大数据科学丛书
BIG DATA SCIENCE

Spark

内核机制解析及性能调优

王家林　段智华　等编著

机械工业出版社
CHINA MACHINE PRESS

大数据科学丛书

Spark 内核机制解析及性能调优

王家林　段智华　等编著

机 械 工 业 出 版 社

Spark 建立在抽象的 RDD 之上，要理解 Spark，就需要理解 RDD。本书详细讲述了 RDD 的功能及内部实现的机制，通过剖析源代码引导读者深入理解 Spark 集群部署的内部机制、Spark 内部调度机制、Executor 的内部机制和 Shuffle 的内部机制，进而讲述了“钨丝计划”的内部机制。本书的最后一章是任何 Spark 应用者都非常关注的 Spark 性能调优内容。通过学习本书，可以使读者对 Spark 内核有更加深入的理解，从而实现对 Spark 系统深度调优、Spark 生产环境下故障的定位和排除，以及 Spark 的二次开发和系统高级运维。

本书适合于对大数据开发有兴趣的在校学生。同时，对于有分布式计算框架应用经验的人员，本书也可以作为 Spark 源代码解析的参考书籍。

图书在版编目（CIP）数据

Spark 内核机制解析及性能调优/王家林等编著. —北京：机械工业出版社，2016.10
（大数据科学丛书）
ISBN 978-7-111-55442-4

Ⅰ.①S… Ⅱ.①王… Ⅲ.①数据处理软件 Ⅳ.①TP274

中国版本图书馆 CIP 数据核字（2016）第 278303 号

机械工业出版社（北京市百万庄大街 22 号 邮政编码 100037）
责任编辑：王 斌
责任校对：张艳霞
责任印制：李 飞
北京铭成印刷有限公司印刷
2017 年 1 月第 1 版·第 1 次印刷
184mm×260mm·22 印张·537 千字
0001-3000 册
标准书号：ISBN 978-7-111-55442-4
定价：59.00 元

凡购本书，如有缺页、倒页、脱页，由本社发行部调换

电话服务
服务咨询热线：（010）88379833
读者购书热线：（010）88379649

网络服务
机 工 官 网：www.cmpbook.com
机 工 官 博：weibo.com/cmp1952
教育服务网：www.cmpedu.com
金 书 网：www.golden-book.com

封面无防伪标均为盗版

前　言

起源于美国加州大学伯克利分校 AMP 实验室的 Spark 是当今大数据领域最活跃、最热门的大数据计算处理框架，2009 年 Spark 诞生于 AMP 实验室，2010 年 Spark 正式成为开源项目，2013 年 Spark 成为 Apache 基金项目，2014 年 Spark 成为 Apache 基金的顶级项目。Spark 成功构建了一体化、多元化的大数据处理体系，成功使用 Spark SQL、Spark Streaming、MLlib、GraphX 解决了大数据领域的 Batch Processing、Stream Processing、Adhoc Query 等核心问题，Spark SQL、Spark Streaming、Mllib、GraphX 四个子框架和 Spark 核心库之间互相共享数据及相互操作，Spark 生态系统强大的集成能力是其他大数据平台无可匹敌的。

本书主要面向的对象是广大的 Spark 爱好者和大数据开发者，以 Spark 内核解析及性能调优为主导，由浅入深，对 Spark 内核运行机制从源代码角度加以详细解析，全书共分 9 章，分别是：RDD 的功能解析、RDD 的运行机制、部署模式（Deploy）解析、Spark 调度器（Scheduler）运行机制、执行器（Executor）、Spark 的存储模块（Storage）、Shuffle 机制、钨丝计划（Project Tungsten）以及性能优化。读者通过对这些内容的深入学习，将能够较为透彻地掌握 Spark 这一大数据计算框架的应用方法。

