

关注公众号：大数据技术派，领取1024G资料，每日推送技术干货。

大数据技术之Spark

版本：V3.0



关注公众号：大数据技术派，回复“资料”，领取1024G资料。

第1章 Spark 概述

1.1 Spark 是什么



Apache Spark™ is a unified analytics engine for large-scale data processing.

Spark 是一种基于内存的快速、通用、可扩展的大数据分析计算引擎。

1.2 Spark and Hadoop

在之前的学习中，Hadoop 的 MapReduce 是大家广为熟知的计算框架，那为什么咱们还要学习新的计算框架 Spark 呢，这里就不得不提到 Spark 和 Hadoop 的关系。

首先从时间节点上来看：

➤ Hadoop

- 2006 年 1 月，Doug Cutting 加入 Yahoo，领导 Hadoop 的开发
- 2008 年 1 月，Hadoop 成为 Apache 顶级项目
- 2011 年 1.0 正式发布
- 2012 年 3 月稳定版发布
- 2013 年 10 月发布 2.X (Yarn)版本

➤ Spark

- 2009 年，Spark 诞生于伯克利大学的AMPLab 实验室
- 2010 年，伯克利大学正式开源了 Spark 项目
- 2013 年 6 月，Spark 成为了 Apache 基金会下的项目
- 2014 年 2 月，Spark 以飞快的速度成为了 Apache 的顶级项目
- 2015 年至今，Spark 变得愈发火爆，大量的国内公司开始重点部署或者使用 Spark

然后我们再从功能上来看：

➤ Hadoop

关注公众号：大数据技术派，回复“资料”，领取1024G资料。

- Hadoop 是由 java 语言编写的，在分布式服务器集群上存储海量数据并运行分布式分析应用的开源框架
- 作为 Hadoop 分布式文件系统，HDFS 处于 Hadoop 生态圈的最下层，存储着所有的数据，支持着 Hadoop 的所有服务。它的理论基础源于 Google 的 TheGoogleFileSystem 这篇论文，它是GFS 的开源实现。
- **MapReduce** 是一种编程模型，Hadoop 根据 Google 的 MapReduce 论文将其实现，作为 Hadoop 的分布式计算模型，是 Hadoop 的核心。基于这个框架，分布式并行程序的编写变得异常简单。综合了 HDFS 的分布式存储和 MapReduce 的分布式计算，Hadoop 在处理海量数据时，性能横向扩展变得非常容易。
- HBase 是对 Google 的 Bigtable 的开源实现，但又和 Bigtable 存在许多不同之处。HBase 是一个基于HDFS 的分布式数据库，擅长实时地随机读/写超大规模数据集。它也是 Hadoop 非常重要的组件。

➤ **Spark**

- Spark 是一种由 Scala 语言开发的快速、通用、可扩展的**大数据分析引擎**
- Spark Core 中提供了 Spark 最基础与最核心的功能
- Spark SQL 是 Spark 用来操作结构化数据的组件。通过 Spark SQL，用户可以使用 SQL 或者 Apache Hive 版本的 SQL 方言（HQL）来查询数据。
- Spark Streaming 是 Spark 平台上针对实时数据进行流式计算的组件，提供了丰富的处理数据流的API。

由上面的信息可以获知，Spark 出现的时间相对较晚，并且主要功能主要是用于数据计算，所以其实 Spark 一直被认为是Hadoop 框架的升级版。

1.3 Spark or Hadoop

Hadoop 的 MR 框架和Spark 框架都是数据处理框架，那么我们在使用时如何选择呢？

- Hadoop MapReduce 由于其设计初衷并不是为了满足循环迭代式数据流处理，因此在多并行运行的数据可复用场景（如：机器学习、图挖掘算法、交互式数据挖掘算法）中存在诸多计算效率等问题。所以 Spark 应运而生，Spark 就是在传统的MapReduce 计算框架的基础上，利用其计算过程的优化，从而大大加快了数据分析、挖掘的运行和读写速度，并将计算单元缩小到更适合并行计算和重复使用的RDD 计算模型。

关注公众号：大数据技术派，回复“资料”，领取1024G资料。

- 机器学习中 ALS、凸优化梯度下降等。这些都需要基于数据集或者数据集的衍生数据反复查询反复操作。MR 这种模式不太合适，即使多 MR 串行处理，性能和时间也是一个问题。数据的共享依赖于磁盘。另外一种交互式数据挖掘，MR 显然不擅长。而 Spark 所基于的 scala 语言恰恰擅长函数的处理。
- Spark 是一个分布式数据快速分析项目。它的核心技术是弹性分布式数据集（Resilient Distributed Datasets），提供了比MapReduce 丰富的模型，可以快速在内存中对数据集进行多次迭代，来支持复杂的数据挖掘算法和图形计算算法。
- Spark 和 Hadoop 的根本差异是多个作业之间的数据通信问题：Spark 多个作业之间数据通信是基于内存，而 Hadoop 是基于磁盘。
- Spark Task 的启动时间快。Spark 采用 fork 线程的方式，而 Hadoop 采用创建新的进程的方式。
- Spark 只有在 shuffle 的时候将数据写入磁盘，而 Hadoop 中多个 MR 作业之间的数据交互都要依赖于磁盘交互
- Spark 的缓存机制比HDFS 的缓存机制高效。

经过上面的比较，我们可以看出在绝大多数的数据计算场景中，Spark 确实会比 MapReduce 更有优势。但是Spark是基于内存的，所以在实际的生产环境中，由于内存的限制，可能会由于内存资源不够导致 Job 执行失败，此时，MapReduce 其实是一个更好的选择，所以 Spark 并不能完全替代 MR。

1.4 Spark 核心模块



➤ **Spark Core**

Spark Core 中提供了 Spark 最基础与最核心的功能，Spark 其他的功能如：Spark SQL，Spark Streaming，GraphX, MLlib 都是在 Spark Core 的基础上进行扩展的

➤ **Spark SQL**

Spark SQL 是 Spark 用来操作结构化数据的组件。通过 Spark SQL，用户可以使用 SQL 或者 Apache Hive 版本的 SQL 方言（HQL）来查询数据。

➤ **Spark Streaming**

Spark Streaming 是 Spark 平台上针对实时数据进行流式计算的组件，提供了丰富的处理数据流的API。

➤ **Spark MLlib**

MLlib 是 Spark 提供的一个机器学习算法库。MLlib 不仅提供了模型评估、数据导入等额外的功能，还提供了一些更底层的机器学习原语。

➤ **Spark GraphX**

GraphX 是 Spark 面向图计算提供的框架与算法库。

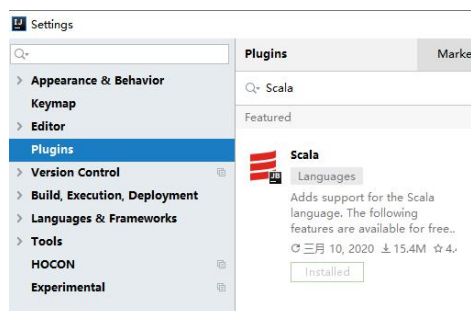
第2章 Spark 快速上手

在大数据早期的课程中我们已经学习了 MapReduce 框架的原理及基本使用，并了解了其底层数据处理的实现方式。接下来，就让咱们走进 Spark 的世界，了解一下它是如何带领我们完成数据处理的。

2.1 创建 Maven 项目

2.1.1 增加 Scala 插件

Spark 由 Scala 语言开发的，所以本课件接下来的开发所使用的语言也为 Scala，咱们当前使用的 Spark 版本为 3.0.0，默认采用的 Scala 编译版本为 2.12，所以后续开发时。我们依然采用这个版本。开发前请保证 IDEA 开发工具中含有 Scala 开发插件



2.1.2 增加依赖关系

修改 Maven 项目中的POM 文件，增加 Spark 框架的依赖关系。本课件基于 Spark3.0 版本，使用时请注意对应版本。

```
<dependencies>
  <dependency>
    <groupId>org.apache.spark</groupId>
    <artifactId>spark-core_2.12</artifactId>
    <version>3.0.0</version>
  </dependency>
</dependencies>
<build>
  <plugins>
    <!-- 该插件用于将 Scala 代码编译成 class 文件 -->
    <plugin>
      <groupId>net.alchim31.maven</groupId>
      <artifactId>scala-maven-plugin</artifactId>
      <version>3.2.2</version>
      <executions>
        <execution>
          <!-- 声明绑定到 maven 的 compile 阶段 -->
          <goals>
            <goal>testCompile</goal>
          </goals>
        </execution>
      </executions>
    </plugin>
  </plugins>
</build>
```

关注公众号：大数据技术派，回复“资料”，领取1024G资料。

```
</executions>
</plugin>
<plugin>
  <groupId>org.apache.maven.plugins</groupId>
  <artifactId>maven-assembly-plugin</artifactId>
  <version>3.1.0</version>
  <configuration>
    <descriptorRefs>
      <descriptorRef>jar-with-dependencies</descriptorRef>
    </descriptorRefs>
  </configuration>
  <executions>
    <execution>
      <id>make-assembly</id>
      <phase>package</phase>
      <goals>
        <goal>single</goal>
      </goals>
    </execution>
  </executions>
</plugin>
</plugins>
</build>
```

2.1.3 WordCount

为了能直观地感受 Spark 框架的效果，接下来我们实现一个大数据学科中最常见的教学

案例 WordCount

```
// 创建 Spark 运行配置对象
val sparkConf = new SparkConf().setMaster("local[*]").setAppName("WordCount")

// 创建 Spark 上下文环境对象（连接对象）
val sc : SparkContext = new SparkContext(sparkConf)

// 读取文件数据
val fileRDD: RDD[String] = sc.textFile("input/word.txt")

// 将文件中的数据进行分词
val wordRDD: RDD[String] = fileRDD.flatMap( _.split(" ") )

// 转换数据结构 word => (word, 1)
val word2OneRDD: RDD[(String, Int)] = wordRDD.map((_,1))

// 将转换结构后的数据按照相同的单词进行分组聚合
val word2CountRDD: RDD[(String, Int)] = word2OneRDD.reduceByKey(_+_ )

// 将数据聚合结果采集到内存中
val word2Count: Array[(String, Int)] = word2CountRDD.collect()

// 打印结果
word2Count.foreach(println)

//关闭 Spark 连接
sc.stop()
```

执行过程中，会产生大量的执行日志，如果为了能够更好的查看程序的执行结果，可以在项

目的 resources 目录中创建log4j.properties 文件，并添加日志配置信息：

```
log4j.rootCategory=ERROR, console
```

关注公众号：大数据技术派，回复“资料”，领取1024G资料。

```
log4j.appender.console=org.apache.log4j.ConsoleAppender
log4j.appender.console.target=System.err
log4j.appender.console.layout=org.apache.log4j.PatternLayout
log4j.appender.console.layout.ConversionPattern=%d{yy/MM/dd
HH:mm:ss} %p %c{1}: %m%n

# Set the default spark-shell log level to ERROR. When running the spark-shell,
the
# log level for this class is used to overwrite the root logger's log level, so
that
# the user can have different defaults for the shell and regular Spark apps.
log4j.logger.org.apache.spark.repl.Main=ERROR

# Settings to quiet third party logs that are too verbose
log4j.logger.org.spark_project.jetty=ERROR
log4j.logger.org.spark_project.jetty.util.component.AbstractLifeCycle=ERROR
log4j.logger.org.apache.spark.repl.SparkIMain$exprTyper=ERROR
log4j.logger.org.apache.spark.repl.SparkILoop$SparkILoopInterpreter=ERROR
log4j.logger.org.apache.parquet=ERROR
log4j.logger.parquet=ERROR

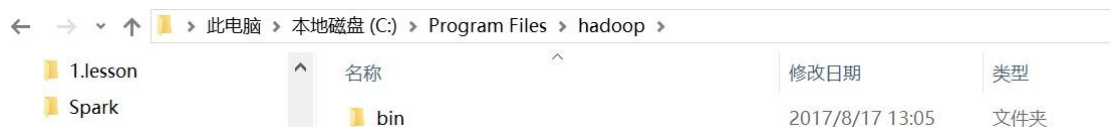
# SPARK-9183: Settings to avoid annoying messages when looking up nonexistent
UDFs in SparkSQL with Hive support
log4j.logger.org.apache.hadoop.hive.metastore.RetryingHMSHandler=FATAL
log4j.logger.org.apache.hadoop.hive ql.exec.FunctionRegistry=ERROR
```

2.1.4 异常处理

如果本机操作系统是 Windows，在程序中使用了 Hadoop 相关的东西，比如写入文件到 HDFS，则会遇到如下异常：

```
2017-09-14 16:08:34,907 ERROR --- [main] org.apache.hadoop.util.Shell(line:303) : Failed to locate the winutils binary in the hadoop binary path
java.io.IOException: Could not locate executable null\bin\winutils.exe in the Hadoop binaries.
    at org.apache.hadoop.util.Shell.getQualifiedBinPath(Shell.java:278)
    at org.apache.hadoop.util.Shell.getWinUtilsPath(Shell.java:300)
    at org.apache.hadoop.util.Shell.<clinit>(Shell.java:293)
    at org.apache.hadoop.util.StringUtils.<clinit>(StringUtils.java:76)
    at org.apache.hadoop.conf.Configuration.getTrimmedStrings(Configuration.java:1546)
    at org.apache.hadoop.hdfs.DFSClient.<init>(DFSClient.java:519)
    at org.apache.hadoop.hdfs.DFSClient.<init>(DFSClient.java:453)
    at org.apache.hadoop.hdfs.DistributedFileSystem.initialize(DistributedFileSystem.java:136)
    at org.apache.hadoop.fs.FileSystem.createFileSystem(FileSystem.java:2433)
```

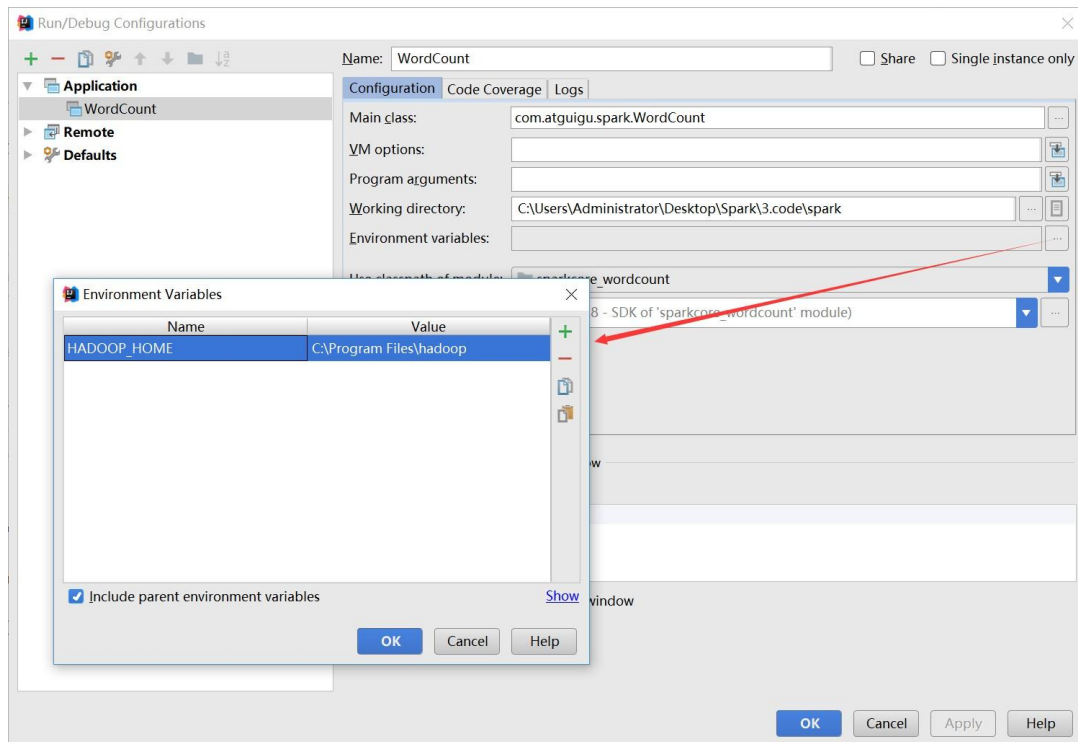
出现这个问题的原因，并不是程序的错误，而是windows 系统用到了 hadoop 相关的服务，解决办法是通过配置关联到 windows 的系统依赖就可以了



在 IDEA 中配置 Run Configuration，添加 HADOOP_HOME 变量

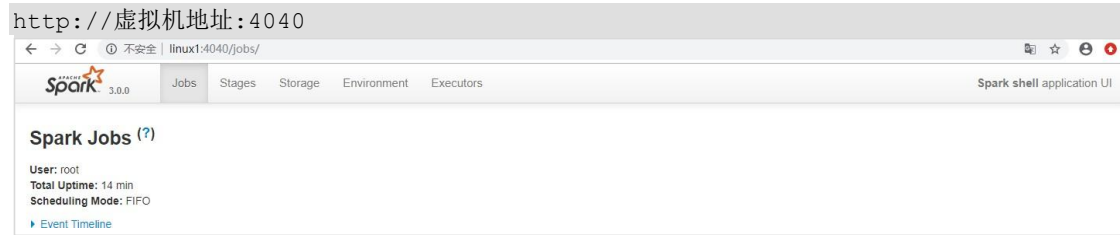


关注公众号：大数据技术派，领取1024G资料，每日推送技术干货。



关注公众号：大数据技术派，领取1024G资料。

2) 启动成功后，可以输入网址进行 Web UI 监控页面访问



3.1.3 命令行工具

在解压缩文件夹下的 data 目录中，添加 word.txt 文件。在命令行工具中执行如下代码指令（和 IDEA 中代码简化版一致）

```
sc.textFile("data/word.txt").flatMap(_.split("
")).map(_._1).reduceByKey(_+_).collect

scala> sc.textFile("data/word.txt").flatMap(_.split(" ")).map(_._1).reduceByKey(_+_).collect
res0: Array[(String, Int)] = Array((Hello,2), (Scala,1), (Spark,1))
```

