissingTasks，submitMissingTasks 确定每个需要计算的 task 的preferredLocations，通过调用getPreferrdeLocations()得到partition 的优先位置，由于一个partition对应一个task，此partition的优先位置就是task的优先位置，对于要提交到TaskScheduler的TaskSet中的每一个task，该task优先位置与其对应的partition对应的优先位置一致。

从调度队列中拿到TaskSetManager后，那么接下来的工作就是TaskSetManager按照一定的规则一个个取出task给TaskScheduler，TaskScheduler再交给SchedulerBackend去发到Executor上执行。前面也提到，TaskSetManager封装了一个Stage的所有task，并负责管理调度这些task。

根据每个task的优先位置，确定task的Locality级别，Locality一共有五种，优先级由高到低顺序：

表5-1 Spark本地化等级

|  |  |
| --- | --- |
| 名称 | 解析 |
| PROCESS\_LOCAL | 进程本地化，task和数据在同一个Executor中，性能最好。 |
| NODE\_LOCAL | 节点本地化，task和数据在同一个节点中，但是task和数据不在同一个Executor中，数据需要在进程间进行传输。 |
| RACK\_LOCAL | 机架本地化，task和数据在同一个机架的两个节点上，数据需要通过网络在节点之间进行传输。 |
| NO\_PREF | 对于task来说，从哪里获取都一样，没有好坏之分。 |
| ANY | task和数据可以在集群的任何地方，而且不在一个机架中，性能最差。 |

在调度执行时，Spark调度总是会尽量让每个task以最高的本地性级别来启动，当一个task以X本地性级别启动，但是该本地性级别对应的所有节点都没有空闲资源而启动失败，此时并不会马上降低本地性级别启动而是在某个时间长度内再次以X本地性级别来启动该task，若超过限时时间则降级启动，去尝试下一个本地性级别，依次类推。

可以通过调大每个类别的最大容忍延迟时间，在等待阶段对应的Executor可能就会有相应的资源去执行此task，这就在在一定程度上提到了运行性能。

### 5.4.3 失败重试与黑名单机制

除了选择合适的Task调度运行外，还需要监控Task的执行状态，前面也提到，与外部打交道的是SchedulerBackend，Task被提交到Executor启动执行后，Executor会将执行状态上报给SchedulerBackend，SchedulerBackend则告诉TaskScheduler，TaskScheduler找到该Task对应的TaskSetManager，并通知到该TaskSetManager，这样TaskSetManager就知道Task的失败与成功状态，对于失败的Task，会记录它失败的次数，如果失败次数还没有超过最大重试次数，那么就把它放回待调度的Task池子中，否则整个Application失败。

在记录Task失败次数过程中，会记录它上一次失败所在的Executor Id和Host，这样下次再调度这个Task时，会使用黑名单机制，避免它被调度到上一次失败的节点上，起到一定的容错作用。黑名单记录Task上一次失败所在的Executor Id和Host，以及其对应的“拉黑”时间，“拉黑”时间是指这段时间内不要再往这个节点上调度这个Task了。

# 6. Spark Shuffle解析

## 6.1 Shuffle的核心要点

### 6.1.1 ShuffleMapStage与FinalStage

图6-1 ShuffleMapStage与FinalStage

在划分stage时，最后一个stage称为FinalStage，它本质上是一个ResultStage对象，前面的所有stage被称为ShuffleMapStage。

ShuffleMapStage的结束伴随着shuffle文件的写磁盘。

ResultStage基本上对应代码中的action算子，即将一个函数应用在RDD的各个partition的数据集上，意味着一个job的运行结束。

6.1.2 Shuffle中的任务个数

1. map端task个数的确定

Shuffle过程中的task个数由RDD分区数决定，而RDD的分区个数与参数spark.default.parallelism有密切关系。

在Yarn Cluster模式下，如果没有手动设置spark.default.parallelism ，则有：

Others: total number of cores on all executor nodes or 2, whichever is larger.

spark.default.parallelism =  max(所有executor使用的core总数， 2)

如果进行了手动配置，则：

spark.default.parallelism = 配置值

还有一个重要的配置：

The maximum number of bytes to pack into a single partition when reading files.

spark.files.maxPartitionBytes = 128 M (默认)