参与本书编写的有王家林、段智华、张敏等。

在本书阅读过程中，如发现任何纰漏或有任何疑问，可以加入本书的阅读群（QQ：284078981）提出问题，会有专人答疑。同时，该群也会提供本书所用案例源代码。

如果读者想要了解或者学习更多大数据相关技术，可以关注 DT 大数据梦工厂微信公众号 DT_Spark 及 QQ 群 284078981，或者扫描下方二维码咨询，也可以通过 YY 客户端登录 68917580 永久频道直接体验。

王家林老师的新浪微博是 http：//weibo. com/ilovepains/，欢迎大家在微博上与作者进行互动。

由于时间仓促，书中难免存在不妥之处，请读者谅解，并提出宝贵意见。

王家林 2016. 10. 8 日于深圳

目　录

第 1 章　RDD 功能解析

Spark 建立在抽象的 RDD（Resilient Distributed Datasets，弹性分布式数据集）之上，使得它可以用一致的方式处理大数据不同的应用场景。Spark 把所有需要处理的数据转化成为 RDD，然后对 RDD 进行一系列的算子运算，从而得到结果。RDD 是一个容错的、并行的数据结构，它可以将数据存储到内存和磁盘中，并能控制数据分区，且提供了丰富的 API 来操作数据。Spark 一体化多元化的解决方案极大地减少了开发和维护的人力成本，以及部署平台的物力成本，并在性能方面有极大的优势，特别适合于迭代计算，如机器学习和图计算；同时 Spark 对 Scala 和 Python 交互式 Shell 的支持也极大地方便了通过 Shell 直接使用 Spark 集群来验证解决问题的方法，这对于原型开发至关重要，对数据分析人员有着无法抗拒的吸引力。本章将详细介绍 RDD 的功能。

Section 1.1 RDD 产生的技术背景及功能

Hadoop 的 MapReduce 是一种基于数据集的工作模式，这种模式的工作方式是：从物理存储上加载数据，然后操作数据，最后写入物理存储设备。

基于数据集操作的系统对两种应用的处理并不高效：一是迭代式的算法，这在图应用和机器学习领域很常见，二是交互式数据挖掘工具（反复查询一个数据子集）。这两种情况下，将数据保存在内存中能够极大地提高性能。基于数据集的方式不能够复用曾经的计算结果或中间计算结果，Hadoop 每次作业都从磁盘上读写数据，而且第二次作业运行时会再次从磁盘上读写数据，不能基于内存共享数据，这种数据集系统对通用的应用的操作处理并不高效，因为通用的处理一般都是用迭代的（如机器学习和图计算），另外假设要对处理的结果进行复用的话，例如，想要复用图 1-1 中 Job#1 的结果，这时候必须在磁盘上重读，不能基于内存复用，只能再次执行 Job#1，不会从内存中复用上次的结果。基于数据

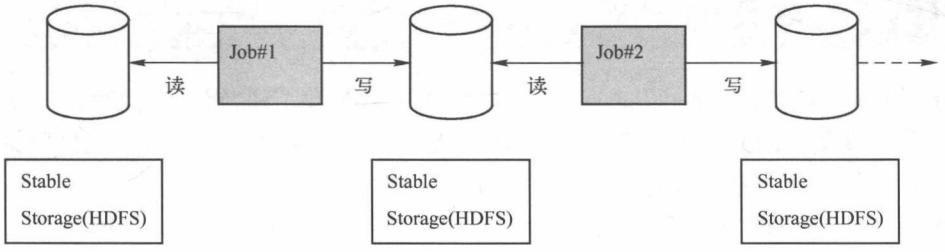

图 1-1　工作模型图

集的系统对交互式数据挖掘工具也不太适用。如果能将数据保存到内存中，其性能将会得到大大提升。

数据集的特性有位置感知、容错性和负载均衡等。在众多特性中，最难实现的是容错性。一般来说，分布式数据集的容错性有两种方式：数据检查点和记录数据更新。当面向大规模数据分析时，数据检查点操作成本很高：需要通过数据中心的网络链接在各台机器之间复制庞大的数据集，而网络带宽往往比内存带宽低得多，同时还需要消耗更多的存储资源（在内存中复制可以减少需要缓存的数据量，而存储到磁盘则会拖慢应用程序）。