3.1.4 退出本地模式

按键 Ctrl+C 或输入 Scala 指令

```
:quit
```

3.1.5 提交应用

```
bin/spark-submit \
--class org.apache.spark.examples.SparkPi \
--master local[2] \
./examples/jars/spark-examples_2.12-3.0.0.jar \
10
```

- 1) --class 表示要执行程序的主类，此处可以更换为咱们自己写的应用程序
- 2) --master local[2] 部署模式，默认为本地模式，数字表示分配的虚拟CPU 核数量
- 3) spark-examples_2.12-3.0.0.jar 运行的应用类所在的 jar 包，实际使用时，可以设定为咱们自己打的 jar 包
- 4) 数字 10 表示程序的入口参数，用于设定当前应用的任务数量

```
20/06/19 16:28:20 INFO Executor: Running task 9.0 in stage 0.0 (TID 9)
20/06/19 16:28:20 INFO TaskSetManager: Finished task 8.0 in stage 0.0 (TID 8) in 60 ms on linux1 (executor driver) (8/10)
20/06/19 16:28:20 INFO TaskSetManager: Finished task 7.0 in stage 0.0 (TID 7) in 77 ms on linux1 (executor driver) (9/10)
20/06/19 16:28:20 INFO Executor: Finished task 9.0 in stage 0.0 (TID 9). 957 bytes result sent to driver
20/06/19 16:28:20 INFO TaskSetManager: Finished task 9.0 in stage 0.0 (TID 9) in 28 ms on linux1 (executor driver) (10/10)
20/06/19 16:28:20 INFO TaskSchedulerImpl: Removed TaskSet 0.0, whose tasks have all completed, from pool
20/06/19 16:28:20 INFO DAGScheduler: ResultStage 0 (reduce at SparkPi.scala:38) finished in 1.436 s
20/06/19 16:28:20 INFO DAGScheduler: Job 0 is finished. Cancelling potential speculative or zombie tasks for this job
20/06/19 16:28:20 INFO TaskSchedulerImpl: Killing all running tasks in stage 0: Stage finished
20/06/19 16:28:20 INFO DAGScheduler: Job 0 finished: reduce at SparkPi.scala:38, took 1.553778 s
Pi is roughly 3.1403991403991403
20/06/19 16:28:20 INFO SparkUI: Stopped Spark web UI at http://linux1:4040
20/06/19 16:28:20 INFO MapOutputTrackerMasterEndpoint: MapOutputTrackerMasterEndpoint stopped!
20/06/19 16:28:20 INFO MemoryStore: MemoryStore cleared
20/06/19 16:28:20 INFO BlockManager: BlockManager stopped
20/06/19 16:28:20 INFO BlockManagerMaster: BlockManagerMaster stopped
20/06/19 16:28:20 INFO OutputCommitCoordinator$OutputCommitCoordinatorEndpoint: OutputCommitCoordinator stopped!
20/06/19 16:28:20 INFO SparkContext: Successfully stopped SparkContext
20/06/19 16:28:20 INFO ShutdownHookManager: Shutdown hook called
20/06/19 16:28:20 INFO ShutdownHookManager: Deleting directory /tmp/spark-c447deaf-e3f1-4207-8d7e-d084860315dd
20/06/19 16:28:20 INFO ShutdownHookManager: Deleting directory /tmp/spark-61053b82-261e-4d9e-b7a7-031c18849b09
```

3.2 Standalone 模式

local 本地模式毕竟只是用来进行练习演示的，真实工作中还是要将应用提交到对应的集群中去执行，这里我们来看看只使用 Spark 自身节点运行的集群模式，也就是我们所谓的独立部署（Standalone）模式。Spark 的 Standalone 模式体现了经典的master-slave 模式。集群规划：

	Linux1	Linux2	Linux3
Spark	Worker Master	Worker	Worker

3.2.1 解压缩文件

将 spark-3.0.0-bin-hadoop3.2.tgz 文件上传到Linux 并解压缩在指定位置

```
tar -zxvf spark-3.0.0-bin-hadoop3.2.tgz -C /opt/module
cd /opt/module
mv spark-3.0.0-bin-hadoop3.2 spark-standalone
```

3.2.2 修改配置文件

1) 进入解压缩后路径的 conf 目录，修改 slaves.template 文件名为 slaves

```
mv slaves.template slaves
```

2) 修改 slaves 文件，添加work 节点

```
linux1
linux2
linux3
```

3) 修改 spark-env.sh.template 文件名为 spark-env.sh

```
mv spark-env.sh.template spark-env.sh
```

4) 修改 spark-env.sh 文件，添加 JAVA_HOME 环境变量和集群对应的 master 节点

```
export JAVA_HOME=/opt/module/jdk1.8.0_144
SPARK_MASTER_HOST=linux1
SPARK_MASTER_PORT=7077
```

注意：7077 端口，相当于 hadoop3 内部通信的 8020 端口，此处的端口需要确认自己的 Hadoop 配置

5) 分发 spark-standalone 目录

```
xsync spark-standalone
```

3.2.3 启动集群

1) 执行脚本命令：

```
sbin/start-all.sh
```

```
[root@linux1 spark-standalone]# sbin/start-all.sh
starting org.apache.spark.deploy.master.Master, logging to /opt/module/spark-standalone/logs/spark-root-org.apache.spark.
deploy.master.Master-1-linux1.out
linux1: starting org.apache.spark.deploy.worker.Worker, logging to /opt/module/spark-standalone/logs/spark-root-org.apach
e.spark.deploy.worker.Worker-1-linux1.out
linux3: starting org.apache.spark.deploy.worker.Worker, logging to /opt/module/spark-standalone/logs/spark-root-org.apach
e.spark.deploy.worker.Worker-1-linux3.out
linux2: starting org.apache.spark.deploy.worker.Worker, logging to /opt/module/spark-standalone/logs/spark-root-org.apach
e.spark.deploy.worker.Worker-1-linux2.out
[root@linux1 spark-standalone]#
```

2) 查看三台服务器运行进程

```
=====linux1=====
3330 Jps
3238 Worker
3163 Master
=====linux2=====
2966 Jps
2908 Worker
=====linux3=====
2978 Worker
3036 Jps
```

3) 查看 Master 资源监控Web UI 界面: <http://linux1:8080>

Spark Master at spark://linux1:7077

URL: spark://linux1:7077
Alive Workers: 3
Cores in use: 8 Total, 0 Used
Memory in use: 4.8 GiB Total, 0.0 B Used
Resources in use:
Applications: 0 Running, 0 Completed
Drivers: 0 Running, 0 Completed
Status: ALIVE

Workers (3)

Worker ID	Address	State	Cores	Memory	Resources
worker-20200619163813-192.168.1.101-41219	192.168.1.101:41219	ALIVE	4 (0 Used)	2.8 GiB (0.0 B Used)	
worker-20200619163830-192.168.1.102-36861	192.168.1.102:36861	ALIVE	2 (0 Used)	1024.0 MiB (0.0 B Used)	
worker-20200619163830-192.168.1.103-46868	192.168.1.103:46868	ALIVE	2 (0 Used)	1024.0 MiB (0.0 B Used)	

Running Applications (0)

Application ID	Name	Cores	Memory per Executor	Resources Per Executor	Submitted Time	User	State	Duration
----------------	------	-------	---------------------	------------------------	----------------	------	-------	----------

Completed Applications (0)

Application ID	Name	Cores	Memory per Executor	Resources Per Executor	Submitted Time	User	State	Duration
----------------	------	-------	---------------------	------------------------	----------------	------	-------	----------

3.2.4 提交应用

```
bin/spark-submit \  
--class org.apache.spark.examples.SparkPi \  
--master spark://linux1:7077 \  
./examples/jars/spark-examples_2.12-3.0.0.jar \  
10
```

- 1) --class 表示要执行程序的主类
- 2) --master spark://linux1:7077 独立部署模式，连接到Spark 集群
- 3) spark-examples_2.12-3.0.0.jar 运行类所在的 jar 包
- 4) 数字 10 表示程序的入口参数，用于设定当前应用的任务数量

```
20/06/19 16:42:59 INFO CoarseGrainedSchedulerBackend$DriverEndpoint: Registered executor NettyRpcEndpointRef(spark-client://Executor)
(192.168.1.101:54494) with ID 0
20/06/19 16:42:59 INFO TaskSetManager: Finished task 3.0 in stage 0.0 (TID 3) in 13420 ms on 192.168.1.102 (executor 1) (9/10)
20/06/19 16:42:59 INFO TaskSetManager: Finished task 2.0 in stage 0.0 (TID 2) in 13431 ms on 192.168.1.102 (executor 1) (10/10)
20/06/19 16:42:59 INFO TaskSchedulerImpl: Removed TaskSet 0.0, whose tasks have all completed, from pool
20/06/19 16:42:59 INFO DAGScheduler: ResultStage 0 (reduce at SparkPi.scala:38) finished in 17.999 s
20/06/19 16:42:59 INFO CoarseGrainedSchedulerBackend$DriverEndpoint: Asking each executor to shut down
20/06/19 16:42:59 INFO TaskSchedulerImpl: Killing all running tasks in stage 0: Stage finished
20/06/19 16:42:59 INFO DAGScheduler: Job 0 finished: reduce at SparkPi.scala:38, took 18.633508 s
Pi is roughly 3.141943141943142
20/06/19 16:42:59 INFO SparkUI: Stopped Spark web UI at http://linux1:4040
20/06/19 16:42:59 INFO StandaloneSchedulerBackend: Shutting down all executors
20/06/19 16:42:59 INFO CoarseGrainedSchedulerBackend$DriverEndpoint: Asking each executor to shut down
20/06/19 16:42:59 INFO MapOutputTrackerMasterEndpoint: MapOutputTrackerMasterEndpoint stopped!
20/06/19 16:43:00 INFO MemoryStore: MemoryStore cleared
20/06/19 16:43:00 INFO BlockManager: BlockManager stopped
20/06/19 16:43:00 INFO BlockManagerMaster: BlockManagerMaster stopped
20/06/19 16:43:00 INFO OutputCommitCoordinator$OutputCommitCoordinatorEndpoint: OutputCommitCoordinator stopped!
20/06/19 16:43:00 INFO SparkContext: Successfully stopped SparkContext
20/06/19 16:43:00 INFO ShutdownHookManager: Shutdown hook called
20/06/19 16:43:00 INFO ShutdownHookManager: Deleting directory /tmp/spark-62035251-f7d8-46d7-95c7-a03c9a08edce
20/06/19 16:43:00 INFO ShutdownHookManager: Deleting directory /tmp/spark-2d166a60-b447-43fb-8115-811f379fea8e
```

执行任务时，会产生多个 Java 进程

```
-----linux1-----
4227 Master
4475 CoarseGrainedExecutorBackend
4332 Worker
4524 Jps
4399 SparkSubmit
-----linux2-----
3463 Worker
3561 Jps
3515 CoarseGrainedExecutorBackend
-----linux3-----
3573 Jps
3462 Worker
3516 CoarseGrainedExecutorBackend
```

执行节点进程

提交节点进程

执行任务时，默认采用服务器集群节点的总核数，每个节点内存 1024M。

Completed Applications (2)

Application ID	Name	Cores	Memory per Executor	Resources Per Executor	Submitted Time	User	State	Duration
app-20200619164410-0001	Spark Pi	8	1024.0 MIB		2020/06/19 16:44:10	root	FINISHED	14 s
app-20200619164412-0000	Spark Pi	8	1024.0 MIB		2020/06/19 16:42:12	root	FINISHED	47 s

3.2.5 提交参数说明

在提交应用中，一般会同时一些提交参数

```
bin/spark-submit \
--class <main-class>
--master <master-url> \
... # other options
<application-jar> \
[application-arguments]
```

参数	解释	可选值举例
--class	Spark 程序中包含主函数的类	
--master	Spark 程序运行的模式(环境)	模式: local[*]、spark://linux1:7077、Yarn
--executor-memory 1G	指定每个 executor 可用内存为 1G	符合集群内存配置即可，具体情况具体分析。
--total-executor-cores 2	指定所有executor 使用的cpu 核数为 2 个	
--executor-cores	指定每个executor 使用的cpu 核数	
application-jar	打包好的应用 jar，包含依赖。这个 URL 在集群中全局可见。比如 hdfs:// 共享存储系统，如果是	

	file:// path, 那么所有的节点的 path 都包含同样的 jar	
application-arguments	传给 main()方法的参数	

3.2.6 配置历史服务

由于 spark-shell 停止掉后，集群监控 linux1:4040 页面就看不到历史任务的运行情况，所以开发时都配置历史服务器记录任务运行情况。

1) 修改 spark-defaults.conf.template 文件名为 spark-defaults.conf

```
mv spark-defaults.conf.template spark-defaults.conf
```

2) 修改 spark-default.conf 文件，配置日志存储路径

```
spark.eventLog.enabled      true
spark.eventLog.dir          hdfs://linux1:8020/directory
```

注意：需要启动 hadoop 集群，HDFS 上的 directory 目录需要提前存在。

```
sbin/start-dfs.sh
hadoop fs -mkdir /directory
```

3) 修改 spark-env.sh 文件，添加日志配置

```
export SPARK_HISTORY_OPTS="
-Dspark.history.ui.port=18080
-Dspark.history.fs.logDirectory=hdfs://linux1:8020/directory
-Dspark.history.retainedApplications=30"
```

- 参数 1 含义：WEB UI 访问的端口号为 18080
- 参数 2 含义：指定历史服务器日志存储路径
- 参数 3 含义：指定保存 Application 历史记录的个数，如果超过这个值，旧的应用程序信息将被删除，这个是内存中的应用数，而不是页面上显示的应用数。

4) 分发配置文件

```
xsync conf
```

5) 重新启动集群和历史服务

```
sbin/start-all.sh
sbin/start-history-server.sh
```

6) 重新执行任务

```
bin/spark-submit \
--class org.apache.spark.examples.SparkPi \
--master spark://linux1:7077 \
./examples/jars/spark-examples_2.12-3.0.0.jar \
10
```

```
20/06/19 17:41:32 INFO StandaloneSchedulerBackend: Connected to Spark cluster with app ID app-20200619174132-0000
20/06/19 17:41:32 INFO Utils: Successfully started service 'org.apache.spark.network.netty.NettyBlockTransferService' on port 34345.
20/06/19 17:41:32 INFO NettyBlockTransferService: Server created on linux1:34345
20/06/19 17:41:32 INFO BlockManager: Using org.apache.spark.storage.RandomBlockReplicationPolicy for block replication policy
20/06/19 17:41:32 INFO StandaloneAppClient$ClientEndpoint: Executor added: app-20200619174132-0000/0 on worker-20200619174053-192.168.1.101-33517 (192.168.1.101:33517) with 4 core(s)
20/06/19 17:41:32 INFO StandaloneSchedulerBackend: Granted executor ID app-20200619174132-0000/0 on hostPort 192.168.1.101:33517 with 4 core(s), 1024.0 MiB RAM
20/06/19 17:41:32 INFO StandaloneAppClient$ClientEndpoint: Executor added: app-20200619174132-0000/1 on worker-20200619174040-192.168.1.102-44849 (192.168.1.102:44849) with 2 core(s)
20/06/19 17:41:32 INFO StandaloneSchedulerBackend: Granted executor ID app-20200619174132-0000/1 on hostPort 192.168.1.102:44849 with 2 core(s), 1024.0 MiB RAM
20/06/19 17:41:32 INFO StandaloneAppClient$ClientEndpoint: Executor added: app-20200619174132-0000/2 on worker-20200619174047-192.168.1.103-33749 (192.168.1.103:33749) with 2 core(s)
20/06/19 17:41:32 INFO StandaloneSchedulerBackend: Granted executor ID app-20200619174132-0000/2 on hostPort 192.168.1.103:33749 with 2 core(s), 1024.0 MiB RAM
20/06/19 17:41:32 INFO BlockManagerMaster: Registering BlockManager BlockManagerId(driver, linux1, 34345, None)
20/06/19 17:41:32 INFO BlockManagerMasterEndpoint: Registering block manager linux1:34345 with 366.3 MiB RAM, BlockManagerId(driver, linux1, 34345, None)
20/06/19 17:41:32 INFO BlockManagerMaster: Registered BlockManager BlockManagerId(driver, linux1, 34345, None)
20/06/19 17:41:32 INFO BlockManager: Initialized BlockManager: BlockManagerId(driver, linux1, 34345, None)
```