代表着rdd的一个分区能存放数据的最大字节数，如果一个400MB的文件，只分了两个区，则在action时会发生错误。

当一个spark应用程序执行时，生成sparkContext，同时会生成两个参数，由上面得到的spark.default.parallelism推导出这两个参数的值:

sc.defaultParallelism = spark.default.parallelism

sc.defaultMinPartitions = min(spark.default.parallelism,2)

当以上参数确定后，就可以推算RDD分区数目了。

① 通过scala 集合方式parallelize生成的RDD

val rdd = sc.parallelize(1 to 10)

这种方式下，如果在parallelize操作时没有指定分区数，则有：

rdd的分区数 = sc.defaultParallelism

② 在本地文件系统通过textFile方式生成的RDD

val rdd = sc.textFile(“path/file”)

rdd的分区数 = max（本地file的分片数， sc.defaultMinPartitions）

③ 在HDFS文件系统生成的RDD

rdd的分区数 = max（HDFS文件的Block数目， sc.defaultMinPartitions）

④ 从HBase数据表获取数据并转换为RDD

rdd的分区数 = Table的region个数

⑤ 通过获取json（或者parquet等等）文件转换成的DataFrame

rdd的分区数 = 该文件在文件系统中存放的Block数目

⑥ Spark Streaming获取Kafka消息对应的分区数

基于Receiver：

在Receiver的方式中，Spark中的partition和kafka中的partition并不是相关的，所以如果我们加大每个topic的partition数量，

仅仅是增加线程来处理由单一Receiver消费的主题。但是这并没有增加Spark在处理数据上的并行度。

基于DirectDStream：

Spark会创建跟Kafka partition一样多的RDD partition，并且会并行从Kafka中读取数据，所以在Kafka partition和RDD partition之间，有一个一对一的映射关系。

2. reduce端task个数的确定

Reduce端进行数据的聚合，一部分聚合算子可以手动指定reducetask的并行度，如果没有指定，则以map端的最后一个RDD的分区数作为其分区数，那么分区数就决定了reduce端的task的个数。

### 6.1.3 reduce端数据的读取

根据stage的划分我们知道，map端task和reduce端task不在相同的stage中，map task位于ShuffleMapStage，reduce task位于ResultStage，map task会先执行，那么后执行的reduce task如何知道从哪里去拉取map task落盘后的数据呢？

reduce端的数据拉取过程如下：

map task 执行完毕后会将计算状态以及磁盘小文件位置等信息封装到mapStatue对象中，然后由本进程中的MapOutPutTrackerWorker对象将mapStatus对象发送给Driver进程的MapOutPutTrackerMaster对象；

在reduce task开始执行之前会先让本进程中的MapOutputTrackerWorker向Driver进程中的MapoutPutTrakcerMaster发动请求，请求磁盘小文件位置信息；

当所有的Map task执行完毕后，Driver进程中的MapOutPutTrackerMaster就掌握了所有的磁盘小文件的位置信息。此时MapOutPutTrackerMaster会告诉MapOutPutTrackerWorker磁盘小文件的位置信息；

完成之前的操作之后，由BlockTransforService去Executor所在的节点拉数据，默认会启动五个子线程。每次拉取的数据量不能超过48M（reduce task每次最多拉取48M数据，将拉来的数据存储到Executor内存的20%内存中）。

## 6.2 HashShuffle解析

以下的讨论都假设每个Executor有1个CPU core。

1. 未经优化的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到拉取完，并得到最终的结果。

未优化的HashShuffleManager工作原理如图1-7所示：

图6-2 未优化的HashShuffleManager工作原理

2. 优化后的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：CPU core的数量 \* 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。

优化后的HashShuffleManager工作原理如图1-8所示：

图6-3 优化后的HashShuffleManager工作原理d

## 6.3 SortShuffle解析

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。

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，也就会产生多个临时文件。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工作原理如图1-9所示：

图6-4 普通运行机制的SortShuffleManager工作原理

bypass运行机制

bypass运行机制的触发条件如下：

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

l 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。

普通运行机制的SortShuffleManager工作原理如图1-10所示：

图6-5 bypass运行机制的SortShuffleManager工作原理

# 7. Spark 内存管理

在执行Spark 的应用程序时，Spark 集群会启动 Driver 和 Executor 两种 JVM 进程，前者为主控进程，负责创建 Spark 上下文，提交 Spark 作业（Job），并将作业转化为计算任务（Task），在各个 Executor 进程间协调任务的调度，后者负责在工作节点上执行具体的计算任务，并将结果返回给 Driver，同时为需要持久化的 RDD 提供存储功能。由于 Driver 的内存管理相对来说较为简单，本节主要对 Executor 的内存管理进行分析，下文中的 Spark 内存均特指 Executor 的内存。