这里所讲的 RDD 就是在这种背景下所产生的。Spark 的 RDD 是一种基于工作集的工作模式。无论数据集还是工作集，它们都有一些共同的特征，如位置感知、容错和负载均衡等，为了有效地实现容错，RDD 本身提供了一种高度受限的共享内存模型，它是只读的记录分区的集合，只能够通过从外界读取文件，或者说由其他的 RDD 来产生。RDD 的读操作可以精确到一条记录，RDD 的写操作则是批量的。RDD 的模型特别适合迭代，因为后面的 RDD 都是依据前面的 RDD 产生的，或者从外部读取数据而产生的，这就有一种前后的依赖关系，创建 RDD 的一系列转换被记录下来（即 Lineage 机制），以便恢复失去的数据。

Spark 的 RDD 为基于工作集的应用提供了更为通用的抽象，用户可以对中间结果进行显式的命名和物化，控制其分区，还能执行用户选择的特定操作（而不是在运行时去循环执行一系列 MapReduce 步骤），Spark 为工作集的应用提供了基本的抽象，同时又具有以 Hadoop 为代表的数据流模型的优势（如自动位置的感知、自动容错、伸缩性和良好的调度等），Spark 的 RDD 之间具有依赖关系且高度抽象，编程模型更容易，容错更好。一些算法，如逻辑回归、k 幂次数列等，都会在多个作业中重用或共用计算结果，在多个共享数据集中交互式查询。RDD 本身采用分布式内存计算的抽象容错机制解决了多步骤迭代，在一个共享数据集中执行多个交互式查询（多个 Job 中重用计算数据），这是 RDD 的使用场景。RDD 不太适合那些异步更新共享状态的应用，如并行 Web 爬虫。RDD 的目标是为大多数分析型应用提供有效的编程模型。

Section 1.2 RDD 的基本概念

1.2.1 RDD 的定义

RDD（Resilient Distributed Datasets，弹性分布式数据集）是分布式内存的一个抽象概念，是一种高度受限的共享内存模型，即 RDD 是只读的记录分区的集合，能横跨集群的所有结点进行并行计算，是一种基于工作集的应用抽象。

RDD 底层存储原理为：其数据分布存储于多台机器上，事实上，每个 RDD 的数据都以 Block 的形式存储在多台机器上。图 1-2 所示是 Spark 的 RDD 存储架构图，其中每个 Executor 会启动一个 BlockManagerSlave，并管理一部分 Block；而 Block 的元数据由 Driver 结点上的

BlockManagerMaster 保存，BlockManagerSlave 生成 Block 后向 BlockManagerMaster 注册该 Block，BlockManagerMaster 管理 RDD 与 Block 的关系，当 RDD 不再需要存储时，将向 BlockManagerSlave 发送指令删除相应的 Block。

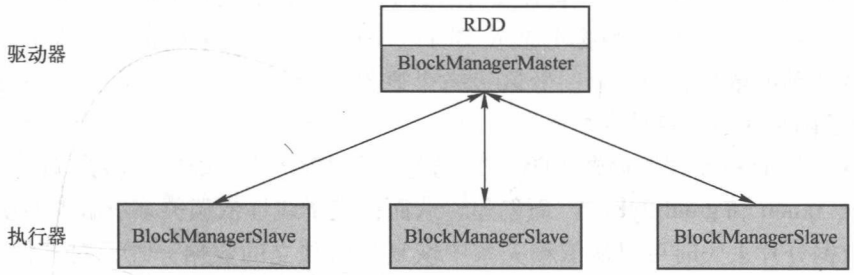

图 1-2 RDD 存储架构图

BlockManager 管理 RDD 的物理分区，每个 Block 就是结点上对应的一个数据块，可以存储在内存或者磁盘上。而 RDD 中的 Partition 是一个逻辑数据块，对应相应的物理块 Block。本质上，一个 RDD 在代码中相当于是数据的一个元数据结构，存储着数据分区及其逻辑结构的映射关系，存储着 RDD 之前的依赖转换关系。