7) 查看历史服务: <http://linux1:18080>

<div>← → ↻ ① 不安全 linux1:18080</div> <div><div>spark 3.0.0</div><div>History Server</div><div>Event log directory: hdfs://linux1:8020/directory</div><div>Last updated: 2020-06-19 17:42:24</div><div>Client local time zone: Asia/Shanghai</div></div> <div>Search: <input type="text"/></div> <table><tr><th>Version</th><th>App ID</th><th>App Name</th><th>Started</th><th>Completed</th><th>Duration</th><th>Spark User</th><th>Last Updated</th><th colspan="2">Event Log</th></tr><tr><td>3.0.0</td><td>app-20200619174132-0000</td><td>Spark Pi</td><td>2020-06-19 17:41:27</td><td>2020-06-19 17:41:45</td><td>18 s</td><td>root</td><td>2020-06-19 17:41:45</td><td colspan="2">Download</td></tr></table>										Version	App ID	App Name	Started	Completed	Duration	Spark User	Last Updated	Event Log		3.0.0	app-20200619174132-0000	Spark Pi	2020-06-19 17:41:27	2020-06-19 17:41:45	18 s	root	2020-06-19 17:41:45	Download	
Version	App ID	App Name	Started	Completed	Duration	Spark User	Last Updated	Event Log																					
3.0.0	app-20200619174132-0000	Spark Pi	2020-06-19 17:41:27	2020-06-19 17:41:45	18 s	root	2020-06-19 17:41:45	Download																					

3.2.7 配置高可用（HA）

所谓的高可用是因为当前集群中的 Master 节点只有一个，所以会存在单点故障问题。为了解决单点故障问题，需要在集群中配置多个 Master 节点，一旦处于活动状态的 Master 发生故障时，由备用 Master 提供服务，保证作业可以继续执行。这里的高可用一般采用 Zookeeper 设置

集群规划:

	Linux1	Linux2	Linux3
Spark	Master Zookeeper Worker	Master Zookeeper Worker	Zookeeper Worker

1) 停止集群

```
sbin/stop-all.sh
```

2) 启动 Zookeeper

```
xstart zk
```

3) 修改 spark-env.sh 文件添加如下配置

注释如下内容:

```
#SPARK_MASTER_HOST=linux1
#SPARK_MASTER_PORT=7077
```

添加如下内容:

```
#Master 监控页面默认访问端口为 8080,但是可能会和 zookeeper 冲突,所以改成 8989,也可以自定义,访问 UI 监控页面时请注意
SPARK_MASTER_WEBUI_PORT=8989

export SPARK_DAEMON_JAVA_OPTS="
```



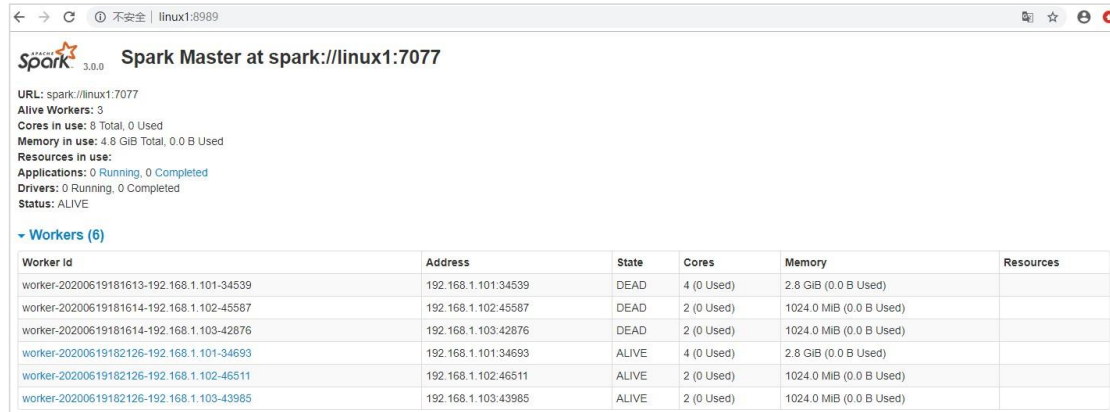
```
-Dspark.deploy.recoveryMode=ZOOKEEPER
-Dspark.deploy.zookeeper.url=linux1,linux2,linux3
-Dspark.deploy.zookeeper.dir=/spark"
```

4) 分发配置文件

```
xsync conf/
```

5) 启动集群

```
sbin/start-all.sh
```



Spark Master at spark://linux1:7077

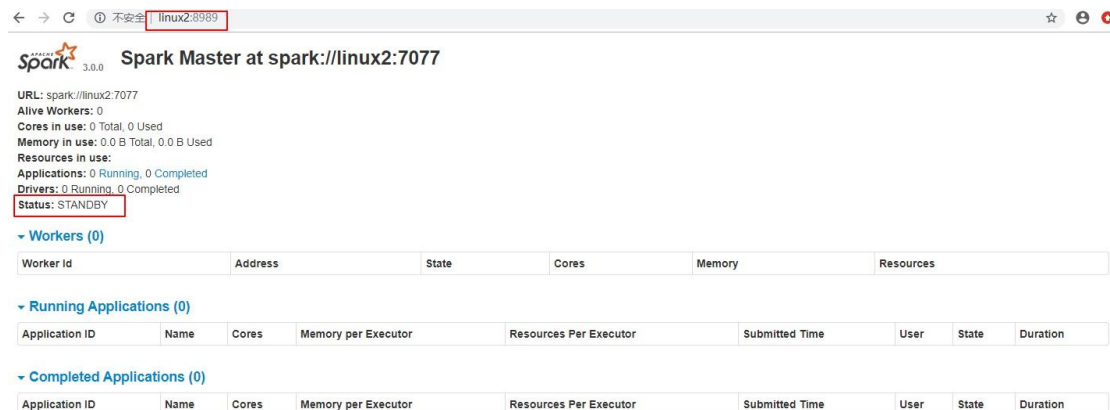
URL: spark://linux1:7077
Alive Workers: 3
Cores in use: 8 Total, 0 Used
Memory in use: 4.8 GiB Total, 0.0 B Used
Resources in use:
Applications: 0 Running, 0 Completed
Drivers: 0 Running, 0 Completed
Status: ALIVE

Workers (6)

Worker Id	Address	State	Cores	Memory	Resources
worker-20200619181613-192.168.1.101-34539	192.168.1.101:34539	DEAD	4 (0 Used)	2.8 GiB (0.0 B Used)	
worker-20200619181614-192.168.1.102-45587	192.168.1.102:45587	DEAD	2 (0 Used)	1024.0 MiB (0.0 B Used)	
worker-20200619181614-192.168.1.103-42876	192.168.1.103:42876	DEAD	2 (0 Used)	1024.0 MiB (0.0 B Used)	
worker-20200619182126-192.168.1.101-34693	192.168.1.101:34693	ALIVE	4 (0 Used)	2.8 GiB (0.0 B Used)	
worker-20200619182126-192.168.1.102-46511	192.168.1.102:46511	ALIVE	2 (0 Used)	1024.0 MiB (0.0 B Used)	
worker-20200619182126-192.168.1.103-43985	192.168.1.103:43985	ALIVE	2 (0 Used)	1024.0 MiB (0.0 B Used)	

6) 启动 linux2 的单独 Master 节点，此时 linux2 节点 Master 状态处于备用状态

```
[root@linux2 spark-standalone]# sbin/start-master.sh
```



Spark Master at spark://linux2:7077

URL: spark://linux2:7077
Alive Workers: 0
Cores in use: 0 Total, 0 Used
Memory in use: 0.0 B Total, 0.0 B Used
Resources in use:
Applications: 0 Running, 0 Completed
Drivers: 0 Running, 0 Completed
Status: STANDBY

Workers (0)

Worker Id	Address	State	Cores	Memory	Resources
-----------	---------	-------	-------	--------	-----------

Running Applications (0)

Application ID	Name	Cores	Memory per Executor	Resources Per Executor	Submitted Time	User	State	Duration
----------------	------	-------	---------------------	------------------------	----------------	------	-------	----------

Completed Applications (0)

Application ID	Name	Cores	Memory per Executor	Resources Per Executor	Submitted Time	User	State	Duration
----------------	------	-------	---------------------	------------------------	----------------	------	-------	----------

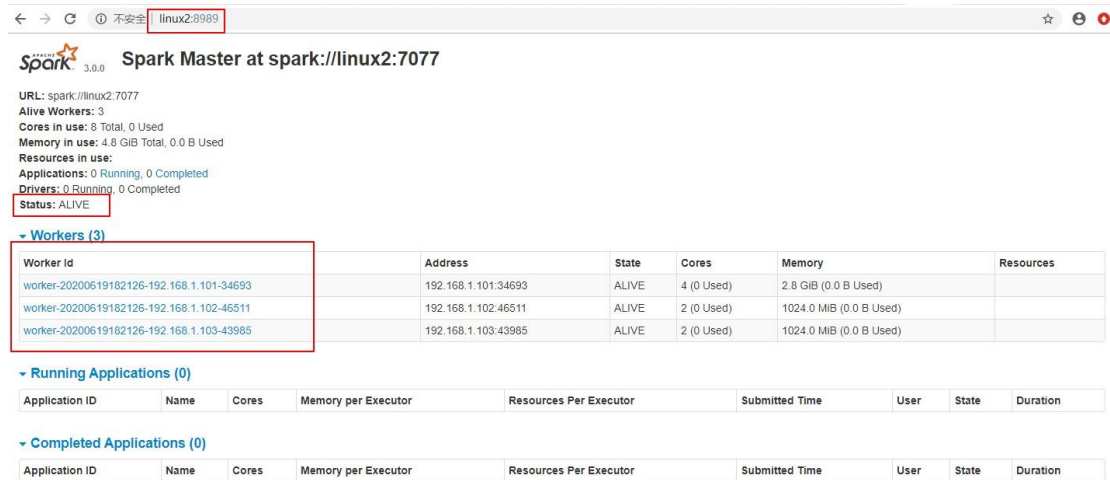
7) 提交应用到高可用集群

```
bin/spark-submit \
--class org.apache.spark.examples.SparkPi \
--master spark://linux1:7077,linux2:7077 \
./examples/jars/spark-examples_2.12-3.0.0.jar \
10
```

8) 停止 linux1 的 Master 资源监控进程

```
[root@linux1 spark-standalone]# jps
4673 JobHistoryServer
6802 Worker
6900 Jps
4342 DataNode
4966 QuorumPeerMain
4151 NameNode
4794 NodeManager
6703 Master
[root@linux1 spark-standalone]# kill -9 6703
[root@linux1 spark-standalone]#
```

- 9) 查看 linux2 的 Master 资源监控 Web UI，稍等一段时间后，linux2 节点的 Master 状态提升为活动状态



Spark Master at spark://linux2:7077

URL: spark://linux2:7077
Alive Workers: 3
Cores in use: 8 Total, 0 Used
Memory in use: 4.8 GiB Total, 0.0 B Used
Resources in use:
Applications: 0 Running, 0 Completed
Drivers: 0 Running, 0 Completed
Status: ALIVE

Workers (3)

Worker id	Address	State	Cores	Memory	Resources
worker-20200619182126-192.168.1.101-34693	192.168.1.101:34693	ALIVE	4 (0 Used)	2.8 GiB (0.0 B Used)	
worker-20200619182126-192.168.1.102-46511	192.168.1.102:46511	ALIVE	2 (0 Used)	1024.0 MiB (0.0 B Used)	
worker-20200619182126-192.168.1.103-43985	192.168.1.103:43985	ALIVE	2 (0 Used)	1024.0 MiB (0.0 B Used)	

Running Applications (0)

Application ID	Name	Cores	Memory per Executor	Resources Per Executor	Submitted Time	User	State	Duration
----------------	------	-------	---------------------	------------------------	----------------	------	-------	----------

Completed Applications (0)

Application ID	Name	Cores	Memory per Executor	Resources Per Executor	Submitted Time	User	State	Duration
----------------	------	-------	---------------------	------------------------	----------------	------	-------	----------

3.3 Yarn 模式

独立部署（Standalone）模式由 Spark 自身提供计算资源，无需其他框架提供资源。这种方式降低了和其他第三方资源框架的耦合性，独立性非常强。但是你也要记住，Spark 主要是计算框架，而不是资源调度框架，所以本身提供的资源调度并不是它的强项，所以还是和其他专业的资源调度框架集成会更靠谱一些。所以接下来我们来学习在强大的Yarn 环境下 Spark 是如何工作的（其实是因为在国内工作中，Yarn 使用的非常多）。

3.3.1 解压缩文件

将 spark-3.0.0-bin-hadoop3.2.tgz 文件上传到 linux 并解压缩，放置在指定位置。

```
tar -zxvf spark-3.0.0-bin-hadoop3.2.tgz -C /opt/module
cd /opt/module
mv spark-3.0.0-bin-hadoop3.2 spark-yarn
```

3.3.2 修改配置文件

- 1) 修改 hadoop 配置文件/opt/module/hadoop/etc/hadoop/yarn-site.xml，并分发

```
<!--是否启动一个线程检查每个任务正使用的物理内存量，如果任务超出分配值，则直接将其杀掉，默认是 true -->
<property>
  <name>yarn.nodemanager.pmem-check-enabled</name>
  <value>>false</value>
</property>

<!--是否启动一个线程检查每个任务正使用的虚拟内存量，如果任务超出分配值，则直接将其杀掉，默认是 true -->
<property>
  <name>yarn.nodemanager.vmem-check-enabled</name>
  <value>>false</value>
</property>
```

2) 修改 conf/spark-env.sh, 添加 JAVA_HOME 和 YARN_CONF_DIR 配置

```
mv spark-env.sh.template spark-env.sh
...
export JAVA_HOME=/opt/module/jdk1.8.0_144
YARN_CONF_DIR=/opt/module/hadoop/etc/hadoop
```

3.3.3 启动 HDFS 以及 YARN 集群

瞅啥呢，自己启动去！

3.3.4 提交应用

```
bin/spark-submit \
--class org.apache.spark.examples.SparkPi \
--master yarn \
--deploy-mode cluster \
./examples/jars/spark-examples_2.12-3.0.0.jar \
10
```

```
2020-06-19 22:47:42,632 INFO yarn.Client: Application report for application_1592559574387_0002 (state: RUNNING)
2020-06-19 22:47:43,875 INFO yarn.Client: Application report for application_1592559574387_0002 (state: RUNNING)
2020-06-19 22:47:44,879 INFO yarn.Client: Application report for application_1592559574387_0002 (state: RUNNING)
2020-06-19 22:47:46,358 INFO yarn.Client: Application report for application_1592559574387_0002 (state: RUNNING)
2020-06-19 22:47:47,601 INFO yarn.Client: Application report for application_1592559574387_0002 (state: RUNNING)
2020-06-19 22:47:48,621 INFO yarn.Client: Application report for application_1592559574387_0002 (state: RUNNING)
2020-06-19 22:47:49,629 INFO yarn.Client: Application report for application_1592559574387_0002 (state: RUNNING)
2020-06-19 22:47:50,643 INFO yarn.Client: Application report for application_1592559574387_0002 (state: RUNNING)
2020-06-19 22:47:51,673 INFO yarn.Client: Application report for application_1592559574387_0002 (state: RUNNING)
2020-06-19 22:47:52,675 INFO yarn.Client: Application report for application_1592559574387_0002 (state: FINISHED)
2020-06-19 22:47:52,676 INFO yarn.Client:
  client token: N/A
  diagnostics: N/A
  ApplicationMaster host: linux1
  ApplicationMaster RPC port: 33892
  queue: default
  start time: 1592578031667
  final status: SUCCEEDED
  tracking URL: http://linux2:8088/proxy/application_1592559574387_0002/
  user: root
2020-06-19 22:47:52,726 INFO util.ShutdownHookManager: Shutdown hook called
2020-06-19 22:47:52,730 INFO util.ShutdownHookManager: Deleting directory /tmp/spark-aedaddf7-b276-49ee-alb6-d98e31ab52da
2020-06-19 22:47:52,736 INFO util.ShutdownHookManager: Deleting directory /tmp/spark-4850a8e3-e949-490d-95cb-649f423d678d
```

查看 <http://linux2:8088> 页面，点击History，查看历史页面

The screenshot shows the Hadoop YARN web interface. On the left is a navigation menu with options like 'Cluster', 'About', 'Nodes', 'Node Labels', 'Applications', and 'Tools'. The main area is titled 'All Applications' and contains several summary tables for 'Cluster Metrics', 'Cluster Nodes Metrics', and 'Scheduler Metrics'. Below these is a detailed table of application entries. Two entries are highlighted with a red box:

ID	User	Name	Application Type	Queue	Application Priority	StartTime	LaunchTime	FinishTime	State	FinalSta
application_1592559574387_0002	root	org.apache.spark.examples.SparkPi	SPARK	default	0	Fri Jun 19 22:47:11 +0800 2020	Fri Jun 19 22:47:15 +0800 2020	Fri Jun 19 22:47:52 +0800 2020	FINISHED	SUCCEED
application_1592559574387_0001	root	Spark Pi	SPARK	default	0	Fri Jun 19 22:45:40 +0800 2020	Fri Jun 19 22:45:43 +0800 2020	Fri Jun 19 22:46:15 +0800 2020	FINISHED	SUCCEED