## 7.1 堆内和堆外内存规划

作为一个 JVM 进程，Executor 的内存管理建立在 JVM 的内存管理之上，Spark 对 JVM 的堆内（On-heap）空间进行了更为详细的分配，以充分利用内存。同时，Spark 引入了堆外（Off-heap）内存，使之可以直接在工作节点的系统内存中开辟空间，进一步优化了内存的使用。

堆内内存受到JVM统一管理，堆外内存是直接向操作系统进行内存的申请和释放。

图7-1 Executor堆内与堆外内存

堆内内存

堆内内存的大小，由 Spark 应用程序启动时的 –executor-memory 或 spark.executor.memory 参数配置。Executor 内运行的并发任务共享 JVM 堆内内存，这些任务在缓存 RDD 数据和广播（Broadcast）数据时占用的内存被规划为存储（Storage）内存，而这些任务在执行 Shuffle 时占用的内存被规划为执行（Execution）内存，剩余的部分不做特殊规划，那些 Spark 内部的对象实例，或者用户定义的 Spark 应用程序中的对象实例，均占用剩余的空间。不同的管理模式下，这三部分占用的空间大小各不相同。

Spark 对堆内内存的管理是一种逻辑上的”规划式”的管理，因为对象实例占用内存的申请和释放都由 JVM 完成，Spark 只能在申请后和释放前记录这些内存，我们来看其具体流程：

申请内存流程如下：

Spark 在代码中 new 一个对象实例；

JVM 从堆内内存分配空间，创建对象并返回对象引用；

Spark 保存该对象的引用，记录该对象占用的内存。

释放内存流程如下：

1. Spark记录该对象释放的内存，删除该对象的引用；

2. 等待JVM的垃圾回收机制释放该对象占用的堆内内存。

我们知道，JVM 的对象可以以序列化的方式存储，序列化的过程是将对象转换为二进制字节流，本质上可以理解为将非连续空间的链式存储转化为连续空间或块存储，在访问时则需要进行序列化的逆过程——反序列化，将字节流转化为对象，序列化的方式可以节省存储空间，但增加了存储和读取时候的计算开销。

对于 Spark 中序列化的对象，由于是字节流的形式，其占用的内存大小可直接计算，而对于非序列化的对象，其占用的内存是通过周期性地采样近似估算而得，即并不是每次新增的数据项都会计算一次占用的内存大小，这种方法降低了时间开销但是有可能误差较大，导致某一时刻的实际内存有可能远远超出预期。此外，在被 Spark 标记为释放的对象实例，很有可能在实际上并没有被 JVM 回收，导致实际可用的内存小于 Spark 记录的可用内存。所以 Spark 并不能准确记录实际可用的堆内内存，从而也就无法完全避免内存溢出（OOM, Out of Memory）的异常。

虽然不能精准控制堆内内存的申请和释放，但 Spark 通过对存储内存和执行内存各自独立的规划管理，可以决定是否要在存储内存里缓存新的 RDD，以及是否为新的任务分配执行内存，在一定程度上可以提升内存的利用率，减少异常的出现。

堆外内存

为了进一步优化内存的使用以及提高 Shuffle 时排序的效率，Spark 引入了堆外（Off-heap）内存，使之可以直接在工作节点的系统内存中开辟空间，存储经过序列化的二进制数据。

堆外内存意味着把内存对象分配在Java虚拟机的堆以外的内存，这些内存直接受操作系统管理（而不是虚拟机）。这样做的结果就是能保持一个较小的堆，以减少垃圾收集对应用的影响。

利用 JDK Unsafe API（从 Spark 2.0 开始，在管理堆外的存储内存时不再基于 Tachyon，而是与堆外的执行内存一样，基于 JDK Unsafe API 实现），Spark 可以直接操作系统堆外内存，减少了不必要的内存开销，以及频繁的 GC 扫描和回收，提升了处理性能。堆外内存可以被精确地申请和释放（堆外内存之所以能够被精确的申请和释放，是由于内存的申请和释放不再通过JVM机制，而是直接向操作系统申请，JVM对于内存的清理是无法准确指定时间点的，因此无法实现精确的释放），而且序列化的数据占用的空间可以被精确计算，所以相比堆内内存来说降低了管理的难度，也降低了误差。