在此需要对 BlockManager 进行简要的解释，BlockManager 的部分源代码如下。

```
1.  /**
2.   * 在每个结点上运行管理 Block(Driver 和 Executors),它提供一个接口来检索本地和远程
       的存储变量
3.   * 如 memory、disk 和 off - heap
4.   * 请注意,在使用 BlockManager 之前必须要先 initialize()
5.   */
6.  private[spark] class BlockManager(
7.      executorId:String,
8.      rpcEnv:RpcEnv,
9.      //负责对各个结点的 BlockManager 内部管理的数据的元数据进行管理与维护
10.     val master:BlockManagerMaster,
11.     defaultSerializer:Serializer,
12.     val conf:SparkConf,
13.     memoryManager:MemoryManager,
14.     mapOutputTracker:MapOutputTracker,
15.     shuffleManager:ShuffleManager,
16.     blockTransferService:BlockTransferService,
17.     securityManager:SecurityManager,
18.     numUsableCores:Int)
19.   extendsBlockDataManager with Logging
```

BlockManagerMaster 会持有整个 Application 的 Block 的位置、Block 所占用的存储空间等元数据信息，在 Spark 的 Driver 的 DAGScheduler 中就是通过这些信息来确认数据运行的本地

性的。Spark 支持重分区，数据通过 Spark 默认的或者用户自定义的分区器来决定数据块分布在哪些结点。RDD 的物理分区是由 Block - Manager 管理的，每个 Block 就是结点上对应的一个数据块，可以存储在内存或者磁盘中。而 RDD 中的 Partition 是一个逻辑数据块，对应相应的物理块 Block。本质上，一个 RDD 在代码中相当于是数据的一个元数据结构（一个 RDD 就是一组分区），存储着数据分区及 Block、Node 等的映射关系以及其他元数据信息，存储着 RDD 之前的依赖转换关系。分区是一个逻辑概念，Transformation 前后的新旧分区在物理上可能是同一块内存存储。

Spark 通过读取外部数据创建 RDD，或通过其他 RDD 执行确定的转换 Transformation 操作（如 map、union 和 groubByKey）而创建，从而构成了线性依赖关系或者说血统关系（lineage），在数据分片丢失时可以从依赖关系中恢复自己独立的数据分片，对其他数据分片或计算机没有影响，基本没有检查点开销，使得实现容错的开销很低，失效时只需要重新计算那些 RDD 分区，就可以在不同结点上并行执行，而不需要回滚（Roll Back）整个程序。关于落后任务（即运行很慢的结点）是通过任务备份，重新调用执行进行处理的。

因为 RDD 本身支持基于工作集的运用，所以可以使 Spark 的 RDD 持久化（Persist）到内存中，在并行计算中高效重用。在进行多个查询时，就可以显性地将工作集中的数据缓存到内存中，为后续查询提供复用，这极大地提升了查询的速度。在 Spark 中，一个 RDD 就是一个分布式对象集合，每个 RDD 可分为多个片（Partitions），而分片可以在集群环境的不同结点上计算。

RDD 作为泛型的抽象的数据结构（如下面源代码的第 8 行代码），支持两种计算操作算子：Transformation（变换）与 Action（行动）。并且 RDD 的写操作是粗粒度的，读操作既可以是粗粒度的也可以是细粒度的。

```
1.  /** 每个 RDD 都有 5 个主要特性 *
2.    *    - 分区列表
3.    *    - 每个分区都有一个计算函数
4.    *    - 依赖于其他 RDD 的列表
5.    *    - 数据类型(Key - Value)的 RDD 分区器
6.    *    - 每个分区都有一个分区位置列表
7.    */
8.  abstract class RDD[T:ClassTag](
9.        @transient private var _sc:SparkContext,
10.       @transient private vardeps:Seq[Dependency[_]]
11.    )extendsSerializable with Logging
```