Showing 1 to 2 of 2 entries

```
command:
{{JAVA_HOME}}/bin/java \
-server \
-Xmx1024m \
-Djava.io.tmpdir={{PWD}}/tmp \
-Dspark.driver.port=49441 \
-Dspark.yarn.app.container.log.dir=LOG_DIR \
-XX:OnOutOfMemoryError=kill %p \
org.apache.spark.executor.CoarseGrainedExecutorBackend \
--driver-url \
spark://CoarseGrainedScheduler@linux1:49441 \
--executor-id \
<executorId> \
--hostname \
<hostname> \
--cores \
1 \
--app-id \
application_1587439373824_0001 \
--user-class-path \
file:{{PWD}}/__app__.jar \
1><LOG_DIR>/stdout \
2><LOG_DIR>/stderr

resources:
__spark_libs__ -> resource { scheme: "hdfs" host: "linux1" port: 9000 file: "/user/root/.sparkStaging/application_1587439373824_0001/__spark_libs__2267073519899007339.zip" }
__spark_conf__ -> resource { scheme: "hdfs" host: "linux1" port: 9000 file: "/user/root/.sparkStaging/application_1587439373824_0001/__spark_conf__.zip" } size: 193430 timestamp: 1587439373824
```

3.3.5 配置历史服务器

- 1) 修改 spark-defaults.conf.template 文件名为 spark-defaults.conf

```
mv spark-defaults.conf.template spark-defaults.conf
```

- 2) 修改 spark-default.conf 文件，配置日志存储路径

```
spark.eventLog.enabled      true
spark.eventLog.dir          hdfs://linux1:8020/directory
```

注意：需要启动 **hadoop** 集群，**HDFS** 上的目录需要提前存在。

```
[root@linux1 hadoop]# sbin/start-dfs.sh
[root@linux1 hadoop]# hadoop fs -mkdir /directory
```

- 3) 修改 spark-env.sh 文件，添加日志配置

```
export SPARK_HISTORY_OPTS="
-Dspark.history.ui.port=18080
-Dspark.history.fs.logDirectory=hdfs://linux1:8020/directory
-Dspark.history.retainedApplications=30"
```

- 参数 1 含义：WEB UI 访问的端口号为 18080
- 参数 2 含义：指定历史服务器日志存储路径
- 参数 3 含义：指定保存 Application 历史记录个数，如果超过这个值，旧的应用程序信息将被删除，这个是内存中的应用数，而不是页面上显示的应用数。

- 4) 修改 spark-defaults.conf

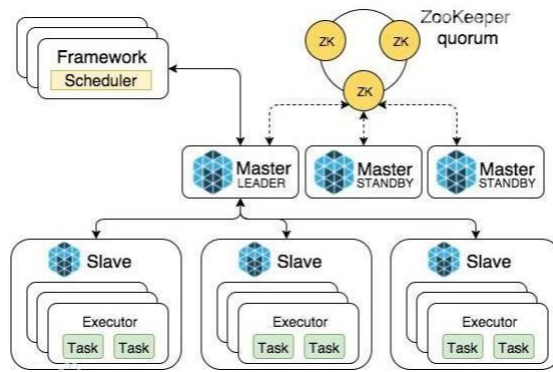
```
spark.yarn.historyServer.address=linux1:18080
spark.history.ui.port=18080
```

- 5) 启动历史服务

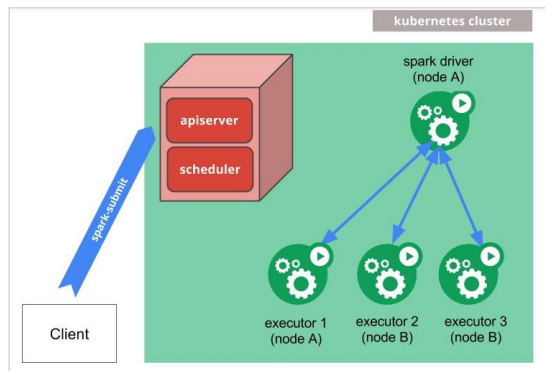
```
sbin/start-history-server.sh
```

- 6) 重新提交应用

```
bin/spark-submit \
--class org.apache.spark.examples.SparkPi \
--master yarn \
--deploy-mode client \
./examples/jars/spark-examples_2.12-3.0.0.jar \
10
```

容器化部署是目前业界很流行的一项技术，基于Docker 镜像运行能够让用户更加方便地对应用进行管理和运维。容器管理工具中最为流行的就是Kubernetes（k8s），而 Spark 也在最近的版本中支持了k8s 部署模式。这里我们也不做过多的讲解。给个链接大家自己感受一下：<https://spark.apache.org/docs/latest/running-on-kubernetes.html>



3.5 Windows 模式

在同学们自己学习时，每次都需要启动虚拟机，启动集群，这是一个比较繁琐的过程，并且会占大量的系统资源，导致系统执行变慢，不仅仅影响学习效果，也影响学习进度，Spark 非常暖心地提供了可以在windows 系统下启动本地集群的方式，这样，在不使用虚拟机的情况下，也能学习 Spark 的基本使用，摸摸哒！



在后续的教学，为了能够给同学们更加流畅的教学效果和教学体验，我们一般情况下都会采用windows 系统的集群来学习 Spark。


3.5.1 解压缩文件

将文件 spark-3.0.0-bin-hadoop3.2.tgz 解压缩到无中文无空格的路径中

3.5.2 启动本地环境

- 1) 执行解压缩文件路径下 bin 目录中的 spark-shell.cmd 文件，启动 Spark 本地环境

```
20/06/19 23:38:06 WARN NativeCodeLoader: Unable to load native-hadoop library for your platform... using builtin-java classes where applicable
Using Spark's default log4j profile: org/apache/spark/log4j-defaults.properties
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://windows10.microdone.cn:4040
Spark context available as 'sc' (master = local[*], app id = local-1592581095879).
Spark session available as 'spark'.
Welcome to


 version 3.0.0

Using Scala version 2.12.10 (Java HotSpot(TM) 64-Bit Server VM, Java 1.8.0_111)
Type in expressions to have them evaluated.
Type :help for more information.

scala>
```

- 2) 在 bin 目录中创建 input 目录，并添加 word.txt 文件，在命令行中输入脚本代码

```
20/06/19 23:43:33 WARN NativeCodeLoader: Unable to load native-hadoop library for your platform... using builtin-java classes where applicable
Using Spark's default log4j profile: org/apache/spark/log4j-defaults.properties
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://windows10.microdone.cn:4040
Spark context available as 'sc' (master = local[*], app id = local-1592581423102).
Spark session available as 'spark'.
Welcome to

 version 3.0.0

Using Scala version 2.12.10 (Java HotSpot(TM) 64-Bit Server VM, Java 1.8.0_111)
Type in expressions to have them evaluated.
Type :help for more information.

scala> sc.textFile("input/word.txt").flatMap(_.split(",")).map((_, 1)).reduceByKey(_+_).collect()
20/06/19 23:43:55 WARN ProcessMetricsGetter: Exception when trying to compute pagesize, as a result reporting of ProcessTree metrics is stopped
res0: Array[(String, Int)] = Array((world,1), (hello,1))
```

3.5.3 命令行提交应用

在 DOS 命令行窗口中执行提交指令

```
spark-submit --class org.apache.spark.examples.SparkPi --master local[2] ../examples/jars/spark-examples_2.12-3.0.0.jar 10
```

```
20/06/19 23:45:44 INFO Executor: Running task 9.0 in stage 0.0 (TID 9)
20/06/19 23:45:44 INFO TaskSetManager: Finished task 7.0 in stage 0.0 (TID 7) in 58 ms on windows10.microdone.cn (executor driver) (8/10)
20/06/19 23:45:44 INFO Executor: Finished task 8.0 in stage 0.0 (TID 8). 914 bytes result sent to driver
20/06/19 23:45:44 INFO TaskSetManager: Finished task 8.0 in stage 0.0 (TID 8) in 38 ms on windows10.microdone.cn (executor driver) (9/10)
20/06/19 23:45:44 INFO Executor: Finished task 9.0 in stage 0.0 (TID 9). 914 bytes result sent to driver
20/06/19 23:45:44 INFO TaskSetManager: Finished task 9.0 in stage 0.0 (TID 9) in 37 ms on windows10.microdone.cn (executor driver) (10/10)
20/06/19 23:45:44 INFO TaskSchedulerImpl: Removed TaskSet 0.0, whose tasks have all completed, from pool
20/06/19 23:45:44 INFO DAGScheduler: ResultStage 0 (reduce at SparkPi.scala:38) finished in 1.780 s
20/06/19 23:45:44 INFO DAGScheduler: Job 0 is finished. Cancelling potential speculative or zombie tasks for this job
20/06/19 23:45:44 INFO TaskSchedulerImpl: Killing all running tasks in stage 0: Stage finished
20/06/19 23:45:44 INFO DAGScheduler: Job 0 finished: reduce at SparkPi.scala:38, took 1.853504 s
Pi is roughly 3.1391991391991394
20/06/19 23:45:44 INFO SparkUI: Stopped Spark web UI at http://windows10.microdone.cn:4040
20/06/19 23:45:44 INFO MapOutputTrackerMasterEndpoint: MapOutputTrackerMasterEndpoint stopped!
20/06/19 23:45:44 INFO MemoryStore: MemoryStore cleared
20/06/19 23:45:44 INFO BlockManager: BlockManager stopped
20/06/19 23:45:44 INFO BlockManagerMaster: BlockManagerMaster stopped
20/06/19 23:45:44 INFO OutputCommitCoordinator$OutputCommitCoordinatorEndpoint: OutputCommitCoordinator stopped!
20/06/19 23:45:44 INFO SparkContext: Successfully stopped SparkContext
20/06/19 23:45:44 INFO ShutdownHookManager: Shutdown hook called
20/06/19 23:45:44 INFO ShutdownHookManager: Deleting directory C:\Users\18801\AppData\Local\Temp\spark-8bd95ad0-30ae-45cf-9257-94304c799329
20/06/19 23:45:44 INFO ShutdownHookManager: Deleting directory C:\Users\18801\AppData\Local\Temp\spark-c471b8d1-6da1-482f-b2bd-d2f5be278e48
```

3.6 部署模式对比

模式	Spark 安装机器数	需启动的进程	所属者	应用场景
Local	1	无	Spark	测试
Standalone	3	Master 及 Worker	Spark	单独部署
Yarn	1	Yarn 及 HDFS	Hadoop	混合部署

3.7 端口号

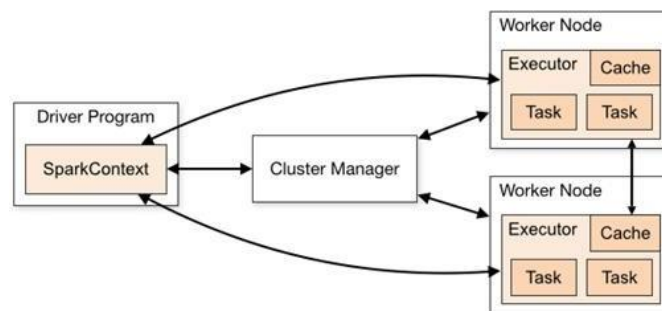
- Spark 查看当前 Spark-shell 运行任务情况端口号：4040（计算）
- Spark Master 内部通信服务端口号：7077
- Standalone 模式下，Spark Master Web 端口号：8080（资源）
- Spark 历史服务器端口号：18080
- Hadoop YARN 任务运行情况查看端口号：8088

第4章 Spark 运行架构

4.1 运行架构

Spark 框架的核心是一个计算引擎，整体来说，它采用了标准 master-slave 的结构。

如下图所示，它展示了一个 Spark 执行时的基本结构。图形中的Driver 表示 master，负责管理整个集群中的作业任务调度。图形中的Executor 则是 slave，负责实际执行任务。



4.2 核心组件

由上图可以看出，对于 Spark 框架有两个核心组件：

4.2.1 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：

- 将用户程序转化为作业（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

实际上，我们无法准确地描述Driver 的定义，因为在整个的编程过程中没有看到任何有关 Driver 的字眼。所以简单理解，所谓的 Driver 就是驱使整个应用运行起来的程序，也称之为 Driver 类。

4.2.2 Executor

Spark Executor 是集群中工作节点（Worker）中的一个 JVM 进程，负责在 Spark 作业中运行具体任务（Task），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：

- 负责运行组成 Spark 应用的任务，并将结果返回给驱动器进程
- 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。

4.2.3 Master & Worker

Spark 集群的独立部署环境中，不需要依赖其他的资源调度框架，自身就实现了资源调度的功能，所以环境中还有其他两个核心组件：Master 和 Worker，这里的 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，类似于 Yarn 环境中的 RM，而 Worker 呢，也是进程，一个 Worker 运行在集群中的一台服务器上，由 Master 分配资源对数据进行并行的处理和计算，类似于 Yarn 环境中 NM。

4.2.4 ApplicationMaster

Hadoop 用户向 YARN 集群提交应用程序时，提交程序中应该包含 ApplicationMaster，用于向资源调度器申请执行任务的资源容器 Container，运行用户自己的程序任务 job，监控整个任务的执行，跟踪整个任务的状态，处理任务失败等异常情况。

说的简单点就是，ResourceManager（资源）和 Driver（计算）之间的解耦合靠的就是 ApplicationMaster。

4.3 核心概念

4.3.1 Executor 与 Core

Spark Executor 是集群中运行在工作节点（Worker）中的一个 JVM 进程，是整个集群中的专门用于计算的节点。在提交应用中，可以提供参数指定计算节点的个数，以及对应的资源。这里的资源一般指的是工作节点 Executor 的内存大小和使用的虚拟 CPU 核（Core）数量。

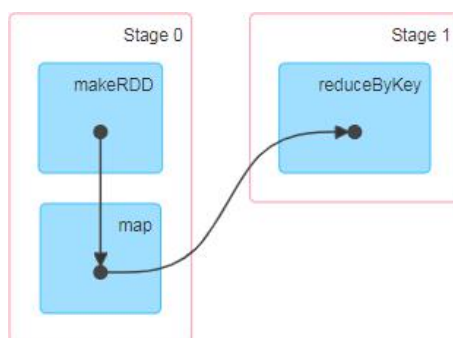
应用程序相关启动参数如下：

名称	说明
--num-executors	配置 Executor 的数量
--executor-memory	配置每个 Executor 的内存大小
--executor-cores	配置每个 Executor 的虚拟 CPU core 数量

4.3.2 并行度（Parallelism）

在分布式计算框架中一般都是多个任务同时执行，由于任务分布在不同的计算节点进行计算，所以能够真正地实现多任务并行执行，记住，这里是并行，而不是并发。这里我们将整个集群并行执行任务的数量称之为**并行度**。那么一个作业到底并行度是多少呢？这个取决于框架的默认配置。应用程序也可以在运行过程中动态修改。

4.3.3 有向无环图（DAG）



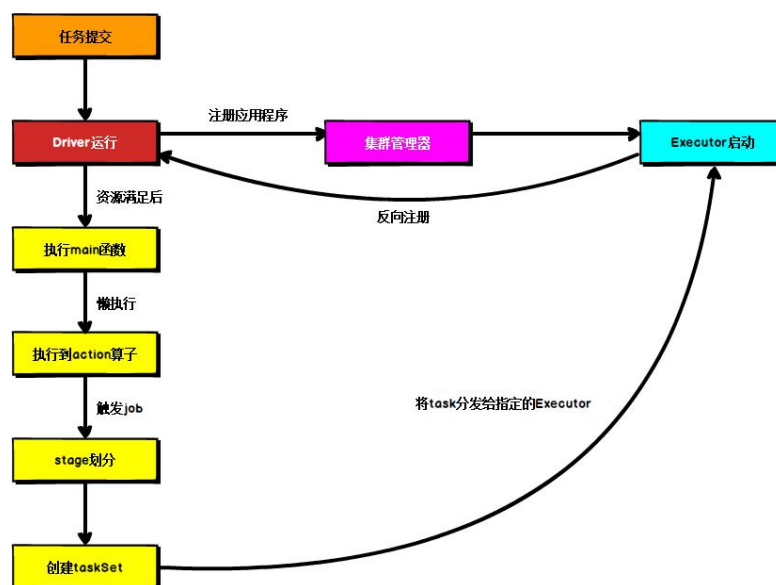
大数据计算引擎框架我们根据使用方式的不同一般会分为四类，其中第一类就是 Hadoop 所承载的 MapReduce,它将计算分为两个阶段，分别为 Map 阶段 和 Reduce 阶段。对于上层应用来说，就不得不想方设法去拆分算法，甚至于不得不在上层应用实现多个 Job 的串联，以完成一个完整的算法，例如迭代计算。由于这样的弊端，催生了支持 DAG 框架的产生。因此，支持 DAG 的框架被划分为第二代计算引擎。如 Tez 以及更上层的 Oozie。这里我们不去细究各种 DAG 实现之间的区别，不过对于当时的 Tez 和 Oozie 来说，大多还是批处理的任务。接下来就是以 Spark 为代表的第三代的计算引擎。第三代计算引擎的特点主要是 Job 内部的 DAG 支持（不跨越 Job），以及实时计算。

这里所谓的有向无环图，并不是真正意义的图形，而是由 Spark 程序直接映射成的数据流的高级抽象模型。简单理解就是将整个程序计算的执行过程用图形表示出来,这样更直观，更便于理解，可以用于表示程序的拓扑结构。

DAG（Directed Acyclic Graph）有向无环图是由点和线组成的拓扑图形，该图形具有方向，不会闭环。

4.4 提交流程

所谓的提交流程，其实就是我们开发人员根据需求写的应用程序通过 Spark 客户端提交给 Spark 运行环境执行计算的流程。在不同的部署环境中，这个提交过程基本相同，但是又有细微的区别，我们这里不进行详细的比较，但是因为国内工作中，将 Spark 引用部署到 Yarn 环境中会更多一些，所以本课程中的提交流程是基于 Yarn 环境的。



