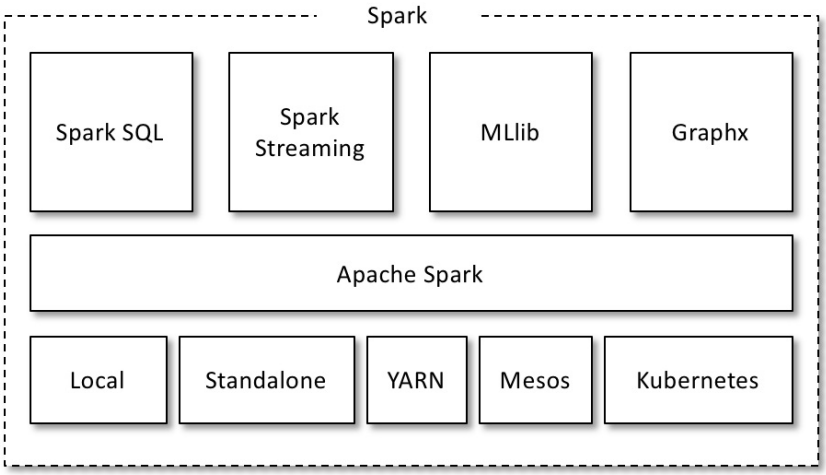
初识Spark

Spark 是一个用来实现快速且通用的集群计算的平台。Spark 是 UC Berkeley AMP Lab（加州大学伯克利分校的AMP实验室）所开源的类 MapReduce 的通用并行框架，现在已经是 Apache 的一个顶级项目。Spark 使用 Scala 语言开发，支持 Scala、Java、Python、R 语言相关的 API，运行于 JVM 之上。Spark 基于内存计算，提高了在大数据环境下数据处理的实时性，同时保证了高容错性和高可伸缩性。Spark 适用于各种各样原先需要多种不同的分布式平台实现的场景，包括批处理、迭代计算、交互式查询、流处理等。



如上图所示，Spark 生态圈即 BDAS（伯克利数据分析栈）包含的组件有 Spark Core、Spark Streaming、Spark SQL、MLlib 和 GraphX，它们都是由AMP实验室提供的，能够无缝地继承，并提供一站式解决平台。

Spark Core 实现了 Spark 的基本功能，包含任务调度、内存管理、错误恢复，以及与存储系统交互等模块。Spark Streaming 属于 Spark Core API 的扩展，支持实时数据流的可扩展、高吞吐、容错的流处理。Spark SQL 是 Spark 的一个结构化数据处理模块，提供了 DataFrame/Dataset 的编程抽象，可以看作一个分布式查询引擎。从 Spark 2.0 开始又引入了 Structured Streaming，它是建立

在 Spark SQL 之上的可扩展和高容错的流处理引擎。MLlib 是 Spark 提供的具有机器学习功能的程序库，它提供了很多种机器学习算法，包括分类、回归、聚类、协同过滤等，还提供了模型评估、数据导入等额外的功能。GraphX 是用来操作图的程序库，可以进行并行的图计算。

Spark 具有很强的适应性，能够使用 HDFS、Cassandra、HBase 等为持久层读写原生数据，资源管理采用 Mesos、YARN、Kubernetes 等集群资源管理模式，或者 Spark 自带的独立运行模式及本地运行模式。

Spark 具有一个庞大的生态圈，用于生产时还需要考虑参数调配、容错处理、监控、性能优化、存储、调度、部署等多个环节，涉及方方面面，仅以一个章节的内容是无法穷尽的。本章的主旨也并非简单地讲解 Spark，而是要讲解 Kafka 与 Spark 之间的集成细节。本章会以尽量少的篇幅让读者对 Spark 有一个初步的了解，并且会以合适的篇幅来讲解 Kafka 与 Spark Streaming 的集成，以及 Kafka 与 Structured Streaming 的集成。