在此需要对 SparkContext 做出解释：SparkContext 是 Spark 功能的主要入口点，一个 SparkContext 代表一个集群连接，可以用其在集群中创建 RDD、累加变量和广播变量等，在每一个可用的 JVM 中只有一个 SparkContext，在创建一个新的 SparkContext 之前必须先停止该 JVM 中可用的 SparkContext，这种限制可能最终会被修改。SparkContext 被实例化时需要一个 SparkConf 对象去描述应用的配置信息，在这个配置对象中设置的信息会覆盖系统默认的配置。

1.2.2 RDD 五大特性

1. 分区列表（a list of partitions）

Spark RDD 是被分区的，每一个分区都会被一个计算任务（Task）处理，分区数决定了并行计算的数量，RDD 的并行度默认从父 RDD 传给子 RDD。默认情况下，一个 HDFS 上的数据分片就是一个 partiton，RDD 分片数决定了并行计算的力度，可以在创建 RDD 时指定 RDD 分片个数，如果不指定分区数量，当 RDD 从集合创建时，则默认分区数量为该程序所分配到的资源的 CPU 核数（每个 Core 可以承载 2 ～ 4 个 partition），如果是从 HDFS 文件创建，默认为文件的 Block 数。

2. 每一个分区都有一个计算函数（a function for computing each split）

每个分区都会有计算函数，Spark 的 RDD 的计算函数是以分片为基本单位的，每个 RDD 都会实现 compute 函数，对具体的分片进行计算，RDD 中的分片是并行的，所以是分布式并行计算，有一点非常重要，就是由于 RDD 有前后依赖关系，遇到宽依赖关系，如 reduceByKey 等这些操作时划分成 Stage，Stage 内部的操作都是通过 Pipeline 进行的，在具体处理数据时它会通过 BlockManager 来获取相关的数据，因为具体的 split 要从外界读数据，也要把具体的计算结果写入外界，所以用了一个管理器，具体的 split 都会映射成 BlockManager 的 Block，而具体的 split 会被函数处理，函数处理的具体形式是以任务的形式进行的。

3. 依赖于其他 RDD 的列表（a list of dependencies on other RDDs）

由于 RDD 每次转换都会生成新的 RDD，所以 RDD 会形成类似流水线一样的前后依赖关系，当然宽依赖就不类似于流水线了，宽依赖后面的 RDD 具体的数据分片会依赖前面所有的 RDD 的所有数据分片，这个时候数据分片就不进行内存中的 Pipeline，一般都是跨机器的，因为有前后的依赖关系，所以当有分区的数据丢失时，Spark 会通过依赖关系进行重新计算，从而计算出丢失的数据，而不是对 RDD 所有的分区进行重新计算。RDD 之间的依赖有两种：窄依赖（Narrow Dependency）和宽依赖（Wide Dependency）。RDD 是 Spark 的核心数据结构，通过 RDD 的依赖关系形成调度关系。通过对 RDD 的操作形成整个 Spark 程序。

RDD 有窄依赖和宽依赖两种不同类型的依赖，其中的窄依赖指的是每一个 parent RDD 的 Partition 最多被 child RDD 的一个 Partition 所使用，而宽依赖指的是多个 child RDD 的 Partition 会依赖于同一个 parent RDD 的 Partition。可以从两个方面来理解 RDD 之间的依赖关系，一方面是 RDD 的 parent RDD 是什么，另一方面是依赖于 parent RDD 的哪些 Partion；根据依赖于 parent RDD 的哪些 Partion 的不同情况，Spark 将 Dependency 分为宽依赖和窄依赖两种。Spark 中的宽依赖指的是生成的 RDD 的每一个 partition 都依赖于父 RDD 所有的 partition，宽依赖典型的操作有 groupByKey、sortByKey 等，宽依赖意味着 Shuffle 操作，这是 Spark 划分 Stage 的边界的依据，Spark 中的宽依赖支持两种 Shuffle Manager，即 HashShuffleManager 和 SortShuffleManager，前者是基于 Hash 的 Shuffle 机制，后者是基于排序的 Shuffle 机制。