在默认情况下堆外内存并不启用，可通过配置 spark.memory.offHeap.enabled 参数启用，并由 spark.memory.offHeap.size 参数设定堆外空间的大小。除了没有 other 空间，堆外内存与堆内内存的划分方式相同，所有运行中的并发任务共享存储内存和执行内存。

（\*该部分内存主要用于程序的共享库、Perm Space、线程Stack和一些Memory mapping等, 或者类C方式allocate object）

## 7.2 内存空间分配

1. 静态内存管理

在 Spark 最初采用的静态内存管理机制下，存储内存、执行内存和其他内存的大小在 Spark 应用程序运行期间均为固定的，但用户可以应用程序启动前进行配置，堆内内存的分配如图 2 所示：

图7-2 静态内存管理——堆内内存

可以看到，可用的堆内内存的大小需要按照代码清单1-1的方式计算：

代码清单1-1 堆内内存计算公式

可用的存储内存 = systemMaxMemory \* spark.storage.memoryFraction \* spark.storage.safety Fraction

可用的执行内存 = systemMaxMemory \* spark.shuffle.memoryFraction \* spark.shuffle.safety Fraction

其中 systemMaxMemory 取决于当前 JVM 堆内内存的大小，最后可用的执行内存或者存储内存要在此基础上与各自的 memoryFraction 参数和 safetyFraction 参数相乘得出。上述计算公式中的两个 safetyFraction 参数，其意义在于在逻辑上预留出 1-safetyFraction 这么一块保险区域，降低因实际内存超出当前预设范围而导致 OOM 的风险（上文提到，对于非序列化对象的内存采样估算会产生误差）。值得注意的是，这个预留的保险区域仅仅是一种逻辑上的规划，在具体使用时 Spark 并没有区别对待，和”其它内存”一样交给了 JVM 去管理。

Storage内存和Execution内存都有预留空间，目的是防止OOM，因为Spark堆内内存大小的记录是不准确的，需要留出保险区域。

堆外的空间分配较为简单，只有存储内存和执行内存，如图1-3所示。可用的执行内存和存储内存占用的空间大小直接由参数spark.memory.storageFraction 决定，由于堆外内存占用的空间可以被精确计算，所以无需再设定保险区域。

图7-3 静态内存管理

静态内存管理机制实现起来较为简单，但如果用户不熟悉 Spark 的存储机制，或没有根据具体的数据规模和计算任务或做相应的配置，很容易造成”一半海水，一半火焰”的局面，即存储内存和执行内存中的一方剩余大量的空间，而另一方却早早被占满，不得不淘汰或移出旧的内容以存储新的内容。由于新的内存管理机制的出现，这种方式目前已经很少有开发者使用，出于兼容旧版本的应用程序的目的，Spark 仍然保留了它的实现。

统一内存管理

Spark 1.6 之后引入的统一内存管理机制，与静态内存管理的区别在于存储内存和执行内存共享同一块空间，可以动态占用对方的空闲区域，统一内存管理的堆内内存结构如图 1-4所示：

图7-4 统一内存管理——堆内内存

统一内存管理的堆外内存结构如图 1-5所示：

图7-5 统一内存管理——堆外内存

其中最重要的优化在于动态占用机制，其规则如下：

设定基本的存储内存和执行内存区域（spark.storage.storageFraction 参数），该设定确定了双方各自拥有的空间的范围；

双方的空间都不足时，则存储到硬盘；若己方空间不足而对方空余时，可借用对方的空间;（存储空间不足是指不足以放下一个完整的 Block）

3. 执行内存的空间被对方占用后，可让对方将占用的部分转存到硬盘，然后”归还”借用的空间；

4. 存储内存的空间被对方占用后，无法让对方”归还”，因为需要考虑 Shuffle 过程中的很多因素，实现起来较为复杂。

统一内存管理的动态占用机制如图 1-6所示：

图7-6 同一内存管理——动态占用机制

凭借统一内存管理机制，Spark 在一定程度上提高了堆内和堆外内存资源的利用率，降低了开发者维护 Spark 内存的难度，但并不意味着开发者可以高枕无忧。如果存储内存的空间太大或者说缓存的数据过多，反而会导致频繁的全量垃圾回收，降低任务执行时的性能，因为缓存的 RDD 数据通常都是长期驻留内存的。所以要想充分发挥 Spark 的性能，需要开发者进一步了解存储内存和执行内存各自的管理方式和实现原理。