Spark的安装及简单应用

下载 Spark 安装包是安装的第一步，下载地址 (http://spark.apache.org/downloads.html)。截至撰写之时，Spark 的最新版本为2.3.1，我们可以从官网中选择 spark-2.3.1-bin-hadoop2.7.tgz 进行下载。

下载完成后，先将安装包复制至/opt 目录下，然后执行相应的解压缩操作，示例如下：

```
[root@node1 opt]# tar zxvf spark-2.3.1-bin-
hadoop2.7.tgz
[root@node1 opt]# mv spark-2.3.1-bin-hadoop2.7
spark
[root@node1 opt]# cd spark
[root@node1 spark]#
```

解压缩之后可以直接运行 Spark，当然前提是要安装好 JDK，并设置好环境变量 JAVA_HOME。进入$SPARK_HOME/sbin 目录下执行 start-all.sh 脚本启动 Spark。脚本执行后，可以通过 jps -l 命令查看当前运行的进程信息，示例如下：

```
[root@node1 spark]# jps -l
23353 org.apache.spark.deploy.master.Master
23452 org.apache.spark.deploy.worker.Worker
```

可以看到 Spark 启动后多了 Master 和 Worker 进程，分别代表主节点和工作节点。我们还可以通过 Spark 提供的 Web 界面来查看 Spark 的运行情况，比如可以通过 http://localhost:8080 查看 Master 的运行情况。

Spark 中带有交互式的 shell，可以用作即时数据分析。现在我们通过 spark-shell 来运行一个简单但又非常经典的单词统计的程序，以便可以简单地了解 Spark 的使用。首先进入$SPARK_HOME/bin 目录下（SPARK_HOME 表示 Spark 安装的根目录，即本例中的/opt/spark）执行 spark-shell 命令来启动 Spark，可以通过 --master 参数来指定需要连接的集群。spark-shell 启动时，会看到一些启动日志，示例如下：

```
[root@node1 spark]# bin/spark-shell --master
spark://localhost:7077
2018-08-07 11:02:04 WARN  Utils:66 - Your
hostname, hidden.zzh.com resolves to
    a loopback address: 127.0.0.1; using
```

```
10.xxx.xxx.xxx instead (on interface
    eth0)
2018-08-07 11:02:04 WARN  Utils:66 - Set
SPARK_LOCAL_IP if you need to bind to
    another address
2018-08-07 11:02:04 WARN  NativeCodeLoader:62 -
Unable to load native-hadoop
    library for your platform... using builtin-
java classes where applicable
Setting default log level to "WARN".
To adjust logging level use
sc.setLogLevel(newLevel). For SparkR, use
    setLogLevel(newLevel).
Spark context Web UI available at http://
10.xxx.xxx.xxx:4040
Spark context available as 'sc' (master =
spark://localhost:7077, app id =
    app-20180807110212-0000).
Spark session available as 'spark'.
Welcome to
      ____              __
     / __/__  ___ _____/ /__
    _\ \/ _ \/ _ `/ __/  '_/
   /___/ .__/\_,_/_/ /_/\_\   version 2.3.1
      /_/

Using Scala version 2.11.8 (Java HotSpot(TM) 64-
Bit Server VM, Java 1.8.0_102)
Type in expressions to have them evaluated.
Type :help for more information.

scala>
```

如此便可以在“scala>”处输入我们想要输入的程序。

在将要演示的示例程序中，我们就近取材，以 bin/spark-shell 文件中的内容来进行单词统计。程序首先读取这个文件的内容，然后进行分词。这里的分词方法是使用空格进行分割的，最后统计单词出现的次数。下面将这些步骤进行拆分，一步步来讲解其中的细节。如无特殊说明，本章编写的示例均使用 Scala 语言。

首先通过 SparkContext（Spark 在启动时已经自动创建了一个 SparkContext 对象，是一个叫作 sc 的变量）的 textFile() 方法读取 bin/spark-shell 文件，参考如下：

```
scala> val rdd =
sc.textFile("/opt/spark/bin/spark-shell")
rdd: org.apache.spark.rdd.RDD[String] =
/opt/spark/bin/spark-shell
    MapPartitionsRDD[3] at textFile at
<console>:24
```