Spark 应用程序提交到 Yarn 环境中执行的时候，一般会有两种部署执行的方式：Client 和 Cluster。两种模式主要区别在于：Driver 程序的运行节点位置。

4.2.1 Yarn Client 模式

Client 模式将用于监控和调度的 Driver 模块在客户端执行，而不是在 Yarn 中，所以一般用于测试。

- 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
- ResourceManager 分配 container，在合适的 NodeManager 上启动 ApplicationMaster，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。

4.2.2 Yarn Cluster 模式

Cluster 模式将用于监控和调度的 Driver 模块启动在Yarn 集群资源中执行。一般应用于实际生产环境。

- 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。

第5章 Spark 核心编程

Spark 计算框架为了能够进行高并发和高吞吐的数据处理，封装了三大数据结构，用于处理不同的应用场景。三大数据结构分别是：

- RDD：弹性分布式数据集
- 累加器：分布式共享只写变量
- 广播变量：分布式共享只读变量

接下来我们一起来看看这三大数据结构是如何在数据处理中使用的。

5.1 RDD

5.1.1 什么是 RDD

RDD（Resilient Distributed Dataset）叫做弹性分布式数据集，是 Spark 中最基本的数据处理模型。代码中是一个抽象类，它代表一个弹性的、不可变、可分区、里面的元素可并行计算的集合。

- 弹性
 - 存储的弹性：内存与磁盘的自动切换；
 - 容错的弹性：数据丢失可以自动恢复；
 - 计算的弹性：计算出错重试机制；
 - 分片的弹性：可根据需要重新分片。
- 分布式：数据存储在大数据集不同节点上
- 数据集：RDD 封装了计算逻辑，并不保存数据
- 数据抽象：RDD 是一个抽象类，需要子类具体实现
- 不可变：RDD 封装了计算逻辑，是不可以改变的，想要改变，只能产生新的RDD，在新的RDD 里面封装计算逻辑
- 可分区、并行计算

5.1.2 核心属性

```
* Internally, each RDD is characterized by five main properties:  
*  
* - A list of partitions  
* - A function for computing each split  
* - A list of dependencies on other RDDs  
* - Optionally, a Partitioner for key-value RDDs (e.g. to say that the RDD is hash-partitioned)  
* - Optionally, a list of preferred locations to compute each split on (e.g. block locations for  
*   an HDFS file)
```

➤ 分区列表

RDD 数据结构中存在分区列表，用于执行任务时并行计算，是实现分布式计算的重要属性。

```
/**  
 * Implemented by subclasses to return the set of partitions in this RDD. This method will only  
 * be called once, so it is safe to implement a time-consuming computation in it.  
 *  
 * The partitions in this array must satisfy the following property:  
 *   rdd.partitions.zipWithIndex.forall { case (partition, index) => partition.index == index }  
 */  
protected def getPartitions: Array[Partition]
```

➤ 分区计算函数

Spark 在计算时，是使用分区函数对每一个分区进行计算

```
/**  
 * :: DeveloperApi ::  
 * Implemented by subclasses to compute a given partition.  
 */  
@DeveloperApi  
def compute(split: Partition, context: TaskContext): Iterator[T]
```

➤ RDD 之间的依赖关系

RDD 是计算模型的封装，当需求中需要将多个计算模型进行组合时，就需要将多个 RDD 建立依赖关系

```
/**  
 * Implemented by subclasses to return how this RDD depends on parent RDDs. This method will only  
 * be called once, so it is safe to implement a time-consuming computation in it.  
 */  
protected def getDependencies: Seq[Dependency[_]] = deps
```

➤ 分区器（可选）

当数据为 KV 类型数据时，可以通过设定分区器自定义数据的分区

```
/** Optionally overridden by subclasses to specify how they are partitioned. */  
@transient val partitioner: Option[Partitioner] = None
```

➤ 首选位置（可选）

关注公众号：大数据技术派，领取1024G资料。

计算数据时，可以根据计算节点的状态选择不同的节点位置进行计算

```
/**  
 * Optionally overridden by subclasses to specify placement preferences.  
 */  
protected def getPreferredLocations(split: Partition): Seq[String] = Nil
```

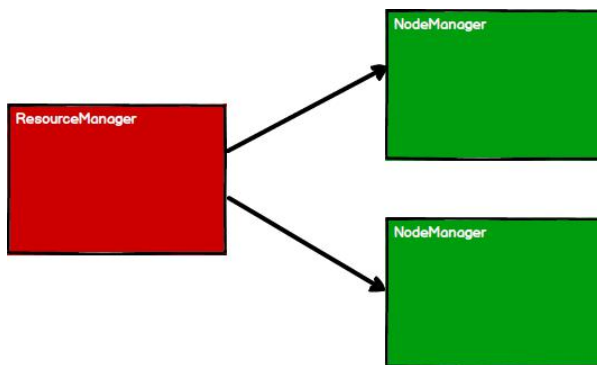
5.1.3 执行原理

从计算的角度来讲，数据处理过程中需要计算资源（内存 & CPU）和计算模型（逻辑）。
执行时，需要将计算资源和计算模型进行协调和整合。

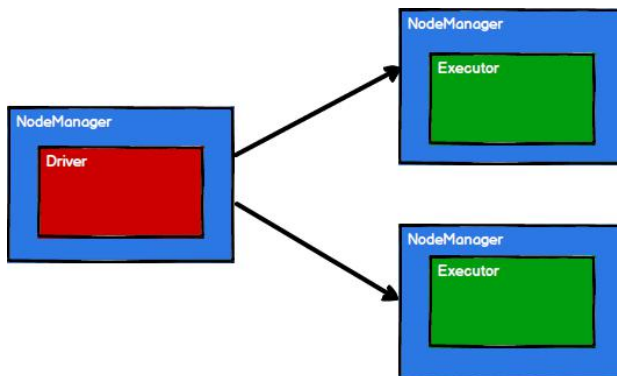
Spark 框架在执行时，先申请资源，然后将应用程序的数据处理逻辑分解成一个一个的计算任务。然后将任务发到已经分配资源的计算节点上，按照指定的计算模型进行数据计算。最后得到计算结果。

RDD 是 Spark 框架中用于数据处理的核心模型，接下来我们看看，在 Yarn 环境中，RDD 的工作原理：

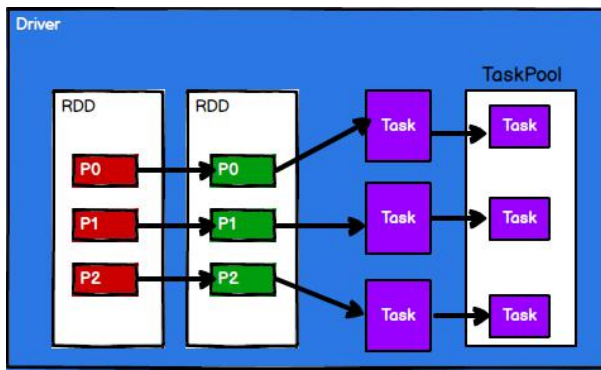
1) 启动 Yarn 集群环境



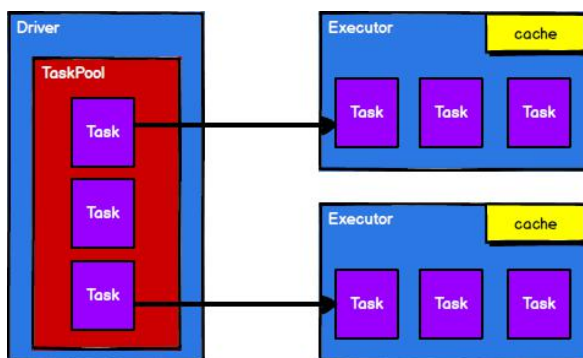
2) Spark 通过申请资源创建调度节点和计算节点



3) Spark 框架根据需求将计算逻辑根据分区划分成不同的任务



- 4) 调度节点将任务根据计算节点状态发送到对应的计算节点进行计算



从以上流程可以看出 RDD 在整个流程中主要用于将逻辑进行封装，并生成 Task 发送给 Executor 节点执行计算，接下来我们就一起看看 Spark 框架中RDD 是具体是如何进行数据处理的。

5.1.4 基础编程

5.1.4.1 RDD 创建

在 Spark 中创建RDD 的创建方式可以分为四种：

1) 从集合（内存）中创建 RDD

从集合中创建RDD，Spark 主要提供了两个方法：parallelize 和 makeRDD

```
val sparkConf =  
    new SparkConf().setMaster("local[*]").setAppName("spark")  
val sparkContext = new SparkContext(sparkConf)  
val rdd1 =  
    sparkContext.parallelize(List(1,  
        2,3,4)  
    )  
val rdd2 =  
    sparkContext.makeRDD(List(1,  
        2,3,4)  
    )  
rdd1.collect().foreach(println)
```

从底层代码实现来讲，makeRDD 方法其实就是parallelize 方法

```
def makeRDD[T:
  ClassTag](seq:
    Seq[T],
    numSlices: Int = defaultParallelism): RDD[T] = withScope
  {parallelize(seq, numSlices)}
```

2) 从外部存储（文件）创建RDD

由外部存储系统的数据集创建RDD 包括：本地的文件系统，所有Hadoop 支持的数据集，比如HDFS、HBase 等。

```
val sparkConf =
  new SparkConf().setMaster("local[*]").setAppName("spark")
val sparkContext = new SparkContext(sparkConf)
val fileRDD: RDD[String] = sparkContext.textFile("input")
fileRDD.collect().foreach(println)
sparkContext.stop()
```

3) 从其他 RDD 创建

主要是通过一个RDD 运算完后，再产生新的RDD。详情请参考后续章节

4) 直接创建 RDD (new)

使用 new 的方式直接构造RDD，一般由Spark 框架自身使用。

5.1.4.2 RDD 并行度与分区

默认情况下，Spark 可以将一个作业切分多个任务后，发送给 Executor 节点并行计算，而能够并行计算的任务数量我们称之为并行度。这个数量可以在构建RDD 时指定。记住，这里的并行执行的任务数量，并不是指的切分任务的数量，不要混淆了。

```
val sparkConf =
  new SparkConf().setMaster("local[*]").setAppName("spark")
val sparkContext = new SparkContext(sparkConf)
val dataRDD: RDD[Int] =
  sparkContext.makeRDD(
    List(1,2,3,4),
    4)
val fileRDD: RDD[String] =
  sparkContext.textFile(
    "input",
    2)
fileRDD.collect().foreach(println)
sparkContext.stop()
```

- 读取内存数据时，数据可以按照并行度的设定进行数据的分区操作，数据分区规则的

Spark 核心源码如下：

```
def positions(length: Long, numSlices: Int): Iterator[(Int, Int)] =
  {(0 until numSlices).iterator.map { i =>
    val start = ((i * length) / numSlices).toInt
    val end = (((i + 1) * length) / numSlices).toInt
    (start, end)
  }
}
```

```
}
```

- 读取文件数据时，数据是按照Hadoop 文件读取的规则进行切片分区，而切片规则和数
据读取的规则有些差异，具体 Spark 核心源码如下

```
public InputSplit[] getSplits(JobConf job, int numSplits)
    throws IOException {

    long totalSize = 0;                                // compute total size
    for (FileStatus file: files)                        // check we have valid files
    {if (file.isDirectory()) {
        throw new IOException("Not a file: " + file.getPath());
    }
    totalSize += file.getLen();
    }

    long goalSize = totalSize / (numSplits == 0 ? 1 : numSplits);
    long minSize = Math.max(job.getLong(org.apache.hadoop.mapreduce.lib.input.
        FileInputFormat.SPLIT_MINSIZE, 1), minSplitSize);

    ...

    for (FileStatus file: files) {

        ...

        if (isSplittable(fs, path)) {
            long blockSize = file.getBlockSize();
            long splitSize = computeSplitSize(goalSize, minSize, blockSize);

            ...

        }
    }
    protected long computeSplitSize(long goalSize, long minSize,
                                    long blockSize) {
        return Math.max(minSize, Math.min(goalSize, blockSize));
    }
}
```

5.1.4.3 RDD 转换算子

RDD 根据数据处理方式的不同将算子整体上分为Value 类型、双 Value 类型和Key-Value 类型

- Value 类型

1) map

➤ 函数签名

```
def map[U: ClassTag](f: T => U): RDD[U]
```

➤ 函数说明

将处理的数据逐条进行映射转换，这里的转换可以是类型的转换，也可以是值的转换。

```
val dataRDD: RDD[Int] = sparkContext.makeRDD(List(1,2,3,4))
val dataRDD1: RDD[Int] = dataRDD.map(
    num => {
```

```
        num * 2
      }
    )
    val dataRDD2: RDD[String] =
      dataRDD1.map(num => {
        "" + num
      })
  )
```

❖ 小功能：从服务器日志数据 `apache.log` 中获取用户请求URL 资源路径

2) mapPartitions

➤ 函数签名

```
def mapPartitions[U]:
  ClassTag](f: Iterator[T] =>
  Iterator[U],
  preservesPartitioning: Boolean = false): RDD[U]
```

➤ 函数说明

将待处理的数据以分区为单位发送到计算节点进行处理，这里的处理是指可以进行任意的处理，哪怕是过滤数据。

```
val dataRDD1: RDD[Int] =
  dataRDD.mapPartitions(datas => {
    datas.filter(_==2)
  })
```

❖ 小功能：获取每个数据分区的最大值



思考一个问题：`map` 和 `mapPartitions` 的区别？

➤ 数据处理角度

`Map` 算子是分区内一个数据一个数据的执行，类似于串行操作。而 `mapPartitions` 算子是以分区为单位进行批处理操作。

➤ 功能的角度

`Map` 算子主要目的将数据源中的数据进行转换和改变。但是不会减少或增多数据。

`MapPartitions` 算子需要传递一个迭代器，返回一个迭代器，没有要求的元素的个数保持不变，所以可以增加或减少数据

➤ 性能的角度

`Map` 算子因为类似于串行操作，所以性能比较低，而是 `mapPartitions` 算子类似于批处理，所以性能较高。但是`mapPartitions` 算子会长时间占用内存，那么这样会导致内存可能

关注公众号：大数据技术派，领取1024G资料，每日推送技术干货。

不够用，出现内存溢出的错误。所以在内存有限的情况下，不推荐使用。使用 `map` 操作。

关注公众号：大数据技术派，领取1024G资料。

完成比完美更重要

3) mapPartitionsWithIndex

➤ 函数签名

```
def mapPartitionsWithIndex[U]:  
  ClassTag](f: (Int, Iterator[T]) =>  
    Iterator[U],  
  preservesPartitioning: Boolean = false): RDD[U]
```

➤ 函数说明

将待处理的数据以分区为单位发送到计算节点进行处理，这里的处理是指可以进行任意的处理，哪怕是过滤数据，在处理时同时可以获取当前分区索引。

```
val dataRDD1 =  
  dataRDD.mapPartitionsWithIndex((index,  
    datas) => {  
    datas.map(index, _)  
  })
```

❖ 小功能：获取第二个数据分区的数据

4) flatMap

➤ 函数签名

```
def flatMap[U: ClassTag](f: T => TraversableOnce[U]): RDD[U]
```

➤ 函数说明

将处理的数据进行扁平化后再进行映射处理，所以算子也称之为扁平映射

```
val dataRDD =  
  sparkContext.makeRDD(List(List(1,2),  
    List(3,4)  
  ),1)  
val dataRDD1 =  
  dataRDD.flatMap(list =>
```

❖ 小功能：将 List(List(1,2),3,List(4,5))进行扁平化操作

5) glom

➤ 函数签名

```
def glom(): RDD[Array[T]]
```

➤ 函数说明

关注公众号：大数据技术派，领取1024G资料，每日推送技术干货。

将同一个分区的数据直接转换为相同类型的内存数组进行处理，分区不变

关注公众号：大数据技术派，领取1024G资料。

```
val dataRDD =  
    sparkContext.makeRDD(List(1,2,3,4  
) ,1)  
val dataRDD1:RDD[Array[Int]] = dataRDD.glom()
```

- ❖ 小功能：计算所有分区最大值求和（分区内取最大值，分区间最大值求和）

6) groupBy

- 函数签名

```
def groupBy[K](f: T => K)(implicit kt: ClassTag[K]): RDD[(K, Iterable[T])]
```

- 函数说明

将数据根据指定的规则进行分组, 分区默认不变, 但是数据会被打乱重新组合, 我们将这样的操作称之为shuffle。极限情况下, 数据可能被分在同一个分区中

一个组的数据在一个分区中, 但是并不是说一个分区中只有一个组

```
val dataRDD = sparkContext.makeRDD(List(1,2,3,4),1)  
val dataRDD1 = dataRDD.groupBy(  
    _%2  
)
```

- ❖ 小功能：将 List("Hello", "hive", "hbase", "Hadoop")根据单词首写字母进行分组。
- ❖ 小功能：从服务器日志数据 apache.log 中获取每个时间段访问量。
- ❖ 小功能：WordCount。

7) filter

- 函数签名

```
def filter(f: T => Boolean): RDD[T]
```

- 函数说明

将数据根据指定的规则进行筛选过滤, 符合规则的数据保留, 不符合规则的数据丢弃。

当数据进行筛选过滤后, 分区不变, 但是分区内的数据可能不均衡, 生产环境下, 可能会出现数据倾斜。

```
val dataRDD =  
    sparkContext.makeRDD(List(1,2,3,4  
) ,1)  
val dataRDD1 = dataRDD.filter(_%2 == 0)
```