## 7.3 存储内存管理

RDD的持久化机制

弹性分布式数据集（RDD）作为 Spark 最根本的数据抽象，是只读的分区记录（Partition）的集合，只能基于在稳定物理存储中的数据集上创建，或者在其他已有的 RDD 上执行转换（Transformation）操作产生一个新的 RDD。转换后的 RDD 与原始的 RDD 之间产生的依赖关系，构成了血统（Lineage）。凭借血统，Spark 保证了每一个 RDD 都可以被重新恢复。但 RDD 的所有转换都是惰性的，即只有当一个返回结果给 Driver 的行动（Action）发生时，Spark 才会创建任务读取 RDD，然后真正触发转换的执行。

Task 在启动之初读取一个分区时，会先判断这个分区是否已经被持久化，如果没有则需要检查 Checkpoint 或按照血统重新计算。所以如果一个 RDD 上要执行多次行动，可以在第一次行动中使用 persist 或 cache 方法，在内存或磁盘中持久化或缓存这个 RDD，从而在后面的行动时提升计算速度。

事实上，cache 方法是使用默认的 MEMORY\_ONLY 的存储级别将 RDD 持久化到内存，故缓存是一种特殊的持久化。 堆内和堆外存储内存的设计，便可以对缓存 RDD 时使用的内存做统一的规划和管理。

RDD 的持久化由 Spark 的 Storage 模块负责，实现了 RDD 与物理存储的解耦合。Storage 模块负责管理 Spark 在计算过程中产生的数据，将那些在内存或磁盘、在本地或远程存取数据的功能封装了起来。在具体实现时 Driver 端和 Executor 端的 Storage 模块构成了主从式的架构，即 Driver 端的 BlockManager 为 Master，Executor 端的 BlockManager 为 Slave。

Storage 模块在逻辑上以 Block 为基本存储单位，RDD 的每个 Partition 经过处理后唯一对应一个 Block（BlockId 的格式为 rdd\_RDD-ID\_PARTITION-ID ）。Driver端的Master 负责整个 Spark 应用程序的 Block 的元数据信息的管理和维护，而Executor端的 Slave 需要将 Block 的更新等状态上报到 Master，同时接收 Master 的命令，例如新增或删除一个 RDD。

图7-7 Storage模块示意图

在对 RDD 持久化时，Spark 规定了 MEMORY\_ONLY、MEMORY\_AND\_DISK 等 7 种不同的[存储级别](https://www.cnblogs.com/dflmg/p/10430177.html#rdd-persistence)，而存储级别是以下 5 个变量的组合：

代码清单5-1 resourceOffer代码

class StorageLevel private(

private var \_useDisk: Boolean, //磁盘

private var \_useMemory: Boolean, //这里其实是指堆内内存

private var \_useOffHeap: Boolean, //堆外内存

private var \_deserialized: Boolean, //是否为非序列化

private var \_replication: Int = 1 //副本个数

)

Spark中7种存储级别如下：

表5-1 Spark持久化级别

|  |  |
| --- | --- |
| 持久化级别 | 含义 |
| MEMORY\_ONLY | 以非序列化的Java对象的方式持久化在JVM内存中。如果内存无法完全存储RDD所有的partition，那么那些没有持久化的partition就会在下一次需要使用它们的时候，重新被计算 |
| MEMORY\_AND\_DISK | 同上，但是当某些partition无法存储在内存中时，会持久化到磁盘中。下次需要使用这些partition时，需要从磁盘上读取 |
| MEMORY\_ONLY\_SER | 同MEMORY\_ONLY，但是会使用Java序列化方式，将Java对象序列化后进行持久化。可以减少内存开销，但是需要进行反序列化，因此会加大CPU开销 |
| MEMORY\_AND\_DISK\_SER | 同MEMORY\_AND\_DISK，但是使用序列化方式持久化Java对象 |
| DISK\_ONLY | 使用非序列化Java对象的方式持久化，完全存储到磁盘上 |
| MEMORY\_ONLY\_2  MEMORY\_AND\_DISK\_2  等等 | 如果是尾部加了2的持久化级别，表示将持久化数据复用一份，保存到其他节点，从而在数据丢失时，不需要再次计算，只需要使用备份数据即可 |