然后使用 split() 方法按照空格进行分词，之后又通过 flatMap() 方法对处理后的单词进行展平，展平之后使用 map(x=>(x,1))对每个单词计数1，参考如下：

```
scala> val wordmap = rdd.flatMap(_.split("
")).map(x=>(x,1))
wordmap: org.apache.spark.rdd.RDD[(String, Int)]
= MapPartitionsRDD[5] at map at
    <console>:25
```

最后使用 reduceByKey(_+_) 根据 key（也就是单词）进行计数，这个过程是一个混洗（Shuffle）的过程，参考如下：

```
scala> val wordreduce = wordmap.reduceByKey(_+_)
wordreduce: org.apache.spark.rdd.RDD[(String,
Int)] = ShuffledRDD[6] at
    reduceByKey at <console>:25
```

到这里我们便完成了单词统计，进一步地使用 take(10) 方法获取前面 10 个单词统计的结果，参考如下：

```
scala> wordreduce.take(10)
res3: Array[(String, Int)] = Array((scala,2),
(!=,1), (Unless,1), (this,4),
    (starting,1), (under,4), (its,1),
(reenable,2), (-Djline.terminal=unix",1),
    (CYGWIN*),1))
```

发现结果并没有按照某种顺序进行排序，如果要看到诸如单词出现次数前 10 的内容，那么还需要对统计后的结果进行排序。

```
scala> val wordsort =
    wordreduce.map(x=>
(x._2,x._1)).sortByKey(false).map(x=>(x._2,x._1))
wordsort: org.apache.spark.rdd.RDD[(String, Int)]
= MapPartitionsRDD[11] at map
    at <console>:25

scala> wordsort.take(10)
res2: Array[(String, Int)] = Array(("",91),
(#,37), (the,19), (in,7), (to,7),
    (for,6), (if,5), (then,5), (this,4),
(under,4))
```

上面的代码中首先使用 map(x=>(x._2,x._1) 对单词统计结果的键和值进行互换，然后通过 sortByKey(false) 方法对值进行降序排序，然后再次通过 map(x=>(x._2,x._1) 将键和值进行互换，最

终的结果按照降序排序。

Spark编程模型

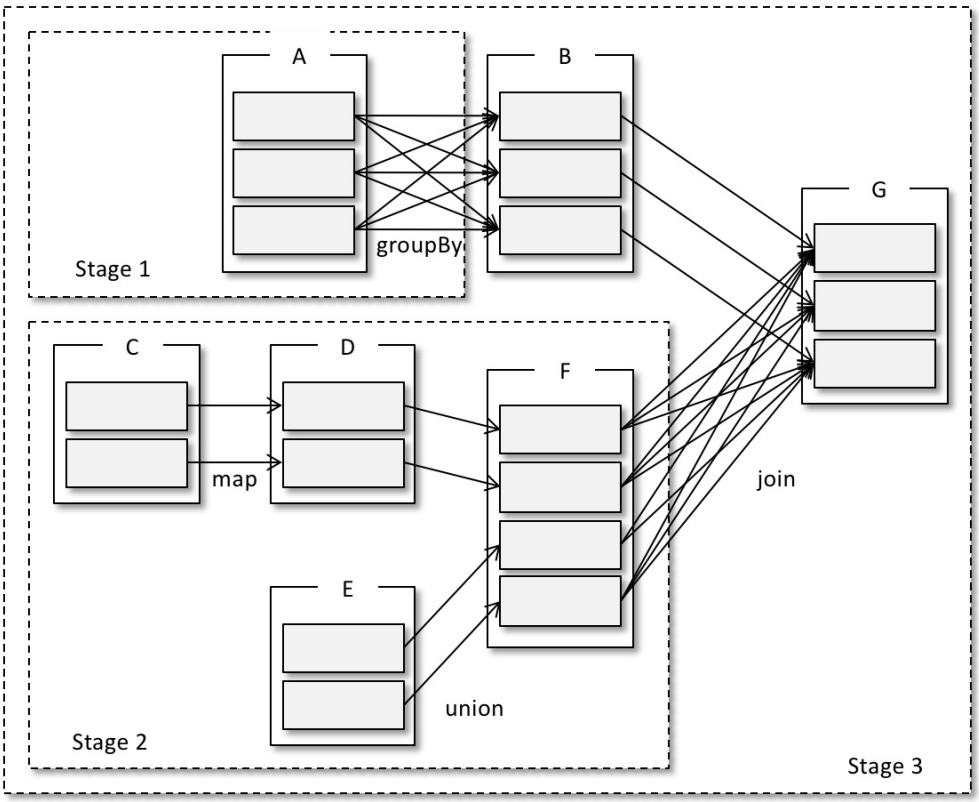
在 Spark 中，我们通过对分布式数据集的操作来表达计算意图，这些计算会自动在集群上并行执行。这样的数据集被称为弹性分布式数据集（Resilient Distributed Dataset），简称 RDD。RDD 是 Spark 对分布式数据和计算的基本抽象。在 Spark 中，对数据的所有操作不外乎创建 RDD、转换已有 RDD，以及调用 RDD 操作进行求值。在上一节的单词统计示例中，rdd 和 wordmap 都是 MapPartitionsRDD 类型的 RDD，而 wordreduce 是 ShuffledRDD 类型的 RDD。

