许鹏：从零开始学习，Apache Spark源码走读（一）

发表于 2014-05-29 16:35 | 38047次阅读 | 来源 个人博客 | 60 条评论 | 作者 许鹏 大数据 Spark RDD 开源 Hadoop

摘要：自2013年6月进入Apache孵化器，Spark已经有来自25个组织的120多位开发者参与贡献。而在不久前，更成为了Apache软件基金会的顶级项目，当下已是知名Hadoop开发商Cloudera和MapR的新宠。

Spark是发源于美国加州大学伯克利分校AMPLab的集群计算平台，它立足于内存计算，性能超过Hadoop百倍，即使使用磁盘，迭代类型的计算也会有10倍速度的提升。Spark从多迭代批量处理出发，兼收并蓄数据仓库、流处理和图计算等多种计算范式，是罕见的全能选手。Spark当下已成为Apache基金会的顶级开源项目，拥有着庞大的社区支持——活跃开发者人数已超过Hadoop MapReduce）。这里，我们为大家分享许鹏的“Apache Spark源码走读”系列博文，从源码方面对这个流行大数据计算框架进行深度了解。

关于博主：许鹏，花名@徽沪一郎，**2000**年毕业于南京邮电学院，现就业于爱立信上海，在**UDM**部门从事相关产品研发，个人关注于**Linux** 内核及实时计算框架如**Storm**、**Spark**等。

以下为博文

楔子

源码阅读是一件非常容易的事，也是一件非常难的事。容易的是代码就在那里，一打开就可以看到。难的是要通过代码明白作者当初为什么要这样设计，设计之初要解决的主要问题是什么。

在对Spark的源码进行具体的走读之前，如果想要快速对Spark的有一个整体性的认识，阅读Matei Zaharia做的Spark论文是一个非常不错的选择。

在阅读该论文的基础之上，再结合Spark作者在2012 Developer Meetup上做的演讲Introduction to Spark Internals，那么对于Spark的内部实现会有一个比较大概的了解。