通过对数据结构的分析，可以看出存储级别从三个维度定义了 RDD 的 Partition（同时也就是 Block）的存储方式：

1) 存储位置：磁盘／堆内内存／堆外内存。如 MEMORY\_AND\_DISK 是同时在磁盘和堆内内存上存储，实现了冗余备份。OFF\_HEAP 则是只在堆外内存存储，目前选择堆外内存时不能同时存储到其他位置。

2) 存储形式：Block 缓存到存储内存后，是否为非序列化的形式。如 MEMORY\_ONLY 是非序列化方式存储，OFF\_HEAP 是序列化方式存储。

3) 副本数量：大于 1 时需要远程冗余备份到其他节点。如 DISK\_ONLY\_2 需要远程备份 1 个副本。

RDD的缓存过程

RDD 在缓存到存储内存之前，Partition 中的数据一般以迭代器（[Iterator](http://www.scala-lang.org/docu/files/collections-api/collections_43.html)）的数据结构来访问，这是 Scala 语言中一种遍历数据集合的方法。通过 Iterator 可以获取分区中每一条序列化或者非序列化的数据项(Record)，这些 Record 的对象实例在逻辑上占用了 JVM 堆内内存的 other 部分的空间，同一 Partition 的不同 Record 的存储空间并不连续。

RDD 在缓存到存储内存之后，Partition 被转换成 Block，Record 在堆内或堆外存储内存中占用一块连续的空间。将Partition由不连续的存储空间转换为连续存储空间的过程，Spark称之为"展开"（Unroll）。

Block 有序列化和非序列化两种存储格式，具体以哪种方式取决于该 RDD 的存储级别。非序列化的 Block 以一种 DeserializedMemoryEntry 的数据结构定义，用一个数组存储所有的对象实例，序列化的 Block 则以 SerializedMemoryEntry的数据结构定义，用字节缓冲区（ByteBuffer）来存储二进制数据。每个 Executor 的 Storage 模块用一个链式 Map 结构（LinkedHashMap）来管理堆内和堆外存储内存中所有的 Block 对象的实例，对这个 LinkedHashMap 新增和删除间接记录了内存的申请和释放。

因为不能保证存储空间可以一次容纳 Iterator 中的所有数据，当前的计算任务在 Unroll 时要向 MemoryManager 申请足够的 Unroll 空间来临时占位，空间不足则 Unroll 失败，空间足够时可以继续进行。

对于序列化的 Partition，其所需的 Unroll 空间可以直接累加计算，一次申请。

对于非序列化的 Partition 则要在遍历 Record 的过程中依次申请，即每读取一条 Record，采样估算其所需的 Unroll 空间并进行申请，空间不足时可以中断，释放已占用的 Unroll 空间。

如果最终 Unroll 成功，当前 Partition 所占用的 Unroll 空间被转换为正常的缓存 RDD 的存储空间，如下图所示。

图7-8 Spark Unroll

在静态内存管理时，Spark 在存储内存中专门划分了一块 Unroll 空间，其大小是固定的，统一内存管理时则没有对 Unroll 空间进行特别区分，当存储空间不足时会根据动态占用机制进行处理。

淘汰与落盘

由于同一个 Executor 的所有的计算任务共享有限的存储内存空间，当有新的 Block 需要缓存但是剩余空间不足且无法动态占用时，就要对 LinkedHashMap 中的旧 Block 进行淘汰（Eviction），而被淘汰的 Block 如果其存储级别中同时包含存储到磁盘的要求，则要对其进行落盘（Drop），否则直接删除该 Block。

存储内存的淘汰规则为：

l 被淘汰的旧 Block 要与新 Block 的 MemoryMode 相同，即同属于堆外或堆内内存；

l 新旧 Block 不能属于同一个 RDD，避免循环淘汰；

l 旧 Block 所属 RDD 不能处于被读状态，避免引发一致性问题；

l 遍历 LinkedHashMap 中 Block，按照最近最少使用（LRU）的顺序淘汰，直到满足新 Block 所需的空间。其中 LRU 是 LinkedHashMap 的特性。