RDD 支持2种类型的操作：转换操作（Transformation Operation）和行动操作（Action Operation）。有些资料还会细分为创建操作、转换操作、控制操作和行动操作4种类型。转换操作会由一个 RDD 生成一个新的 RDD。行动操作会对 RDD 计算出一个结果，并把结果返回驱动器程序，或者把结果存储到外部存储系统中。转换操作和行动操作的区别在于 Spark 计算 RDD 的方式不同。虽然可以在任何时候定义新的 RDD，但 Spark 只会惰性计算这些 RDD。它们只有第一次在一个行动操作中用到时才会真正计算。下表中给出了转换操作和行动操作之间对比的更多细节。

类别	函 数	区 别
转换操作	map、filter、groupBy、join、union、reduce、sort、partitionBy 等	返回值还是 RDD，不会马上提交给 Spark 集群运行
行动操作	count、collect、take、save、show 等	返回值不是 RDD，会形成 DAG 图，提交给 Spark 集群运行并立即返回结果

通过转换操作，从已有的 RDD 中派生出新的 RDD，Spark 会使用谱系图（Lineage Graph，很多资料也会翻译为“血统”）来记录这些不同 RDD 之间的依赖关系。Spark 需要用这些信息来按需计算每个 RDD，也可以依赖谱系图在持久化的 RDD 丢失部分数据时恢复丢失的数据。行动操作会把最终求得的结果返回驱动器程序，或者写入外部存储系统。由于行动操作需要生产实际的输出，所以它们会强制执行那些求值必须用到的 RDD 的转换操作。

Spark 中 RDD 计算是以分区（Partition）为单位的，将 RDD 划分为很多个分区分布到集群的节点中，分区的多少涉及对这个 RDD 进行并行计算的粒度。如图12-2所示，实线方框 A、B、C、D、E、F、G 都表示的是 RDD，阴影背景的矩形则表示分区。A、B、C、D、E、F、G 之间的依赖关系构成整个应用的谱系图。



依赖关系还可以分为窄依赖和宽依赖。窄依赖（Narrow Dependencies）是指每个父 RDD 的分区都至多被一个子 RDD 的分区使用，而宽依赖（Wide Dependencies）是指多个子 RDD 的分区依赖一个父 RDD 的分区。上图中，C和D之间是窄依赖，而A和