4. key – value 数据类型的 RDD 分区器（a Partitioner for key – value RDDS）、控制分区策略和分区数

每个 key – value 形式的 RDD 都有 Partitoner 属性，它决定了 RDD 如何分区。当然，Partiton 的个数还决定了每个 Stage 的 Task 个数。RDD 的分片函数可以分区（Partitioner），可传入相关的参数，如 HashPartitioner 和 RangePartitioner，它本身针对 key – value 的形式，如果不是 key – value 的形式它就不会有具体的 Partitioner，Partitioner 本身决定了下一步会产生多少并行的分片，同时它本身也决定了当前并行（Parallelize）Shuflle 输出的并行数据，从而使 Spark 具有能够控制数据在不同结点上分区的特性，用户可以自定义分区策略，如 Hash 分区等。Spark 提供了 partitionBy 运算符，能通过集群对 RDD 进行数据再分配来创建一个新的 RDD。

5. 每个分区都有一个优先位置列表（a list of preferred locations to compute each split on）

优先位置列表会存储每个 Partition 的优先位置，对于一个 HDFS 文件来说，就是每个 Partition 块的位置。观察运行 Spark 集群的控制台就会发现，Spark 在具体计算、具体分片以前，它已经清楚地知道任务发生在哪个结点上，也就是说任务本身是计算层面的、代码层面的，代码发生运算之前它就已经知道它要运算的数据在什么地方，有具体结点的信息。这就符合大数据中数据不动代码动的原则。数据不动代码动的最高境界是数据就在当前结点的内存中。这时候有可能是 Memory 级别或 Tachyon 级别的，Spark 本身在进行任务调度时会尽可能地将任务分配到处理数据的数据块所在的具体位置。据 Spark 的 RDD. Scala 源代码函数 getParferredLocations 可知，每次计算都符合完美的数据本地性。

可在 RDD 类源代码文件中找到 4 个方法和 1 个属性，对应上述所阐述的 RDD 的五大特性，源代码剪辑如下。

```
1.  /** 返回一个 RDD 分区列表,这个方法仅被调用一次,它是安全地执行一次耗时计算 */
2.  protected def getPartitions:Array[Partition]
3.  /** 通过子类来实现给定分区的计算 */
4.    @DeveloperApi
5.  def compute(split:Partition,context:TaskContext):Iterator[T]
6.  /** 返回对父 RDD 的依赖列表,这个方法仅被调用一次,它是安全地执行一次耗时计算 */
7.  protected defgetDependencies:Seq[Dependency[_]] = deps
8.  /** 可选的,分区的方法,可指定如何分区 */
9.  @transient val partitioner:Option[Partitioner] = None
10. /** 可选的,指定优先位置,输入参数是 split 分片,输出结果是一组优先的结点位置 */
11. protected def getPreferredLocations(split:Partition):Seq[String] = Nil
```

在此需要对 TaskContext、Partitioner 和 Partition 等概念做出解释，TaskContext 是读取或改变执行任务的环境，用 org. apache. spark. TaskContext. get( ) 可返回当前可用的 TaskContext，可以调用内部的函数访问正在运行任务的环境信息。Partitioner 是一个对象，定义了如何在 key – value 类型的 RDD 的元素中用 key 分区，从 0 到 numPartitions – 1 区间内映射每一个 key 到 partition ID。Partition 是在一个 RDD 的分区标识符，源代码如下。

```
1.  trait Partition extendsSerializable {
2.    //在它的父 RDD 的分区索引
3.    def index:Int
4.    //最好默认实现 HashCode
5.    override defhashCode( ):Int = index
6.  }
```

1.2.3 RDD 弹性特性的 7 个方面

RDD 作为弹性分布式数据集，它的弹性具体体现在以下 7 个方面。

1. 自动进行内存和磁盘数据存储的切换

Spark 会优先把数据放到内存中，如果内存实在放不下，则会放到磁盘里面，不但能计算内存放下的数据，也能计算内存放不下的数据。如果实际数据大于内存，则要考虑数据放置策略和优化算法。当应用程序内存不足时，Spark 应用程序将数据自动从内存存储切换到磁盘存储，以保障其高效运行。