有了上述的两篇文章奠定基础之后，再来进行源码阅读，那么就会知道分析的重点及难点。

基本概念（**Basic Concepts**）

1. RDD——Resillient Distributed Dataset 弹性分布式数据集。

2. Operation——作用于RDD的各种操作分为transformation和action。

3. Job——作业，一个JOB包含多个RDD及作用于相应RDD上的各种operation。

4. Stage——一个作业分为多个阶段。

5. Partition——数据分区， 一个RDD中的数据可以分成多个不同的区。

6. DAG——Directed Acycle graph，有向无环图，反应RDD之间的依赖关系。

7. Narrow dependency——窄依赖，子RDD依赖于父RDD中固定的data partition。

8. Wide Dependency——宽依赖，子RDD对父RDD中的所有data partition都有依赖。

9. Caching Managenment——缓存管理，对RDD的中间计算结果进行缓存管理以加快整体的处理速度。

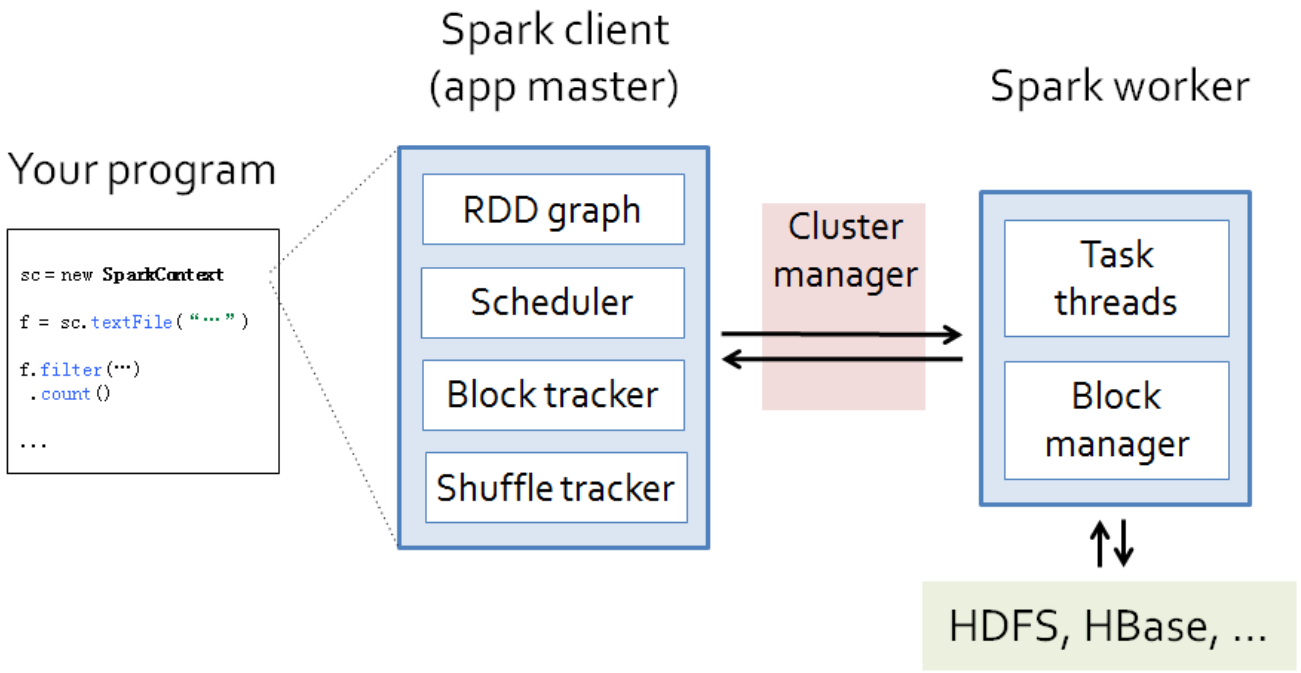
编程模型（**Programming Model**）

RDD是只读的数据分区集合，注意是数据集。

作用于RDD上的Operation分为transformantion和action。 经Transformation处理之后，数据集中的内容会发生更改，由数据集A转换成为数据集B；而经Action处理之后，数据集中的内容会被归约为一个具体的数值。

只有当RDD上有action时，该RDD及其父RDD上的所有operation才会被提交到cluster中真正的被执行。

从代码到动态运行，涉及到的组件如下图所示。



演示代码

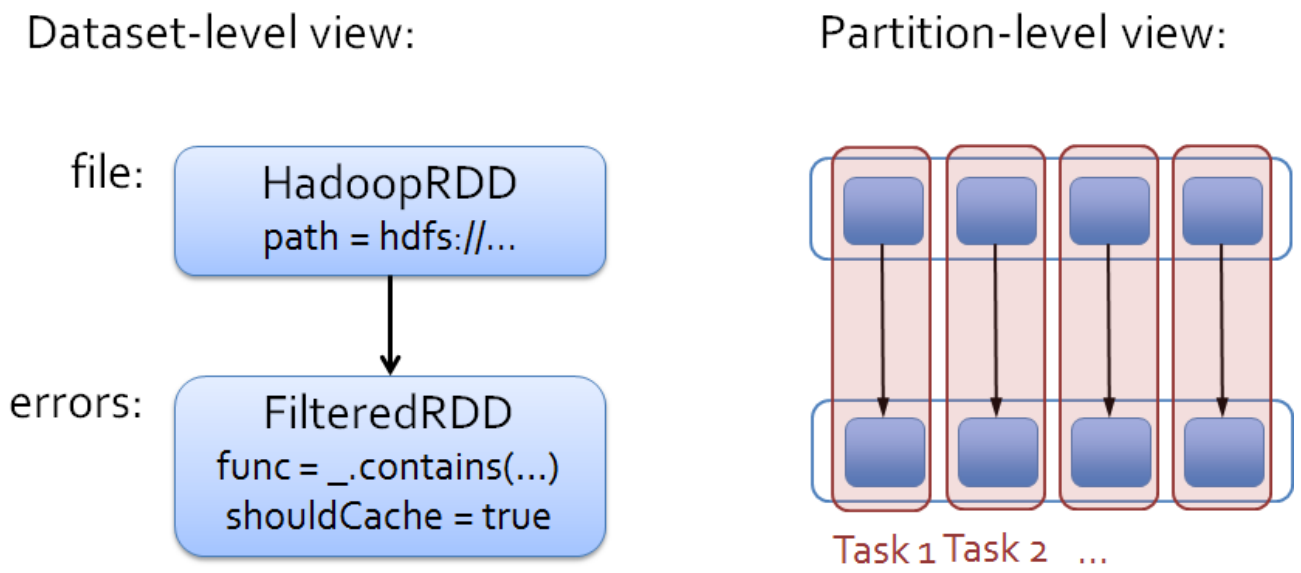
```
val sc = new SparkContext("Spark://...", "MyJob", home, jars)
val file = sc.textFile("hdfs://...")
```

```
val errors = file.filter(_.contains("ERROR"))
errors.cache()
errors.count()
```