B之间是宽依赖。RDD 中行动操作的执行会以宽依赖为分界来构建各个调度阶段，各个调度阶段内部的窄依赖前后链接构成流水线。图中的3个虚线方框分别代表了3个不同的调度阶段。

对于执行失败的任务，只要它对应的调度阶段的父类信息仍然可用，那么该任务就会分散到其他节点重新执行。如果某些调度阶段不可用，则重新提交相应的任务，并以并行方式计算丢失的地方。在整个作业中，如果某个任务执行缓慢，则系统会在其他节点上执行该任务的副本，并取最先得到的结果作为最终的结果。

下面就以与上一节中相同的单词统计程序为例来分析 Spark 的编程模型，与上一节中所不同的是，这里是一个完整的 Scala 程序，程序对应的 Maven 依赖如下：

```
<dependency>
    <groupId>org.apache.spark</groupId>
    <artifactId>spark-core_2.11</artifactId>
    <version>2.3.1</version>
</dependency>
```

单词统计程序如代码清单33-1所示。

```
//代码清单33-1 单词统计程序
package scala.spark.demo
import org.apache.spark.{SparkConf, SparkContext}

object WordCount {
  def main(args: Array[String]): Unit ={
    val conf = new
SparkConf().setAppName("WordCount").setMaster("lo
cal")①
    val sc = new SparkContext(conf)②
    val rdd = sc.textFile("/opt/spark-2.3.1-bin-
hadoop2.7/bin/spark-shell")③
    val wordcount = rdd.flatMap(_.split("
")).map(x=>(x,1)).reduceByKey(_+_)④
    val wordsort = wordcount.map(x=>(x._2,x._1))
      .sortByKey(false).map(x=>(x._2,x._1))⑤
    wordsort.saveAsTextFile("/tmp/spark")⑥
    sc.stop()⑦
  }
}
```

main() 方法主体的第①和第②行中首先创建一个 SparkConf 对象来配置应用程序，然后基于这个 SparkConf 创建了一个 SparkContext 对象。一旦有了 SparkContext，就可以用它来创建 RDD，第③行代码中调用 sc.textFile() 来创建一个代表文件中各行文本的 RDD。第④行中 rdd.flatMap(_.split(" ")).map(x=>(x,1))这一段内容的依赖关系是窄依赖，而 reduceByKey(_+_) 操作对单词进行计数时属于宽依赖。第⑥行中将排序后的结果存储起来。最后第⑦行中使用 stop() 方法来关闭应用。

在$SPARK_HOME/bin 目录中还有一个 spark-submit 脚本，用于将应用快速部署到 Spark 集群。比如这里的 WordCount 程序，当我们希望通过 spark-submit 进行部署时，只需要将应用打包成 jar 包（即下面示例中的 wordcount.jar）并上传到 Spark 集群，然后通过 spark-submit 进行部署即可，示例如下：

```
[root@node1  spark]# bin/spark-submit --class
scala.spark.demo.WordCount wordcount.jar --
executor-memory 1G --master
spark://localhost:7077
2018-08-06 15:39:54 WARN  NativeCodeLoader:62 -
Unable to load native-hadoop
    library for your platform... using builtin-
java classes where applicable
2018-08-06 15:39:55 INFO  SparkContext:54 -
Running Spark version 2.3.1
2018-08-06 15:39:55 INFO  SparkContext:54 -
Submitted application: WordCount
2018-08-06 15:39:55 INFO  SecurityManager:54 -
Changing view acls to: root
2018-08-06 15:39:55 INFO  SecurityManager:54 -
Changing modify acls to: root
(....省略若干)
2018-08-07 12:25:47 INFO  AbstractConnector:318 -
Stopped
    Spark@6299e2c1{HTTP/1.1,[http/1.1]}
{0.0.0.0:4040}
2018-08-07 12:25:47 INFO  SparkUI:54 - Stopped
Spark web UI at
    http://10.199.172.111:4040
2018-08-07 12:25:47 INFO
MapOutputTrackerMasterEndpoint:54 -
    MapOutputTrackerMasterEndpoint stopped!
```