- ❖ 小功能：从服务器日志数据 apache.log 中获取 2015 年 5 月 17 日的请求路径

8) sample

➤ 函数签名

```
def sample( withReplacement: Boolean, fraction: Double, seed: Long = Utils.random.nextLong): RDD[T]
```

➤ 函数说明

根据指定的规则从数据集中抽取数据

```
val dataRDD =
  sparkContext.makeRDD(List(1,2,3,4),1)
// 抽取数据不放回（伯努利算法）
// 伯努利算法：又叫 0、1 分布。例如扔硬币，要么正面，要么反面。
// 具体实现：根据种子和随机算法算出一个数和第二个参数设置几率比较，小于第二个参数要，大于不要
// 第一个参数：抽取的数据是否放回，false：不放回
// 第二个参数：抽取的几率，范围在[0,1]之间,0：全不取；1：全取；
// 第三个参数：随机数种子
val dataRDD1 = dataRDD.sample(false, 0.5)
// 抽取数据放回（泊松算法）
// 第一个参数：抽取的数据是否放回，true：放回；false：不放回
// 第二个参数：重复数据的几率，范围大于等于 0.表示每一个元素被期望抽取到的次数
// 第三个参数：随机数种子
val dataRDD2 = dataRDD.sample(true, 2)
```



思考一个问题：有啥用，抽奖吗？

9) distinct

➤ 函数签名

```
def distinct()(implicit ord: Ordering[T] = null): RDD[T]
def distinct(numPartitions: Int)(implicit ord: Ordering[T] = null): RDD[T]
```

➤ 函数说明

将数据集中重复的数据去重

```
val dataRDD =
  sparkContext.makeRDD(List(1,2,3,4,1,2),1)
val dataRDD1 = dataRDD.distinct()
```



思考一个问题：如果不用该算子，你有什么办法实现数据去重？

10) coalesce

➤ 函数签名

```
def coalesce(numPartitions: Int, shuffle: Boolean = false,  
             partitionCoalescer: Option[PartitionCoalescer] = Option.empty)  
             (implicit ord: Ordering[T] = null)  
             : RDD[T]
```

➤ 函数说明

根据数据量**缩减分区**，用于大数据集过滤后，提高小数据集的执行效率

当 spark 程序中，存在过多的小任务的时候，可以通过 coalesce 方法，收缩合并分区，减少分区的个数，减小任务调度成本

```
val dataRDD =  
    sparkContext.makeRDD(List(1,2,3,4,1,  
    2  
), 6)
```



思考一个问题：我想要扩大分区，怎么办？

11) repartition

➤ 函数签名

```
def repartition(numPartitions: Int)(implicit ord: Ordering[T] = null): RDD[T]
```

➤ 函数说明

该操作内部其实执行的是 coalesce 操作，参数 shuffle 的默认值为 true。无论是将分区数多的 RDD 转换为分区数少的 RDD，还是将分区数少的 RDD 转换为分区数多的 RDD，repartition 操作都可以完成，因为无论如何都会经 shuffle 过程。

```
val dataRDD =  
    sparkContext.makeRDD(List(1,2,3,4,1,  
    2  
), 2)
```



思考一个问题：coalesce 和 repartition 区别？

12) sortBy

➤ 函数签名

```
def  
sortBy[K](f:
```

关注公众号：大数据技术派，领取1024G资料，每日推送技术干货。

$(T) \Rightarrow K,$

关注公众号：大数据技术派，领取1024G资料。

```
ascending: Boolean = true,  
numPartitions: Int = this.partitions.length)  
(implicit ord: Ordering[K], ctag: ClassTag[K]): RDD[T]
```

➤ 函数说明

该操作用于排序数据。在排序之前，可以将数据通过 f 函数进行处理，之后按照 f 函数处理的结果进行排序，默认为升序排列。排序后新产生的 RDD 的分区数与原RDD 的分区数一致。中间存在 shuffle 的过程

```
val dataRDD =  
  sparkContext.makeRDD(List(1,2,3,4,1,  
    2  
  ),2)
```

● 双 Value 类型

13) intersection

➤ 函数签名

```
def intersection(other: RDD[T]): RDD[T]
```

➤ 函数说明

对源RDD 和参数RDD 求交集后返回一个新的RDD

```
val dataRDD1 = sparkContext.makeRDD(List(1,2,3,4))  
val dataRDD2 = sparkContext.makeRDD(List(3,4,5,6))  
val dataRDD = dataRDD1.intersection(dataRDD2)
```



思考一个问题：如果两个RDD 数据类型不一致怎么办？

14) union

➤ 函数签名

```
def union(other: RDD[T]): RDD[T]
```

➤ 函数说明

对源RDD 和参数RDD 求并集后返回一个新的RDD

```
val dataRDD1 = sparkContext.makeRDD(List(1,2,3,4))  
val dataRDD2 = sparkContext.makeRDD(List(3,4,5,6))  
val dataRDD = dataRDD1.union(dataRDD2)
```



思考一个问题：如果两个RDD 数据类型不一致怎么办？

15) subtract

➤ 函数签名

```
def subtract(other: RDD[T]): RDD[T]
```

➤ 函数说明

以一个 RDD 元素为主，去除两个 RDD 中重复元素，将其他元素保留下来。求差集

```
val dataRDD1 = sparkContext.makeRDD(List(1,2,3,4))
val dataRDD2 = sparkContext.makeRDD(List(3,4,5,6))
val dataRDD = dataRDD1.subtract(dataRDD2)
```



思考一个问题：如果两个RDD 数据类型不一致怎么办？

16) zip

➤ 函数签名

```
def zip[U: ClassTag](other: RDD[U]): RDD[(T, U)]
```

➤ 函数说明

将两个 RDD 中的元素，以键值对的形式进行合并。其中，键值对中的Key 为第 1 个 RDD 中的元素，Value 为第 2 个 RDD 中的相同位置的元素。

```
val dataRDD1 = sparkContext.makeRDD(List(1,2,3,4))
val dataRDD2 = sparkContext.makeRDD(List(3,4,5,6))
val dataRDD = dataRDD1.zip(dataRDD2)
```



思考一个问题：如果两个RDD 数据类型不一致怎么办？



思考一个问题：如果两个RDD 数据分区不一致怎么办？



思考一个问题：如果两个RDD 分区数据数量不一致怎么办？

● Key - Value 类型

17) partitionBy

➤ 函数签名

```
def partitionBy(partitioner: Partitioner): RDD[(K, V)]
```

➤ 函数说明

将数据按照指定Partitioner 重新进行分区。Spark 默认的分区器是HashPartitioner

```
val rdd: RDD[(Int, String)] =
  sc.makeRDD(Array((1, "aaa"), (2, "bbb"), (3, "ccc")), 3)
import org.apache.spark.HashPartitioner
```

```
val rdd2: RDD[(Int, String)] =  
  rdd.partitionBy(new HashPartitioner(2))
```



思考一个问题：如果重分区的分区器和当前RDD的分区器一样怎么办？



思考一个问题：Spark 还有其他分区器吗？



思考一个问题：如果想按照自己的方法进行数据分区怎么办？



思考一个问题：哪那么多问题？

18) reduceByKey

➤ 函数签名

```
def reduceByKey(func: (V, V) => V): RDD[(K, V)]
```

```
def reduceByKey(func: (V, V) => V, numPartitions: Int): RDD[(K, V)]
```

➤ 函数说明

可以将数据按照相同的Key 对 Value 进行聚合

```
val dataRDD1 = sparkContext.makeRDD(List(("a", 1), ("b", 2), ("c", 3)))  
val dataRDD2 = dataRDD1.reduceByKey(_+_)  
val dataRDD3 = dataRDD1.reduceByKey(_+_ , 2)
```

❖ 小功能：WordCount

19) groupByKey

➤ 函数签名

```
def groupByKey(): RDD[(K, Iterable[V])]
```

```
def groupByKey(numPartitions: Int): RDD[(K, Iterable[V])]
```

```
def groupByKey(partitioner: Partitioner): RDD[(K, Iterable[V])]
```

➤ 函数说明

将数据源的数据根据 key 对 value 进行分组

```
val dataRDD1 =  
  sparkContext.makeRDD(List(("a", 1), ("b", 2), ("c", 3)))  
val dataRDD2 = dataRDD1.groupByKey()  
val dataRDD3 = dataRDD1.groupByKey(2)  
val dataRDD4 = dataRDD1.groupByKey(new HashPartitioner(2))
```



思考一个问题：reduceByKey 和 groupByKey 的区别？

从 shuffle 的角度：reduceByKey 和 groupByKey 都存在 shuffle 的操作，但是reduceByKey 可以在 shuffle 前对分区内相同 key 的数据进行预聚合（combine）功能，这样会减少落盘的数据量，而groupByKey 只是进行分组，不存在数据量减少的问题，reduceByKey 性能比较高。

从功能的角度：reduceByKey 其实包含分组和聚合的功能。GroupByKey 只能分组，不能聚合，所以在分组聚合的场合下，推荐使用 reduceByKey，如果仅仅是分组而不需要聚合。那么还是只能使用groupByKey

❖ 小功能：WordCount

20) aggregateByKey

➤ 函数签名

```
def aggregateByKey[U: ClassTag](zeroValue: U)(seqOp: (U, V) => U,
    combOp: (U, U) => U): RDD[(K, U)]
```

➤ 函数说明

将数据根据不同的规则进行分区内计算和分区间计算

```
val dataRDD1 =
  sparkContext.makeRDD(List(("a",1), ("b",2), ("c",3)))
val dataRDD2 =
  dataRDD1.aggregateByKey(0) (_+_ , _+_)
```

❖ 取出每个分区内相同 key 的最大值然后分区间相加

```
// TODO : 取出每个分区内相同 key 的最大值然后分区间相加
// aggregateByKey 算子是函数柯里化，存在两个参数列表
// 1. 第一个参数列表中的参数表示初始值
// 2. 第二个参数列表中含有两个参数
// 2.1 第一个参数表示分区内的计算规则
// 2.2 第二个参数表示分区间计算规则
val rdd =
  sc.makeRDD(List( ("a",1), ("a",2), ("c"
    ,3),
    ("b",4), ("c",5), ("c",6)
  ),2)
// 0: ("a",1), ("a",2), ("c",3) => (a,10) (c,10)
//                               => (a,10) (b,10) (c,20)
// 1: ("b",4), ("c",5), ("c",6) => (b,10) (c,10)

val resultRDD =
  rdd.aggregateByKey(10) (
    (x, y) => math.max(x,y),
    (x, y) => x + y
  )

resultRDD.collect().foreach(println)
```




思考一个问题：分区内计算规则和分区间计算规则相同怎么办？（WordCount）

21) foldByKey

➤ 函数签名

```
def foldByKey(zeroValue: V)(func: (V, V) => V): RDD[(K, V)]
```

➤ 函数说明

当分区内计算规则和分区间计算规则相同时，`aggregateByKey` 就可以简化为 `foldByKey`

```
val dataRDD1 = sparkContext.makeRDD(List(("a", 1), ("b", 2), ("c", 3)))
val dataRDD2 = dataRDD1.foldByKey(0)(_+_)
```

22) combineByKey

➤ 函数签名

```
def combineByKey[C](createCombiner: V => C,
mergeValue: (C, V) => C,
mergeCombiners: (C, C) => C): RDD[(K, C)]
```

➤ 函数说明

最通用的对key-value 型 rdd 进行聚集操作的聚集函数（aggregation function）。类似于 `aggregate()`，`combineByKey()` 允许用户返回值的类型与输入不一致。

小练习：将数据 `List(("a", 88), ("b", 95), ("a", 91), ("b", 93), ("a", 95), ("b", 98))` 求每个 key 的平均值

```
val list: List[(String, Int)] = List(("a", 88), ("b", 95), ("a", 91), ("b", 93),
("a", 95), ("b", 98))
val input: RDD[(String, Int)] = sc.makeRDD(list, 2)

val combineRdd: RDD[(String, (Int, Int))] =
  input.combineByKey((_, 1),
    (acc: (Int, Int), v) => (acc._1 + v, acc._2 + 1),
    (acc1: (Int, Int), acc2: (Int, Int)) => (acc1._1 + acc2._1, acc1._2 + acc2._2)
  )
```



思考一个问题：`reduceByKey`、`foldByKey`、`aggregateByKey`、`combineByKey` 的区别？

`reduceByKey`: 相同 key 的第一个数据不进行任何计算，分区内和分区间计算规则相同

`FoldByKey`: 相同 key 的第一个数据和初始值进行分区内计算，分区内和分区间计算规则相同

AggregateByKey: 相同 key 的第一个数据和初始值进行分区内计算，分区内和分区间计算规则可以不相同

CombineByKey: 当计算时，发现数据结构不满足要求时，可以让第一个数据转换结构。分区内和分区间计算规则不相同。

23) sortByKey

➤ 函数签名

```
def sortByKey(ascending: Boolean = true, numPartitions: Int = self.partitions.length)
  : RDD[(K, V)]
```

➤ 函数说明

在一个(K,V)的 RDD 上调用，K 必须实现 **Ordered** 接口(特质)，返回一个按照 key 进行排序的

```
val dataRDD1 = sparkContext.makeRDD(List(("a",1), ("b",2), ("c",3)))
val sortRDD1: RDD[(String, Int)] = dataRDD1.sortByKey(true)
val sortRDD1: RDD[(String, Int)] = dataRDD1.sortByKey(false)
```

❖ 小功能：设置 key 为自定义类User

24) join

➤ 函数签名

```
def join[W](other: RDD[(K, W)]): RDD[(K, (V, W))]
```

➤ 函数说明

在类型为(K,V)和(K,W)的 RDD 上调用，返回一个相同 key 对应的所有元素连接在一起的 (K,(V,W))的 RDD

```
val rdd: RDD[(Int, String)] = sc.makeRDD(Array((1, "a"), (2, "b"), (3, "c")))
val rdd1: RDD[(Int, Int)] = sc.makeRDD(Array((1, 4), (2, 5), (3, 6)))
rdd.join(rdd1).collect().foreach(println)
```



思考一个问题：如果 key 存在不相等呢？

25) leftOuterJoin

➤ 函数签名

```
def leftOuterJoin[W](other: RDD[(K, W)]): RDD[(K, (V, Option[W]))]
```

➤ 函数说明

类似于 SQL 语句的左外连接

```
val dataRDD1 = sparkContext.makeRDD(List(("a",1), ("b",2), ("c",3)))
val dataRDD2 = sparkContext.makeRDD(List(("a",1), ("b",2), ("c",3)))

val rdd: RDD[(String, (Int, Option[Int]))] = dataRDD1.leftOuterJoin(dataRDD2)
```

26) cogroup

➤ 函数签名

```
def cogroup[W](other: RDD[(K, W)]): RDD[(K, (Iterable[V], Iterable[W]))]
```

➤ 函数说明

在类型为(K,V)和(K,W)的RDD 上调用，返回一个(K,(Iterable<V>,Iterable<W>))类型的 RDD

```
val dataRDD1 = sparkContext.makeRDD(List(("a",1), ("a",2), ("c",3)))
val dataRDD2 = sparkContext.makeRDD(List(("a",1), ("c",2), ("c",3)))

val value: RDD[(String, (Iterable[Int], Iterable[Int]))] =
dataRDD1.cogroup(dataRDD2)
```

5.1.4.4 案例实操

1) 数据准备

agent.log: 时间戳，省份，城市，用户，广告，中间字段使用空格分隔。

2) 需求描述

统计出**每一个省份每个广告被点击数量**排行的 Top3

3) 需求分析

4) 功能实现

5.1.4.5 RDD 行动算子

1) **reduce**

➤ 函数签名

def reduce(f: (T, T) => T): T

➤ 函数说明

聚合RDD中的所有元素，先聚合分区内数据，再聚合分区间数据

```
val rdd: RDD[Int] = sc.makeRDD(List(1,2,3,4))

// 聚合数据
val reduceResult: Int = rdd.reduce(_+_)
```

2) **collect**

➤ 函数签名

def collect(): Array[T]

➤ 函数说明

在驱动程序中，以数组Array的形式返回数据集的所有元素

```
val rdd: RDD[Int] = sc.makeRDD(List(1,2,3,4))

// 收集数据到 Driver
rdd.collect().foreach(println)
```

3) **count**

➤ 函数签名

def count(): Long

➤ 函数说明

返回RDD中元素的个数

```
val rdd: RDD[Int] = sc.makeRDD(List(1,2,3,4))

// 返回 RDD 中元素的个数
val countResult: Long = rdd.count()
```

4) **first**

➤ 函数签名

def first(): T

➤ 函数说明

返回RDD中的第一个元素

```
val rdd: RDD[Int] = sc.makeRDD(List(1,2,3,4))

// 返回 RDD 中元素的个数
val firstResult: Int = rdd.first()
println(firstResult)
```

5) take

➤ 函数签名

```
def take(num: Int): Array[T]
```

➤ 函数说明

返回一个由RDD的前n个元素组成的数组

```
val rdd: RDD[Int] = sc.makeRDD(List(1,2,3,4))

// 返回 RDD 中元素的个数
val takeResult: Array[Int] = rdd.take(2)
println(takeResult.mkString(","))
```

6) takeOrdered

➤ 函数签名

```
def takeOrdered(num: Int)(implicit ord: Ordering[T]): Array[T]
```

➤ 函数说明

返回该RDD排序后的前n个元素组成的数组

```
val rdd: RDD[Int] = sc.makeRDD(List(1,3,2,4))

// 返回 RDD 中元素的个数
val result: Array[Int] = rdd.takeOrdered(2)
```