落盘的流程则比较简单，如果其存储级别符合\_useDisk 为 true 的条件，再根据其\_deserialized 判断是否是非序列化的形式，若是则对其进行序列化，最后将数据存储到磁盘，在 Storage 模块中更新其信息。

## 7.4 执行内存管理

执行内存主要用来存储任务在执行 Shuffle 时占用的内存，Shuffle 是按照一定规则对 RDD 数据重新分区的过程，我们来看 Shuffle 的 Write 和 Read 两阶段对执行内存的使用：

l Shuffle Write

在 map 端会采用 ExternalSorter 进行外排，在内存中存储数据时主要占用堆内执行空间。

l Shuffle Read

1) 在对 reduce 端的数据进行聚合时，要将数据交给 Aggregator 处理，在内存中存储数据时占用堆内执行空间。

2) 如果需要进行最终结果排序，则要将再次将数据交给 ExternalSorter 处理，占用堆内执行空间。

在 ExternalSorter 和 Aggregator 中，Spark 会使用一种叫 AppendOnlyMap 的哈希表在堆内执行内存中存储数据，但在 Shuffle 过程中所有数据并不能都保存到该哈希表中，当这个哈希表占用的内存会进行周期性地采样估算，当其大到一定程度，无法再从 MemoryManager 申请到新的执行内存时，Spark 就会将其全部内容存储到磁盘文件中，这个过程被称为溢存(Spill)，溢存到磁盘的文件最后会被归并(Merge)。

Spark 的存储内存和执行内存有着截然不同的管理方式：对于存储内存来说，Spark 用一个 LinkedHashMap 来集中管理所有的 Block，Block 由需要缓存的 RDD 的 Partition 转化而成；而对于执行内存，Spark 用 AppendOnlyMap 来存储 Shuffle 过程中的数据，在 Tungsten 排序中甚至抽象成为页式内存管理，开辟了全新的 JVM 内存管理机制。

# 8. Spark 核心组件解析

## 8.1 BlockManager数据存储与管理机制

BlockManager是整个Spark底层负责数据存储与管理的一个组件，Driver和Executor的所有数据都由对应的BlockManager进行管理。

Driver上有BlockManagerMaster，负责对各个节点上的BlockManager内部管理的数据的元数据进行维护，比如block的增删改等操作，都会在这里维护好元数据的变更。

每个节点都有一个BlockManager，每个BlockManager创建之后，第一件事即使去向BlockManagerMaster进行注册，此时BlockManagerMaster会为其长难句对应的BlockManagerInfo。

BlockManager运行原理如下图所示：

图8-1 BlockManager原理

BlockManagerMaster与BlockManager的关系非常像NameNode与DataNode的关系，BlockManagerMaster中保存中BlockManager内部管理数据的元数据，进行维护，当BlockManager进行Block增删改等操作时，都会在BlockManagerMaster中进行元数据的变更，这与NameNode维护DataNode的元数据信息，DataNode中数据发生变化时NameNode中的元数据信息也会相应变化是一致的。

每个节点上都有一个BlockManager，BlockManager中有3个非常重要的组件：

DiskStore：负责对磁盘数据进行读写；

MemoryStore：负责对内存数据进行读写；

BlockTransferService：负责建立BlockManager到远程其他节点的BlockManager的连接，负责对远程其他节点的BlockManager的数据进行读写；

每个BlockManager创建之后，做的第一件事就是想BlockManagerMaster进行注册，此时BlockManagerMaster会为其创建对应的BlockManagerInfo。

使用BlockManager进行写操作时，比如说，RDD运行过程中的一些中间数据，或者我们手动指定了persist()，会优先将数据写入内存中，如果内存大小不够，会使用自己的算法，将内存中的部分数据写入磁盘；此外，如果persist()指定了要replica，那么会使用BlockTransferService将数据replicate一份到其他节点的BlockManager上去。

使用BlockManager进行读操作时，比如说，shuffleRead操作，如果能从本地读取，就利用DiskStore或者MemoryStore从本地读取数据，但是本地没有数据的话，那么会用BlockTransferService与有数据的BlockManager建立连接，然后用BlockTransferService从远程BlockManager读取数据；例如，shuffle Read操作中，很有可能要拉取的数据在本地没有，那么此时就会到远程有数据的节点上，找那个节点的BlockManager来拉取需要的数据。