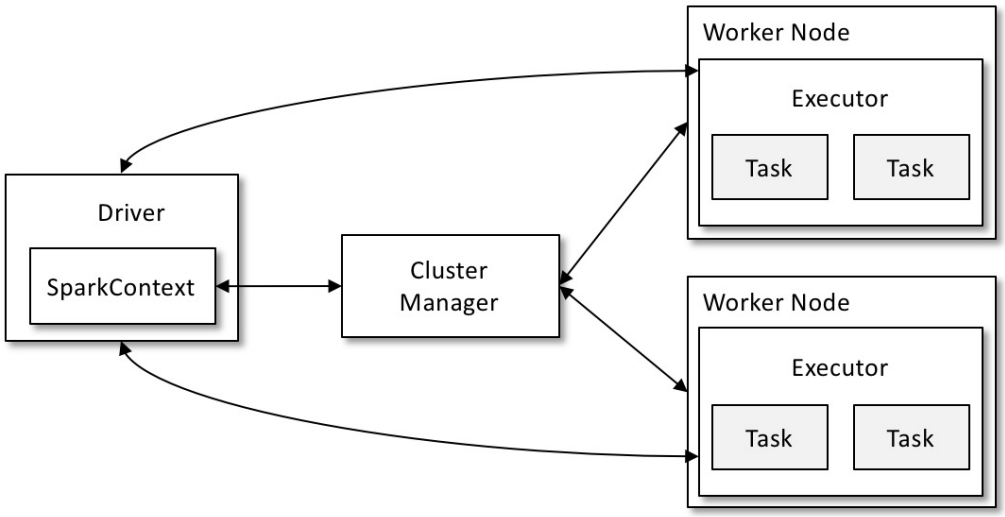
```
2018-08-07 12:25:47 INFO  MemoryStore:54 -
MemoryStore cleared
2018-08-07 12:25:47 INFO  BlockManager:54 -
BlockManager stopped
2018-08-07 12:25:47 INFO  BlockManagerMaster:54 -
BlockManagerMaster stopped
2018-08-07 12:25:47 INFO

OutputCommitCoordinator$OutputCommitCoordinatorEn
dpoint:54 -
    OutputCommitCoordinator stopped!
2018-08-06 15:46:57 INFO  SparkContext:54 -
Successfully stopped SparkContext
2018-08-06 15:46:57 INFO  ShutdownHookManager:54
- Shutdown hook called
2018-08-06 15:46:57 INFO  ShutdownHookManager:54
- Deleting directory
    /tmp/spark-fa955139-270c-4899-82b7-
4959983a1cb0
2018-08-06 15:46:57 INFO  ShutdownHookManager:54
- Deleting directory
    /tmp/spark-3f359966-2167-4bb9-863a-
2d8a8d5e8fbe
```

示例中的 --class 用来指定应用程序的主类，这里为 scala.spark.demo.WordCount；--executor-memory 用来指定执行器节点的内容，这里设置为1G。最后得到的输出结果如下所示。

```
[root@node1 spark]# ls /tmp/spark
part-00000  _SUCCESS
[root@node1 spark]# cat /tmp/spark/part-00000
(,91)
(#,37)
(the,19)
(in,7)
(to,7)
(for,6)
(if,5)
(then,5)
(under,4)
(stty,4)
(not,4)
```

Spark的运行结构

在分布式环境下，Spark 集群采用的是主从架构。如下图所示，在一个 Spark 集群中，有一个节点负责中央协调，调度各个分布式工作节点，这个中央协调节点被称为驱动器（Driver）节点，与之对应的工作节点被称为执行器（Executor）节点。驱动器节点可以和大量的执行器节点进行通信，它们都作为独立的进程运行。驱动器节点和所有的执行器节点一起被称为 Spark 应用（Application)。