7) aggregate

➤ 函数签名

```
def aggregate[U: ClassTag](zeroValue: U)(seqOp: (U, T) => U, combOp: (U, U) => U): U
```

➤ 函数说明

分区的数据通过[初始值](#)和分区内的数据进行聚合，然后再和[初始值](#)进行分区间的数据聚合

```
val rdd: RDD[Int] = sc.makeRDD(List(1, 2, 3, 4), 8)

// 将该 RDD 所有元素相加得到结果
//val result: Int = rdd.aggregate(0)(_ + _, _ + _)
val result: Int = rdd.aggregate(10)(_ + _, _ + _)
```

8) fold

➤ 函数签名

```
def fold(zeroValue: T)(op: (T, T) => T): T
```

➤ 函数说明

折叠操作，aggregate 的简化版操作

```
val rdd: RDD[Int] = sc.makeRDD(List(1, 2, 3, 4))  
val foldResult: Int = rdd.fold(0)(_+_)
```

9) countByKey

➤ 函数签名

```
def countByKey(): Map[K, Long]
```

➤ 函数说明

统计每种 key 的个数

```
val rdd: RDD[(Int, String)] = sc.makeRDD(List((1, "a"), (1, "a"), (1, "a"), (2, "b"), (3, "c"), (3, "c")))  
  
// 统计每种 key 的个数  
val result: collection.Map[Int, Long] = rdd.countByKey()
```

10) save 相关算子

➤ 函数签名

```
def saveAsTextFile(path: String): Unit  
def saveAsObjectFile(path: String): Unit  
def saveAsSequenceFile(  
  path: String,  
  codec: Option[Class[_ <: CompressionCodec]] = None): Unit
```

➤ 函数说明

将数据保存到不同格式的文件中

```
// 保存成 Text 文件  
rdd.saveAsTextFile("output")  
  
// 序列化成对象保存到文件  
rdd.saveAsObjectFile("output1")  
  
// 保存成 Sequencefile 文件  
rdd.map((_, 1)).saveAsSequenceFile("output2")
```

11) **foreach**

➤ 函数签名

```
def foreach(f: T => Unit): Unit = withScope
    {val cleanF = sc.clean(f)
     sc.runJob(this, (iter: Iterator[T]) => iter.foreach(cleanF))
    }
```

➤ 函数说明

分布式遍历RDD 中的每一个元素，调用指定函数

```
val rdd: RDD[Int] = sc.makeRDD(List(1,2,3,4))

// 收集后打印
rdd.map(num=>num).collect().foreach(println)

println("*****")

// 分布式打印
rdd.foreach(println)
```


5.1.4.6 RDD 序列化

1) 闭包检查

从计算的角度，算子以外的代码都是在Driver端执行，算子里面的代码都是在Executor端执行。那么在scala的函数式编程中，就会导致算子内经常会用到算子外的数据，这样就形成了闭包的效果，如果使用的算子外的数据无法序列化，就意味着无法传值给Executor端执行，就会发生错误，所以需要在执行任务计算前，检测闭包内的对象是否可以序列化，这个操作我们称之为闭包检测。Scala2.12版本后闭包编译方式发生了改变

2) 序列化方法和属性

从计算的角度，算子以外的代码都是在Driver端执行，算子里面的代码都是在Executor端执行，看如下代码：

```
object serializable02_function {  
    def main(args: Array[String]): Unit = {  
        //1.创建 SparkConf 并设置 App 名称  
        val conf: SparkConf = new SparkConf().setAppName("SparkCoreTest").setMaster("local[*]")  
  
        //2.创建 SparkContext，该对象是提交 Spark App 的入口  
        val sc: SparkContext = new SparkContext(conf)  
  
        //3.创建一个 RDD  
        val rdd: RDD[String] = sc.makeRDD(Array("hello world", "hello spark", "hive", "atguigu"))  
  
        //3.1 创建一个 Search 对象  
        val search = new Search("hello")  
  
        //3.2 函数传递，打印：ERROR Task not serializable  
        search.getMatch1(rdd).collect().foreach(println)  
  
        //3.3 属性传递，打印：ERROR Task not serializable  
        search.getMatch2(rdd).collect().foreach(println)  
  
        //4.关闭连接  
        sc.stop()  
    }  
}  
  
class Search(query:String) extends Serializable  
  
    {def isMatch(s: String): Boolean = {  
        s.contains(query)  
    }  
  
    // 函数序列化案例  
    def getMatch1 (rdd: RDD[String]): RDD[String] = {  
        //rdd.filter(this.isMatch)  
        rdd.filter(isMatch)  
    }  
}
```

```
// 属性序列化案例
def getMatch2(rdd: RDD[String]): RDD[String] = {
  //rdd.filter(x => x.contains(this.query))
  rdd.filter(x => x.contains(query))
  //val q = query
  //rdd.filter(x => x.contains(q))
}
}
```

3) Kryo 序列化框架

参考地址: <https://github.com/EsotericSoftware/kryo>

Java 的序列化能够序列化任何的类。但是**比较重**（字节多），序列化后，对象的提交也比较大。Spark 出于性能的考虑，Spark2.0 开始支持另外一种Kryo 序列化机制。Kryo 速度是 Serializable 的 10 倍。当 RDD 在 Shuffle 数据的时候，简单数据类型、数组和字符串类型已经在 Spark 内部使用 Kryo 来序列化。

注意：即使使用Kryo 序列化，也要继承Serializable 接口。

```
object serializable_Kryo {

  def main(args: Array[String]): Unit =

    { val conf: SparkConf = new

      SparkConf()
        .setAppName("SerDemo")
        .setMaster("local[*]")
        // 替换默认的序列化机制
        .set("spark.serializer",
"org.apache.spark.serializer.KryoSerializer")
        // 注册需要使用 kryo 序列化的自定义类
        .registerKryoClasses(Array(classOf[Searcher]))

      val sc = new SparkContext(conf)

      val rdd: RDD[String] = sc.makeRDD(Array("hello world", "hello atguigu",
"atguigu", "hahah"), 2)

      val searcher = new Searcher("hello")
      val result: RDD[String] = searcher.getMatchedRDD1(rdd)

      result.collect.foreach(println)
    }
}

case class Searcher(val query: String) {

  def isMatch(s: String) =
    {s.contains(query)
    }

  def getMatchedRDD1(rdd: RDD[String]) =
    {rdd.filter(isMatch)
    }

  def getMatchedRDD2(rdd: RDD[String]) =
    {val q = query
    rdd.filter(_.contains(q))
    }
}
```

5.1.4.7 RDD 依赖关系

1) RDD 血缘关系

RDD 只支持粗粒度转换，即在大量记录上执行的单个操作。将创建 RDD 的一系列Lineage（血统）记录下来，以便恢复丢失的分区。RDD 的 Lineage 会记录RDD 的元数据信息和转换行为，当该RDD的部分分区数据丢失时，它可以根据这些信息来重新运算和恢复丢失的数据分区。

```
val fileRDD: RDD[String] = sc.textFile("input/1.txt")
println(fileRDD.toDebugString)
println("-----")

val wordRDD: RDD[String] = fileRDD.flatMap(_.split(" "))
println(wordRDD.toDebugString)
println("-----")

val mapRDD: RDD[(String, Int)] = wordRDD.map(_._1)
println(mapRDD.toDebugString)
println("-----")

val resultRDD: RDD[(String, Int)] = mapRDD.reduceByKey(_+_ )
println(resultRDD.toDebugString)

resultRDD.collect()
```

2) RDD 依赖关系

这里所谓的依赖关系，其实就是两个相邻 RDD 之间的关系

```
val sc: SparkContext = new SparkContext(conf)

val fileRDD: RDD[String] = sc.textFile("input/1.txt")
println(fileRDD.dependencies)
println("-----")

val wordRDD: RDD[String] = fileRDD.flatMap(_.split(" "))
println(wordRDD.dependencies)
println("-----")

val mapRDD: RDD[(String, Int)] = wordRDD.map(_._1)
println(mapRDD.dependencies)
println("-----")

val resultRDD: RDD[(String, Int)] = mapRDD.reduceByKey(_+_ )
println(resultRDD.dependencies)

resultRDD.collect()
```

3) RDD 窄依赖

窄依赖表示每一个父(上游)RDD 的 Partition 最多被子（下游）RDD 的一个 Partition 使用，窄依赖我们形象的比喻为独生子女。

```
class OneToOneDependency[T](rdd: RDD[T]) extends NarrowDependency[T](rdd)
```

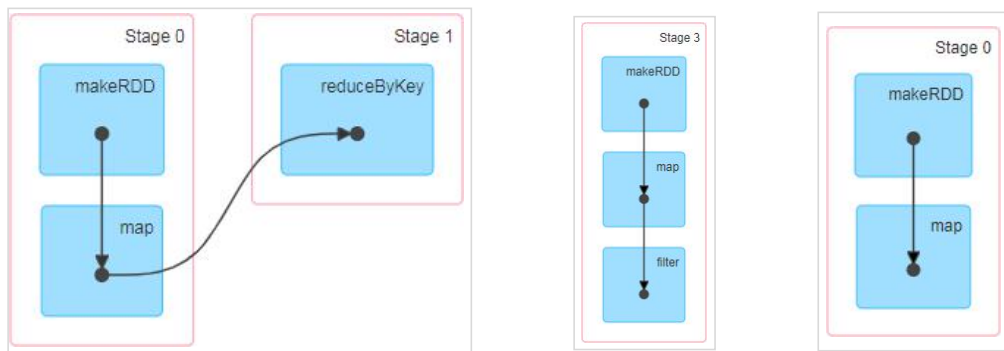
4) RDD 宽依赖

宽依赖表示同一个父（上游）RDD 的 Partition 被多个子（下游）RDD 的 Partition 依赖，会引起 Shuffle，总结：宽依赖我们形象的比喻为多生。

```
class ShuffleDependency[K: ClassTag, V: ClassTag, C:
  ClassTag](@transient private val _rdd: RDD[_ <: Product2[K,
    V]],
  val partitioner: Partitioner,
  val serializer: Serializer = SparkEnv.get.serializer,
  val keyOrdering: Option[Ordering[K]] = None,
  val aggregator: Option[Aggregator[K, V, C]] = None,
  val mapSideCombine: Boolean = false)
```

5) RDD 阶段划分

DAG（Directed Acyclic Graph）有向无环图是由点和线组成的拓扑图形，该图形具有方向，不会闭环。例如，DAG 记录了 RDD 的转换过程和任务的阶段。



6) RDD 阶段划分源码

```
try {
  // New stage creation may throw an exception if, for example, jobs are run on a
  // HadoopRDD whose underlying HDFS files have been deleted.
  finalStage = createResultStage(finalRDD, func, partitions, jobId, callSite)
} catch {
  case e: Exception =>
    logWarning("Creating new stage failed due to exception - job: " + jobId, e)
    listener.jobFailed(e)
    return
}

.....

private def
  createResultStage(rdd:
    RDD[_],
    func: (TaskContext, Iterator[_]) => _,
    partitions: Array[Int],
    jobId: Int,
    callSite: CallSite): ResultStage = {
  val parents = getOrCreateParentStages(rdd, jobId)
  val id = nextStageId.getAndIncrement()
  val stage = new ResultStage(id, rdd, func, partitions, parents, jobId, callSite)
  stageIdToStage(id) = stage
  updateJobIdStageIdMaps(jobId, stage)
  stage
}
```

```
.....

private def getOrCreateParentStages(rdd: RDD[_], firstJobId: Int): List[Stage]
= {
  getShuffleDependencies(rdd).map { shuffleDep =>
    getOrCreateShuffleMapStage(shuffleDep, firstJobId)
  }.toList
}

.....

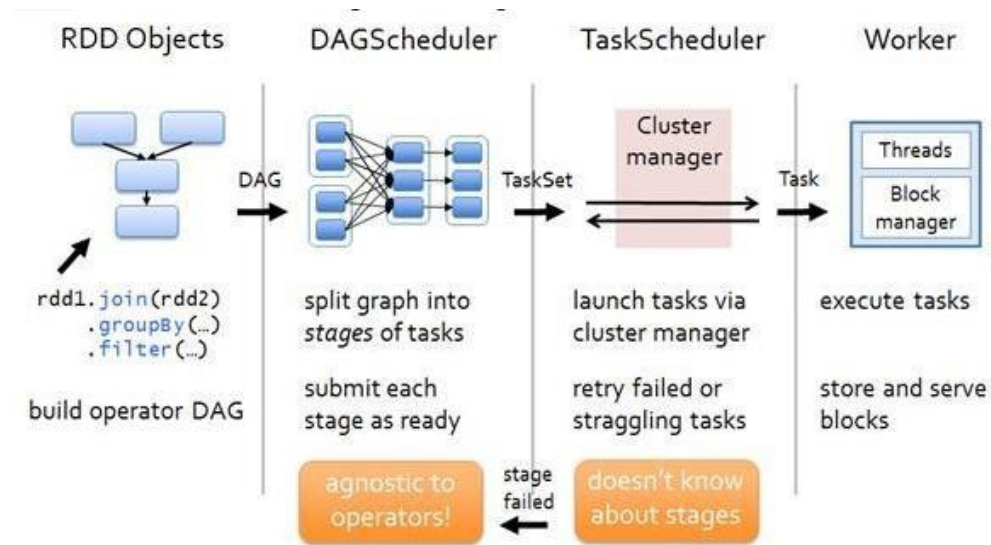
private[scheduler] def getShuffleDependencies(
  rdd: RDD[_]): HashSet[ShuffleDependency[_, _, _]] =
{val parents = new HashSet[ShuffleDependency[_, _, _]]
val visited = new HashSet[RDD[_]]
val waitingForVisit = new Stack[RDD[_]]
waitingForVisit.push(rdd)
while (waitingForVisit.nonEmpty)
{ val toVisit =
  waitingForVisit.pop() if
  (!visited(toVisit)) {
    visited += toVisit
    toVisit.dependencies.foreach {
      case shuffleDep: ShuffleDependency[_, _, _] =>
        parents += shuffleDep
      case dependency =>
        waitingForVisit.push(dependency.rdd)
    }
  }
}
}
parents
```

7) RDD 任务划分

RDD 任务切分中间分为：Application、Job、Stage 和 Task

- Application: 初始化一个 SparkContext 即生成一个Application;
- Job: 一个 Action 算子就会生成一个Job;
- Stage: Stage 等于宽依赖(ShuffleDependency)的个数加 1;
- Task: 一个 Stage 阶段中，最后一个RDD 的分区个数就是Task 的个数。

注意：Application->Job->Stage->Task 每一层都是 1 对 n 的关系。



8) RDD 任务划分源码

```
val tasks: Seq[Task[_]] = try
{stage match {
  case stage: ShuffleMapStage =>
    partitionsToCompute.map { id =>
      val locs = taskIdToLocations(id)
      val part = stage.rdd.partitions(id)
      new ShuffleMapTask(stage.id, stage.latestInfo.attemptId,
        taskBinary, part, locs, stage.latestInfo.taskMetrics, properties,
        Option(jobId),
        Option(sc.applicationId), sc.applicationAttemptId)
    }

  case stage: ResultStage =>
    partitionsToCompute.map { id =>
      val p: Int = stage.partitions(id)
      val part = stage.rdd.partitions(p)
      val locs = taskIdToLocations(id)
      new ResultTask(stage.id, stage.latestInfo.attemptId,
        taskBinary, part, locs, id, properties, stage.latestInfo.taskMetrics,
        Option(jobId), Option(sc.applicationId), sc.applicationAttemptId)
    }
}
}

.....

val partitionsToCompute: Seq[Int] = stage.findMissingPartitions()

.....

override def findMissingPartitions(): Seq[Int] =
{mapOutputTrackerMaster
  .findMissingPartitions(shuffleDep.shuffleId)
  .getOrElse(0 until numPartitions)
}
```

5.1.4.8 RDD 持久化

1) RDD Cache 缓存

RDD 通过 Cache 或者 Persist 方法将前面的计算结果缓存，默认情况下会把数据以缓存在 JVM 的堆内存中。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 算子时，该RDD 将会被缓存在计算节点的内存中，并供后面重用。

```
// cache 操作会增加血缘关系，不改变原有的血缘关系
println(wordToOneRdd.toDebugString)

// 数据缓存。
wordToOneRdd.cache()

// 可以更改存储级别
//mapRdd.persist(StorageLevel.MEMORY_AND_DISK_2)
```

存储级别

```
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(true, true, true, false, 1)
```

级 别	使用的 空间	CPU 时间	是否在 内存中	是否在 磁盘上	备 注
MEMORY_ONLY	高	低	是	否	
MEMORY_ONLY_SER	低	高	是	否	
MEMORY_AND_DISK	高	中等	部分	部分	如果数据在内存中放不下，则溢写到磁盘上
MEMORY_AND_DISK_SER	低	高	部分	部分	如果数据在内存中放不下，则溢写到磁盘上。在内存中存放序列化后的数据
DISK_ONLY	低	高	否	是	

缓存有可能丢失，或者存储于内存的数据由于内存不足而被删除，RDD 的缓存容错机制保证了即使缓存丢失也能保证计算的正确执行。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。

Spark 会自动对一些 Shuffle 操作的中间数据做持久化操作(比如：reduceByKey)。这样做的目的是为了当一个节点 Shuffle 失败了避免重新计算整个输入。但是，在实际使用的时候，如果想重用数据，仍然建议调用 persist 或 cache。