运行态（**Runtime view**）

不管什么样的静态模型，其在动态运行的时候无外乎由进程，线程组成。

用Spark的术语来说，static view称为dataset view，而dynamic view称为parition view。关系如图所示



在Spark中的task可以对应于线程，worker是一个个的进程，worker由driver来进行管理。

那么问题来了，这一个个的task是如何从RDD演变过来的呢？下节将详细回答这个问题。

部署（**Deployment view**）

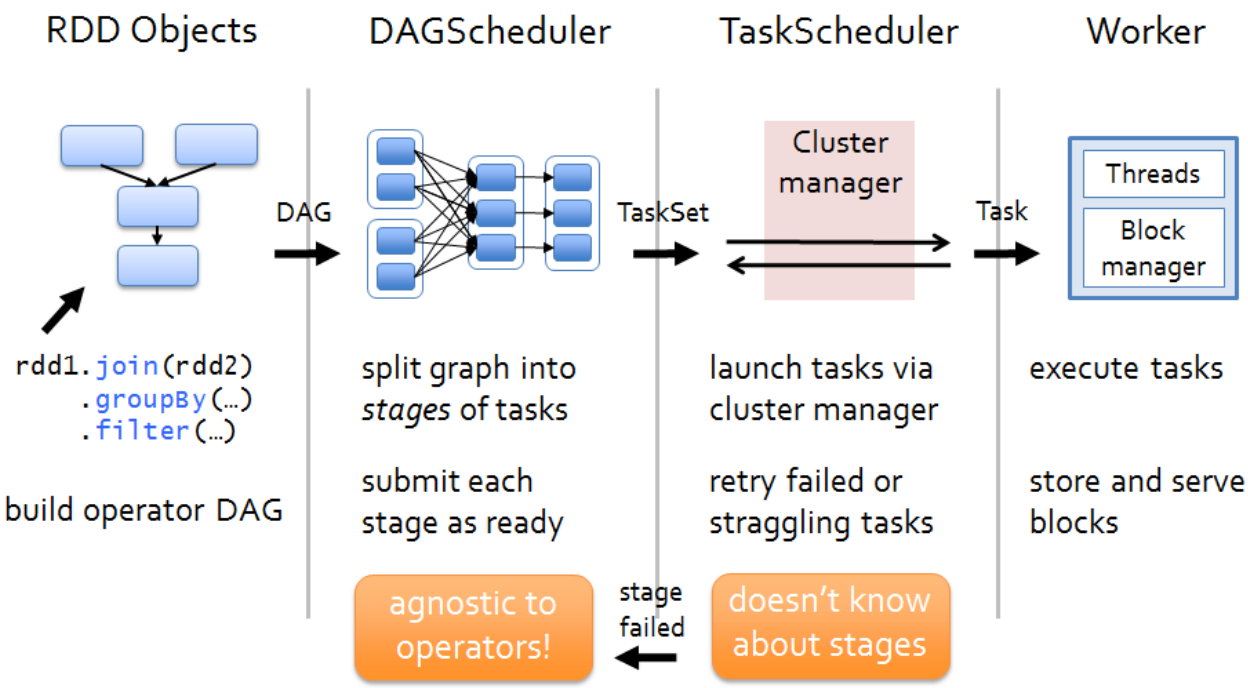
当有Action作用于某RDD时，该action会作为一个job被提交。

在提交的过程中，DAGScheduler模块介入运算，计算RDD之间的依赖关系。RDD之间的依赖关系就形成了DAG。

每一个JOB被分为多个stage，划分stage的一个主要依据是当前计算因子的输入是否是确定的，如果是则将其分在同一个stage，避免多个stage之间的消息传递开销。

当stage被提交之后，由taskscheduler来根据stage来计算所需要的task，并将task提交到对应的worker。

Spark支持以下几种部署模式，Standalone、Mesos和YARN。这些部署模式将作为taskscheduler的初始化入参。



RDD接口（RDD Interface）

RDD由以下几个主要部分组成

1. partitions——partition集合，一个RDD中有多少data partition
2. dependencies——RDD依赖关系
3. compute(parition)——对于给定的数据集，需要作哪些计算
4. preferredLocations——对于data partition的位置偏好
5. partitioner——对于计算出来的数据结果如何分发