Spark 应用通过一个叫作集群管理器（Cluster Manager）的外部服务在集群中的机器上启动。Spark 自带的集群管理器被称为独立集群管理器。Spark 也能运行在 YARN、Mesos、Kubernetes 这类开源集群管理器上。

Spark 驱动器节点是执行程序中的 `main()` 方法的进程。它执行用户编写的用来创建 SparkContext、RDD，以及进行 RDD 的转换操作和行动操作的代码。其实，当启动 spark-shell 时，就启动了一个 Spark 驱动程序。驱动程序一旦停止， Spark 应用也就结束了。

驱动器程序在 Spark 应用中有两个职责：将用户程序转为任务，以及为执行器节点调度任务。

Spark 驱动器程序负责把用户程序转为多个物理执行的单元，这些单元也被称为任务（Task）。任务是 Spark 中最小的工作单元，用户程序通常要启动成百上千的独立任务。从上层来看，所有的 Spark 程序都遵循同样的结构：程序从输入数据创建一系列 RDD，再使用转换操作派生出新的 RDD，最后使用行动操作收集或存储结果 RDD 中的数据。Spark 程序其实是隐式地创建了一个由操作组成的逻辑上的有向无环图（Directed Acyclic Graph，简称DAG）。当驱动器程序运行时，它会把这个逻辑图转为物理执行计划。

有了物理执行计划之后，Spark 驱动器程序必须在各执行器进程间协调任务的调度。执行器进程启动后，会向驱动器进程注册自己。因此，驱动器进程始终对应用中所有的执行器节点有完整的记录。每个执行器节点代表一个能够处理任务和存储 RDD 数据的进程。

Spark 驱动器程序会根据当前的执行器节点集合，尝试把所有任务基于数据所在位置分配给合适的执行器进程。当任务执行时，执行器进程会把缓存数据存储起来，而驱动器进程同样会跟踪这些缓存数据的位置，并且利用这些位置信息来调度以后的任务，以尽量减少数据的网络传输。

Spark 执行器节点是一种工作进程，负责在 Spark 作业中运行任务，任务间相互独立。Spark 应用启动时，执行器节点就被同步启动，并且始终伴随整个 Spark 应用的生命周期而存在。如果执行器节点发生异常或崩溃，那么 Spark 应用也可以继续执行。执行器进程有两大作用：第一，它们负责运行组成 Spark 应用的任务，并将结果返回给驱动器进程；第二，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 是直接缓存在执行器进程内的，因此任务可以在运行时充分利用缓存数据加速运算。

Spark 依赖于集群管理器来启动执行器节点，在某些特殊的情况下，也依赖集群管理器来启动驱动器节点。集群管理器是 Spark 中的可插拔式组件，这样既可选择 Spark 自带的独立集群管理，也可以选择前面提及的 YARN、Mesos 之类的外部集群管理器。

不论使用的是哪一种集群管理器，都可以使用 Spark 提供的统一脚本 spark-submit 将应用提交到该集群管理器上。通过不同的配置选项，spark-submit 可以连接到相应的集群管理器上，并控制应用使用的资源数量。在使用某些特定集群管理器时，spark-submit 也可以将驱动器节点运行在集群内部（比如一个 YARN 的工作节点）。但对于其他的集群管理器，驱动器节点只能被运行在本地机器上。

在集群上运行 Spark 应用的详细过程如下。

1. 用户通过 spark-submit 脚本提交应用。
2. spark-submit 脚本启动驱动器程序，调用用户定义的 main() 方法。
3. 驱动器程序与集群管理器通信，申请资源以启动执行器节点。
4. 集群管理器为驱动器程序启动执行器节点。
5. 驱动器执行用户应用中的操作。根据程序中定义的对 RDD 的转换操作和行动操作，驱动器节点把工作以任务的形式发送到执行器执行。
6. 任务在执行器程序中进行计算并保存结果。
7. 如果驱动器程序的 main() 方法退出，或者调用了 SparkContext.stop()，那么驱动器程序会中止执行器进程，并且通过集群管理器释放资源。