只要使用BlockManager执行了数据增删改的操作，那么必须将Block的BlockStatus上报到BlockManagerMaster，在BlockManagerMaster上会对指定BlockManager的BlockManagerInfo内部的BlockStatus进行增删改操作，从而达到元数据的维护功能。

## 8.2 Spark 共享变量底层实现

Spark一个非常重要的特性就是共享变量。

默认情况下，如果在一个算子的函数中使用到了某个外部的变量，那么这个变量的值会被拷贝到每个task中，此时每个task只能操作自己的那份变量副本。如果多个task想要共享某个变量，那么这种方式是做不到的。

Spark为此提供了两种共享变量，一种是Broadcast Variable（广播变量），另一种是Accumulator（累加变量）。Broadcast Variable会将用到的变量，仅仅为每个节点拷贝一份，即每个Executor拷贝一份，更大的用途是优化性能，减少网络传输以及内存损耗。Accumulator则可以让多个task共同操作一份变量，主要可以进行累加操作。Broadcast Variable是共享读变量，task不能去修改它，而Accumulator可以让多个task操作一个变量。

8.2.1 广播变量

广播变量允许编程者在每个Executor上保留外部数据的只读变量，而不是给每个任务发送一个副本。

每个task都会保存一份它所使用的外部变量的副本，当一个Executor上的多个task都使用一个大型外部变量时，对于Executor内存的消耗是非常大的，因此，我们可以将大型外部变量封装为广播变量，此时一个Executor保存一个变量副本，此Executor上的所有task共用此变量，不再是一个task单独保存一个副本，这在一定程度上降低了Spark任务的内存占用。

图8-2 task使用外部变量

图8-3 使用广播变量

Spark还尝试使用高效的广播算法分发广播变量，以降低通信成本。

Spark提供的Broadcast Variable是只读的，并且在每个Executor上只会有一个副本，而不会为每个task都拷贝一份副本，因此，它的最大作用，就是减少变量到各个节点的网络传输消耗，以及在各个节点上的内存消耗。此外，Spark内部也使用了高效的广播算法来减少网络消耗。

可以通过调用SparkContext的broadcast()方法来针对每个变量创建广播变量。然后在算子的函数内，使用到广播变量时，每个Executor只会拷贝一份副本了，每个task可以使用广播变量的value()方法获取值。

在任务运行时，Executor并不获取广播变量，当task执行到 使用广播变量的代码时，会向Executor的内存中请求广播变量，如下图所示：

图8-4 task向Executor请求广播变量

之后Executor会通过BlockManager向Driver拉取广播变量，然后提供给task进行使用，如下图所示：

图8-5 Executor从Driver拉取广播变量

广播大变量是Spark中常用的基础优化方法，通过减少内存占用实现任务执行性能的提升。

8.2.2 累加器

累加器（accumulator）：Accumulator是仅仅被相关操作累加的变量，因此可以在并行中被有效地支持。它们可用于实现计数器（如MapReduce）或总和计数。

Accumulator是存在于Driver端的，集群上运行的task进行Accumulator的累加，随后把值发到Driver端，在Driver端汇总（Spark UI在SparkContext创建时被创建，即在Driver端被创建，因此它可以读取Accumulator的数值），由于Accumulator存在于Driver端，从节点读取不到Accumulator的数值。

Spark提供的Accumulator主要用于多个节点对一个变量进行共享性的操作。Accumulator只提供了累加的功能，但是却给我们提供了多个task对于同一个变量并行操作的功能，但是task只能对Accumulator进行累加操作，不能读取它的值，只有Driver程序可以读取Accumulator的值。

Accumulator的底层原理如下图所示：

图8-6 累加器原理

# 9. 总结

Spark的内核原理对于更好的使用Spark完成开发任务有着非常重要的作用，同时Spark的内核知识也是面试过程中经常被问到的知识点。

在本课程的学习中，我们对Spark的部署模式、通信架构、任务调度机制、Shuffle过程、内存管理机制以及Spark的核心组件进行了详细分析，这些内容都是Spark最为重要的架构原理，希望在之后的学习中大家可以不断深化对于Spark内核架构的理解，在更高的层次上去使用Spark技术框架。