缓存机制（caching）

RDD的中间计算结果可以被缓存起来，缓存先选Memory，如果Memory不够的话，将会被写入到磁盘中。

根据LRU（last-recent update）来决定哪先内容继续保存在内存，哪些保存到磁盘。

容错性（Fault-tolerant）

从最初始的RDD到衍生出来的最后一个RDD，中间要经过一系列的处理。那么如何处理中间环节出现错误的场景呢？

Spark提供的解决方案是只对失效的data partition进行事件重演，而无须对整个数据全集进行事件重演，这样可以大大加快场景恢复的开销。

RDD又是如何知道自己的data partition的number该是多少？如果是HDFS文件，那么HDFS文件的block将会成为一个重要的计算依据。

集群管理（cluster management）

task运行在cluster之上，除了Spark自身提供的Standalone部署模式之外，Spark还内在支持Yarn和mesos。

Yarn来负责计算资源的调度和监控，根据监控结果来重启失效的task或者是重新distributed task一旦有新的node加入cluster的话。

这一部分的内容需要参Yarn的文档。

小结

在源码阅读时，需要重点把握以下两大主线。

- 静态***view*** 即 RDD，transformation and action
- 动态***view*** 即 ***life of a job***， 每一个job又分为多个stage，每一个stage中可以包含多个rdd及其transformation，这些stage又是如何映射成为task被distributed到cluster中

概要

本文以wordCount为例，详细说明Spark创建和运行job的过程，重点是在进程及线程的创建。

实验环境搭建

在进行后续操作前，确保下列条件已满足。

1. 下载spark binary 0.9.1
2. 安装scala
3. 安装sbt
4. 安装java

启动**spark-shell**

单机模式运行，即**local**模式

local模式运行非常简单，只要运行以下命令即可，假设当前目录是$SPARK_HOME

```
MASTER=local bin/spark-shell
```

"MASTER=local"就是表明当前运行在单机模式

local cluster方式运行

local cluster模式是一种伪cluster模式，在单机环境下模拟Standalone的集群，启动顺序分别如下

1. 启动master
2. 启动worker
3. 启动spark-shell

master

```
$SPARK_HOME/sbin/start-master.sh
```

注意运行时的输出，日志默认保存在$SPARK_HOME/logs目录。

master主要是运行类 **org.apache.spark.deploy.master.Master**，在8080端口启动监听，日志如下图所示

```
14/04/21 10:14:15 INFO Slf4jLogger: Slf4jLogger started
14/04/21 10:14:15 INFO Remoting: Starting remoting
14/04/21 10:14:15 INFO Remoting: Remoting started; listening on addresses :[akka.tcp://sparkMaster@localhost:7077]
14/04/21 10:14:15 INFO Master: Starting Spark master at spark://localhost:7077
14/04/21 10:14:15 INFO MasterWebUI: Started Master web UI at http://localhost:8080
14/04/21 10:14:15 INFO Master: I have been elected leader! New state: ALIVE
```

修改配置

1. 进入$SPARK_HOME/conf目录
2. 将spark-env.sh.template重命名为spark-env.sh
3. 修改spark-env.sh，添加如下内容

```
export SPARK_MASTER_IP=localhost
export SPARK_LOCAL_IP=localhost
```

运行worker

```
bin/spark-class org.apache.spark.deploy.worker.Worker spark://localhost:7077 -i 127.0.0.1
-c 1 -m 512M
<br>
```

worker启动完成，连接到master。打开maser的web ui可以看到连接上来的worker. Master WEb UI的监听地址是***http://localhost:8080***

启动spark-shell

```
MASTER=spark://localhost:7077 bin/spark-shell
```

如果一切顺利，将看到下面的提示信息。

```
Created spark context..
Spark context available as sc.
```