2. 基于 Lineage（血统）的高效容错机制

Lineage 是基于 Spark RDD 的依赖关系来完成的（依赖分为窄依赖和宽依赖两种形态），这种前后的依赖关系可恢复失去的数据。每个操作只关联其父操作，各个分片的数据之间互不影响，出现错误时只需恢复单个 Split 的特定部分即可。常规容错有两种方式：一个是数据检查点，另一个是记录数据的更新。数据检查点的基本工作方式就是通过数据中心的网络链接不同的机器，然后每次操作时都要复制数据集，就相当于每次都有一个拷贝，拷贝是要通过网络的，网络带宽就是分布式的瓶颈，对存储资源也是很大的消耗。记录数据更新就是每次数据变化了就记录一下，这种方式不需要重新复制一份数据，但是比较复杂，会消耗性能。Spark 的 RDD 通过记录数据更新的方式很高效，原因为：一是 RDD 是不可变的且是 Lazy 级别；二是 RDD 的写操作是粗粒度的。但是 RDD 的读操作既可以是粗粒度的也可以是细粒度的。

3. Task 如果失败会自动进行特定次数的重试

默认重试次数为 4 次。源代码如下。

```
1.  private[ spark] class TaskSchedulerImpl(
2.       val sc:SparkContext,
3.       valmaxTaskFailures:Int,
4.       isLocal:Boolean = false)
5.    extends TaskScheduler with Logging //继承任务调度器、日志 trait
6.  {
7.    def this(sc:SparkContext) = this(sc,sc. conf. getInt("spark. task. maxFailures",4))
8.    …
9.  }
```

TaskSchedulerImpl 是底层的任务调度接口 TaskScheduler 的实现，这些 Schedulers 从每一个 Stage 中的 DAGScheduler 中获取 TaskSet，运行它们。DAGScheduler 是高层调度，它计算每个 Job 的 Stages 的 DAG，然后提交给 Stages，用 TaskSets 的形式启动底层 TaskScheduler 调度在集群中运行。

4. Stage 如果失败会自动进特定次数的重试

这样 Stage 对象可以跟踪多个 StageInfo（存储 SparkListeners 监听到的 Stage 信息，将 Stage 信息传递给 Listeners 或 web UI。默认重试次数为 4 次，且可以直接运行计算失败的阶段，只计算失败的数据分片，Stage 源代码如下。

```
1.  private[scheduler] abstract class Stage(
2.      val id:Int,
3.      val rdd:RDD[_],
4.      val numTasks:Int,
5.      val parents:List[Stage],
6.      val firstJobId:Int,
7.      val callSite:CallSite)
8.    extends Logging {
9.    //Partition 的个数
10.   val numPartitions = rdd. partitions. length
11.   //属于这个工作集的 Stage
12.   val jobIds = new HashSet[Int]
13.   val pendingPartitions = new HashSet[Int]
14.   //用于此 Stage 的下一个新 Attempt 的标识 ID
15.   private var nextAttemptId:Int = 0
16.   val name:String = callSite. shortForm
17.   val details:String = callSite. longForm
18.   private var _internalAccumulators:Seq[Accumulator[Long]] = Seq. empty
19.   //Stage 内部所有任务共享的累加器
20.   def internalAccumulators:Seq[Accumulator[Long]] = _internalAccumulators
21.   /**
22.    * 重新初始化与该 Stage 相关联的内部累加器
23.    * 当属于这个 Stage 的任务的一个子集已经完成时,称为一次提交,否则
24.  * 重新初始化内部累加器,这里又将覆盖部分任务
25.  */
26.   def resetInternalAccumulators():Unit = {
27.     _internalAccumulators = InternalAccumulator. create(rdd. sparkContext)
28.   }
29.   /**
30.  * 最新的[StageInfo] object 指针。这需要在这里被初始化,
31.    * 任何 Attempts 都是被创造出来的,因为 DAGScheduler 使用 StageInfo
32.    * 告诉 SparkListeners 工作开始时(即发生之前的任何阶段已经创建)
```