2) RDD CheckPoint 检查点

所谓的检查点其实就是通过将RDD 中间结果写入磁盘

由于血缘依赖过长会造成容错成本过高，这样就不如在中间阶段做检查点容错，如果检查点之后有节点出现问题，可以从检查点开始重做血缘，减少了开销。

对 RDD 进行 checkpoint 操作并不会马上被执行，必须执行 Action 操作才能触发。

```
// 设置检查点路径
sc.setCheckpointDir("./checkpoint1")

// 创建一个 RDD，读取指定位置文件:hello atguigu atguigu
val lineRdd: RDD[String] = sc.textFile("input/1.txt")

// 业务逻辑
val wordRdd: RDD[String] = lineRdd.flatMap(line => line.split(" "))

val wordToOneRdd: RDD[(String, Long)] = wordRdd.map
  {word => {
    (word, System.currentTimeMillis())
  }}

// 增加缓存, 避免再重新跑一个 job 做 checkpoint
wordToOneRdd.cache()
// 数据检查点: 针对 wordToOneRdd 做检查点计算
wordToOneRdd.checkpoint()

// 触发执行逻辑
wordToOneRdd.collect().foreach(println)
```

3) 缓存和检查点区别

- 1) Cache 缓存只是将数据保存起来，不切断血缘依赖。Checkpoint 检查点切断血缘依赖。
- 2) Cache 缓存的数据通常存储在磁盘、内存等地方，可靠性低。Checkpoint 的数据通常存储在 HDFS 等容错、高可用的文件系统，可靠性高。
- 3) 建议对 checkpoint() 的 RDD 使用 Cache 缓存，这样 checkpoint 的 job 只需从 Cache 缓存中读取数据即可，否则需要再从头计算一次 RDD。

5.1.4.9 RDD 分区器

Spark 目前支持Hash 分区和 Range 分区，和用户自定义分区。Hash 分区为当前的默认分区。分区器直接决定了RDD 中分区的个数、RDD 中每条数据经过Shuffle 后进入哪个分区，进而决定了Reduce 的个数。

- 只有Key-Value 类型的RDD 才有分区器，非 Key-Value 类型的RDD 分区的值是 None
- 每个RDD 的分区 ID 范围：0 ~ (numPartitions - 1)，决定这个值是属于那个分区的。

1) **Hash 分区**：对于给定的 key，计算其hashCode,并除以分区个数取余

```
class HashPartitioner(partitions: Int) extends Partitioner
{ require(partitions >= 0, s"Number of partitions ($partitions) cannot be negative.")

  def numPartitions: Int = partitions

  def getPartition(key: Any): Int = key match
  { case null => 0
    case _ => Utils.nonNegativeMod(key.hashCode, numPartitions)
  }

  override def equals(other: Any): Boolean = other match
  { case h: HashPartitioner =>
    h.numPartitions == numPartitions
    case _ =>
      false
  }

  override def hashCode: Int = numPartitions
```

2) **Range 分区**：将一定范围内的数据映射到一个分区中，尽量保证每个分区数据均匀，而且分区间有序

```
class RangePartitioner[K : Ordering : ClassTag,
V](partitions: Int,
  rdd: RDD[_ <: Product2[K, V]],
  private var ascending: Boolean = true)
  extends Partitioner {

  // We allow partitions = 0, which happens when sorting an empty RDD under the
  // default settings.
  require(partitions >= 0, s"Number of partitions cannot be negative but found $partitions.")

  private var ordering = implicitly[Ordering[K]]

  // An array of upper bounds for the first (partitions - 1) partitions
  private var rangeBounds: Array[K] = {
    ...
  }

  def numPartitions: Int = rangeBounds.length + 1

  private var binarySearch: ((Array[K], K) => Int) =
    CollectionsUtils.makeBinarySearch[K]
```

```
def getPartition(key: Any): Int =
  {val k = key.asInstanceOf[K]
   var partition = 0
   if (rangeBounds.length <= 128) {
     // If we have less than 128 partitions naive search
     while (partition < rangeBounds.length && ordering.gt(k,
rangeBounds(partition)))
       {partition += 1
        }
   } else {
     // Determine which binary search method to use only once.
     partition = binarySearch(rangeBounds, k)
     // binarySearch either returns the match location or -[insertion point]-1
     if (partition < 0) {
       partition = -partition-1
     }
     if (partition > rangeBounds.length)
       {partition = rangeBounds.length
        }
   }
   if (ascending)
     {partition
    } else {
     rangeBounds.length - partition
    }
  }

  override def equals(other: Any): Boolean = other match {
    ...
  }

  override def hashCode(): Int = {
    ...
  }

  @throws(classOf[IOException])
  private def writeObject(out: ObjectOutputStream): Unit =
  Utils.tryOrIOException {
    ...
  }

  @throws(classOf[IOException])
  private def readObject(in: ObjectInputStream): Unit = Utils.tryOrIOException
  {
    ...
  }
}
```

5.1.4.10 RDD 文件读取与保存

Spark 的数据读取及数据保存可以从两个维度来作区分：文件格式以及文件系统。

文件格式分为：text 文件、csv 文件、sequence 文件以及 Object 文件；

文件系统分为：本地文件系统、HDFS、HBASE 以及数据库。

➤ text 文件

```
// 读取输入文件
val inputRDD: RDD[String] = sc.textFile("input/1.txt")

// 保存数据
inputRDD.saveAsTextFile("output")
```

➤ sequence 文件

SequenceFile 文件是 Hadoop 用来存储二进制形式的 key-value 对而设计的一种平面文件(Flat File)。在 SparkContext 中，可以调用 sequenceFile[keyClass, valueClass](path)。

```
// 保存数据为 SequenceFile
dataRDD.saveAsSequenceFile("output")

// 读取 SequenceFile 文件
sc.sequenceFile[Int, Int]("output").collect().foreach(println)
```

➤ object 对象文件

对象文件是将对象序列化后保存的文件，采用 Java 的序列化机制。可以通过 objectFile[T: ClassTag](path) 函数接收一个路径，读取对象文件，返回对应的 RDD，也可以通过调用 saveAsObjectFile() 实现对对象文件的输出。因为是序列化所以要指定类型。

```
// 保存数据
dataRDD.saveAsObjectFile("output")

// 读取数据
sc.objectFile[Int]("output").collect().foreach(println)
```

5.2 累加器

5.2.1 实现原理

累加器用来把Executor 端变量信息聚合到Driver 端。在Driver 程序中定义的变量，在Executor 端的每个Task 都会得到这个变量的一份新的副本，每个 task 更新这些副本的值后，传回Driver 端进行 merge。

5.2.2 基础编程

5.2.2.1 系统累加器

```
val rdd = sc.makeRDD(List(1,2,3,4,5))
// 声明累加器
var sum = sc.longAccumulator("sum");
rdd.foreach(
  num => {
    // 使用累加器
    sum.add(num)
  }
)
// 获取累加器的值
println("sum = " + sum.value)
```

5.2.2.2 自定义累加器

```
// 自定义累加器
// 1. 继承 AccumulatorV2，并设定泛型
// 2. 重写累加器的抽象方法
class WordCountAccumulator extends AccumulatorV2[String, mutable.Map[String, Long]] {

  var map : mutable.Map[String, Long] = mutable.Map()

  // 累加器是否为初始状态
  override def isZero: Boolean =
    {map.isEmpty}
}

// 复制累加器
override def copy(): AccumulatorV2[String, mutable.Map[String, Long]] =
  {new WordCountAccumulator}

// 重置累加器
override def reset(): Unit =
  {map.clear()}

// 向累加器中增加数据 (In)
override def add(word: String): Unit = {
  // 查询 map 中是否存在相同的单词
  // 如果有相同的单词，那么单词的数量加 1
  // 如果没有相同的单词，那么在 map 中增加这个单词
  map(word) = map.getOrElse(word, 0L) + 1L
}
```

```
// 合并累加器
override def merge(other: AccumulatorV2[String, mutable.Map[String, Long]]):
Unit = {

  val map1 = map
  val map2 = other.value

  // 两个 Map 的合并
  map =
    map1.foldLeft(map2)(( in
      nerMap, kv ) => {
        innerMap(kv._1) = innerMap.getOrElse(kv._1, 0L) + kv._2
        innerMap
      }
    )
}

// 返回累加器的结果 (Out)
override def value: mutable.Map[String, Long] = map
```

5.3 广播变量

5.3.1 实现原理

广播变量用来高效分发较大的对象。向所有工作节点发送一个较大的只读值，以供一个或多个 Spark 操作使用。比如，如果你的应用需要向所有节点发送一个较大的只读查询表，广播变量用起来都很顺手。在多个并行操作中使用同一个变量，但是 Spark 会为每个任务分别发送。

5.3.2 基础编程

```
val rdd1 = sc.makeRDD(List( ("a",1), ("b", 2), ("c", 3), ("d", 4) ),4)
val list = List( ("a",4), ("b", 5), ("c", 6), ("d", 7) )
// 声明广播变量
val broadcast: Broadcast[List[(String, Int)]] = sc.broadcast(list)

val resultRDD: RDD[(String, (Int, Int))] = rdd1.map
{case (key, num) => {
  var num2 = 0
  // 使用广播变量
  for ((k, v) <- broadcast.value)
    {if (k == key) {
      num2 = v
    }
  }
  (key, (num, num2))
}
}
```

第6章 Spark 案例实操

在之前的学习中，我们已经学习了 Spark 的基础编程方式，接下来，我们看看在实际的工作中如何使用这些 API 实现具体的需求。这些需求是电商网站的真实需求，所以在实现功能前，咱们必须先将要数据准备好。

日期	用户ID	Session ID	页面ID	动作时间	搜索关键字
2019-05-05	85	e2eef06e-beaa-4b49-acaf-38e057e1cd6e	3,2	2019-05-05 02:54:16	苹果
2019-05-05	60	be4f888f-9c0b-44c7-8ecc-1865379b85d9	6,2	2019-05-05 01:18:12	18,84
2019-05-05	57	fcelfdf7-4678-4349-9a70-edee5a5e580f	47,	2019-05-05 05:30:32	,,-1,-1,1-2-3,1-2-3,11
2019-05-05	34	aa7ff24e-d6fc-4c81-99e7-9756e9a7e18d	32,	2019-05-05 05:03:28	,,-1,-1,1-2-3,1-2-3,12
2019-05-05	60	be4f888f-9c0b-44c7-8ecc-1865379b85d9	23,	2019-05-05 01:20:00	点击品类ID和产品ID
2019-05-05	60	be4f888f-9c0b-44c7-8ecc-1865379b85d9	11,	2019-05-05 01:29:37	17,73
2019-05-05	85	e2eef06e-beaa-4b49-acaf-38e057e1cd6e	8,	2019-05-05 01:38:17	,,5,95,,26
2019-05-05	85	e2eef06e-beaa-4b49-acaf-38e057e1cd6e	18,	2019-05-05 01:41:04	,,5,77,,13
2019-05-05	85	e2eef06e-beaa-4b49-acaf-38e057e1cd6e	11,	2019-05-05 01:48:43	,,2,87,,13
2019-05-05	85	e2eef06e-beaa-4b49-acaf-38e057e1cd6e	46,	2019-05-05 01:52:50	支付品类ID和产品ID
2019-05-05	85	e2eef06e-beaa-4b49-acaf-38e057e1cd6e	14,	2019-05-05 01:53:23	,,14,62,,9
2019-05-05	85	e2eef06e-beaa-4b49-acaf-38e057e1cd6e	8,	2019-05-05 02:02:38	,,-1,-1,1-2-3,1-2-3,20
2019-05-05	85	e2eef06e-beaa-4b49-acaf-38e057e1cd6e	48,	2019-05-05 02:08:07	,,7,13,,5
2019-05-05	85	e2eef06e-beaa-4b49-acaf-38e057e1cd6e	25,	2019-05-05 02:12:29	,,18,27,,9
2019-05-05	85	e2eef06e-beaa-4b49-acaf-38e057e1cd6e	36,	2019-05-05 02:20:02	,,-1,-1,1,城市ID-2-3,820
2019-05-05	85	e2eef06e-beaa-4b49-acaf-38e057e1cd6e	19,	2019-05-05 02:23:10	,,17,-1,-1,,21
2019-05-05	85	e2eef06e-beaa-4b49-acaf-38e057e1cd6e	38,	2019-05-05 02:23:51	,,15,16,,21
2019-05-05	85	e2eef06e-beaa-4b49-acaf-38e057e1cd6e	12,	2019-05-05 02:31:32	,,20,27,,5

上面的数据图是从数据文件中截取的一部分内容，表示为电商网站的用户行为数据，主要包含用户的 4 种行为：搜索，点击，下单，支付。数据规则如下：

- 数据文件中每行数据采用下划线分隔数据
- 每一行数据表示用户的一次行为，这个行为只能是 4 种行为的一种
- 如果搜索关键字为 null,表示数据不是搜索数据
- 如果点击的品类 ID 和产品 ID 为-1，表示数据不是点击数据
- 针对于下单行为，一次可以下单多个商品，所以品类 ID 和产品 ID 可以是多个，id 之间采用逗号分隔，如果本次不是下单行为，则数据采用 null 表示
- 支付行为和下单行为类似

详细字段说明：

编号	字段名称	字段类型	字段含义
1	date	String	用户点击行为的日期
2	user_id	Long	用户的 ID
3	session_id	String	Session 的 ID
4	page_id	Long	某个页面的 ID
5	action_time	String	动作的时间点
6	search_keyword	String	用户搜索的关键词

关注公众号：大数据技术派，领取1024G资料，每日推送技术干货。

7	click_category_id	Long	某一个商品品类的 ID
8	click_product_id	Long	某一个商品的 ID
9	order_category_ids	String	一次订单中所有品类的 ID 集合
10	order_product_ids	String	一次订单中所有商品的 ID 集合
11	pay_category_ids	String	一次支付中所有品类的 ID 集合
12	pay_product_ids	String	一次支付中所有商品的 ID 集合
13	city_id	Long	城市 id

样例类：

```
//用户访问动作表
case class UserVisitAction(
    date: String, //用户点击行为的日期
    user_id: Long, //用户的 ID
    session_id: String, //Session 的 ID
    page_id: Long, //某个页面的 ID
    action_time: String, //动作的时间点
    search_keyword: String, //用户搜索的关键词
    click_category_id: Long, //某一个商品品类的 ID
    click_product_id: Long, //某一个商品的 ID
    order_category_ids: String, //一次订单中所有品类的 ID 集合
    order_product_ids: String, //一次订单中所有商品的 ID 集合
    pay_category_ids: String, //一次支付中所有品类的 ID 集合
    pay_product_ids: String, //一次支付中所有商品的 ID 集合
    city_id: Long
) //城市 id
```

6.1 需求 1: Top10 热门品类



6.1.1 需求说明

品类是指产品的分类，大型电商网站品类分多级，咱们的项目中品类只有一级，不同的公司可能对热门的定义不一样。我们按照每个品类的点击、下单、支付的量来统计热门品类。

鞋 点击数 下单数 支付数

衣服 点击数 下单数 支付数

关注公众号：大数据技术派，领取1024G资料。

电脑 点击数 下单数 支付数

例如，综合排名 = 点击数*20%+下单数*30%+支付数*50%

本项目需求优化为：先按照点击数排名，靠前的就排名高；如果点击数相同，再比较下单数；下单数再相同，就比较支付数。

6.1.2 实现方案一

6.1.2.1 需求分析

分别统计每个品类点击的次数，下单的次数和支付的次数：

（品类，点击总数）（品类，下单总数）（品类，支付总数）

6.1.2.2 需求实现

6.1.3 实现方案二

6.1.3.1 需求分析

一次性统计每个品类点击的次数，下单的次数和支付的次数：

（品类，（点击总数，下单总数，支付总数））

6.1.3.2 需求实现

6.1.4 实现方案三

6.1.4.1 需求分析

使用累加器的方式聚合数据

6.1.4.2 需求实现

6.2 需求 2：Top10 热门品类中每个品类的 Top10 活跃 Session 统计

6.2.1 需求说明

在需求一的基础上，增加每个品类用户 session 的点击统计

6.2.2 需求分析

6.2.3 功能实现

6.3 需求 3：页面单跳转换率统计

6.3.1 需求说明

1) 页面单跳转化率

计算页面单跳转化率，什么是页面单跳转换率，比如一个用户在一次 Session 过程中访问的页面路径 3,5,7,9,10,21，那么页面 3 跳到页面 5 叫一次单跳，7-9 也叫一次单跳，那么单跳转化率就是要统计页面点击的概率。

比如：计算 3-5 的单跳转化率，先获取符合条件的 Session 对于页面 3 的访问次数（PV）为 A，然后获取符合条件的 Session 中访问了页面 3 又紧接着访问了页面 5 的次数为 B，那么 B/A 就是 3-5 的页面单跳转化率。



2) 统计页面单跳转化率意义

产品经理和运营总监，可以根据这个指标，去尝试分析，整个网站，产品，各个页面的表现怎么样，是不是需要去优化产品的布局；吸引用户最终可以进入最后的支付页面。

数据分析师，可以此数据做更深一步的计算和分析。

企业管理层，可以看到整个公司的网站，各个页面的之间的跳转的表现如何，可以适当调整公司的经营战略或策略。

6.3.2 需求分析

6.3.3 功能实现