可以用浏览器打开localhost:4040来查看如下内容

1. stages
2. storage
3. environment
4. executors

wordcount

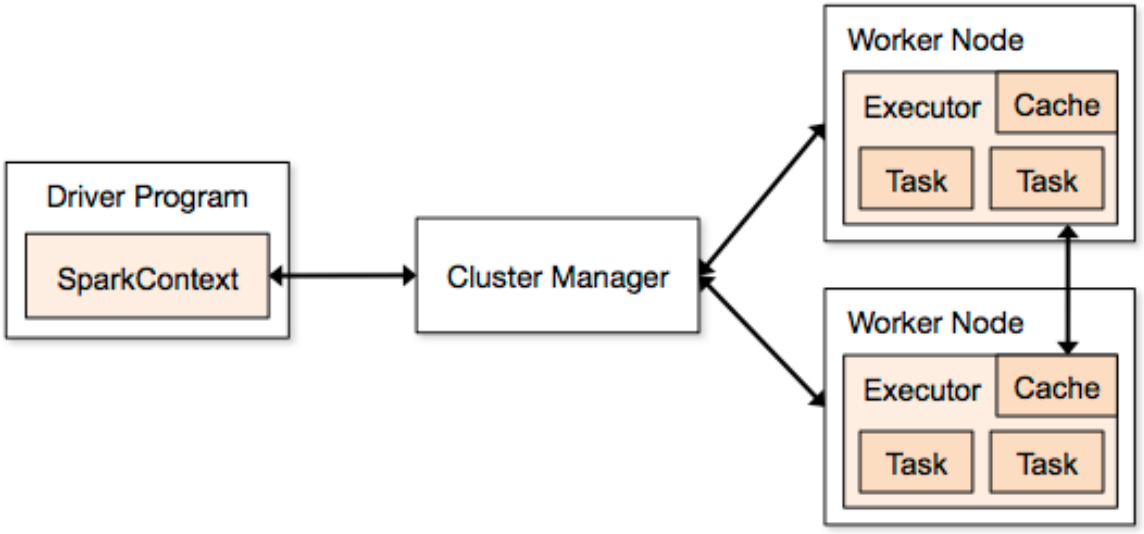
上述环境准备妥当之后，我们在sparkshell中运行一下最简单的例子，在spark-shell中输入如下代码

```
scala>sc.textFile("README.md").filter(_.contains("Spark")).count
```

上述代码统计在README.md中含有Spark的行数有多少

部署过程详解

Spark布置环境中组件构成如下图所示。



- **Driver Program** 简要来说在spark-shell中输入的wordcount语句对应于上图的Driver Program。
- **Cluster Manager** 就是对应于上面提到的master，主要起到deploy management的作用
- **Worker Node** 与Master相比，这是slave node。上面运行各个executor，executor可以对应于线程。executor处理两种基本的业务逻辑，一种就是driver programme，另一种就是job在提交之后拆分成各个stage，每个stage可以运行一到多个task

Notes: 在集群（cluster）方式下，Cluster Manager运行在一个**jvm**进程之中，而worker运行在另一个**jvm**进程中。在local cluster中，这些jvm进程都在同一台机器中，如果是真正的Standalone或Mesos及Yarn集群，worker与master或分布于不同的主机之上。

JOB的生成和运行

job生成的简单流程如下

1. 首先应用程序创建SparkContext的实例，如实例为sc
2. 利用SparkContext的实例来创建生成RDD
3. 经过一连串的transformation操作，原始的RDD转换成为其它类型的RDD
4. 当action作用于转换之后RDD时，会调用SparkContext的runJob方法
5. sc.runJob的调用是后面一连串反应的起点，关键性的跃变就发生在此处

调用路径大致如下

1. sc.runJob->dagScheduler.runJob->submitJob
2. DAGScheduler::submitJob会创建JobSummitted的event发送给内嵌类**eventProcessActor**
3. eventProcessActor在接收到JobSubmmitted之后调用**processEvent**处理函数

4. job到stage的转换，生成finalStage并提交运行，关键是调用**submitStage**
5. 在submitStage中会计算stage之间的依赖关系，依赖关系分为宽依赖和窄依赖两种
6. 如果计算中发现当前的stage没有任何依赖或者所有的依赖都已经准备完毕，则提交task
7. 提交task是调用函数**submitMissingTasks**来完成
8. task真正运行在哪个worker上面是由TaskScheduler来管理，也就是上面的submitMissingTasks会调用TaskScheduler::submitTasks
9. TaskSchedulerImpl中会根据Spark的当前运行模式来创建相应的backend，如果是在单机运行则创建LocalBackend
10. LocalBackend收到TaskSchedulerImpl传递进来的**ReceiveOffers**事件
11. receiveOffers->executor.launchTask->TaskRunner.run

代码片段executor.lauchTask

```
def launchTask(context: ExecutorBackend, taskId: Long, serializedTask: ByteBuffer) {
   val tr = new TaskRunner(context, taskId, serializedTask)
   runningTasks.put(taskId, tr)
   threadPool.execute(tr)
 }
```

说了这么一大通，也就是讲最终的逻辑处理切切实实是发生在TaskRunner这么一个executor之内。

运算结果是包装成为MapStatus然后通过一系列的内部消息传递，反馈到DAGScheduler，这一个消息传递路径不是过于复杂，有兴趣可以自行勾勒。