四川大学网络空间安全研究院
CyberSecurity Research Institute, Sichuan University

2017 年 1-3 月份工作总结

版　本　号：　v1.0

项　目　组：　大数据平台项目组

文 档 作 者：　吴天雄

撰 写 日 期：　**2017** 年 **03** 月 **13** 号

评审负责人：

评 审 日 期：

四 川 大 学 网 络 空 间 安 全 研 究 院

目 录

第 1 章　Spark 应用启动过程分析

1.1　WordCount 程序

阅读 Spark 源码以 Spark 集群中运行 WordCount 应用顺序为根据，理解出 Spark 应用运行的流程，从而达到阅读源码的目的。

本 WordCount 源码如程序1.1所示:

```
1  package org.apache.spark.examples
2  import org.apache.spark.{SparkConf, SparkContext}
3  object HelloWorld {
4  def main(args: Array[String]): Unit = {
5      val scf = new SparkConf();
6      scf.setAppName("Spark Count")
7      val sc = new SparkContext(scf)
8      val scfclone = sc.getConf
9      val threshold = args(1).toInt
10     val tokenizde = sc.textFile(args(0)).flatMap(_.split(" "))
11     println("tokenizde: " + tokenizde)
12     val wordCount = tokenizde.map((_, 1)).reduceByKey(_ + _)
13     println("wordCount: " + wordCount)
14     val filtered = wordCount.filter(_._2 >= threshold)
15     val charCount = filtered.flatMap(_._1.toCharArray).map((_,1)).
           ↪ reduceByKey(_+_)
16     println(filtered.collect().mkString(","))
17   }
18 }
```

程序 1.1　基于 Spark 的 WordCount 程序

将输入案例 input.txt 放入 hdfs 中，已搭建好环境的 Spark 集群，打包好的 WordCount jar 包且提交到集群中的任意一台机器（本例中为 master）中。

1.2　提交 WordCount 程序

程序提交的整个过程如图1.1所示：

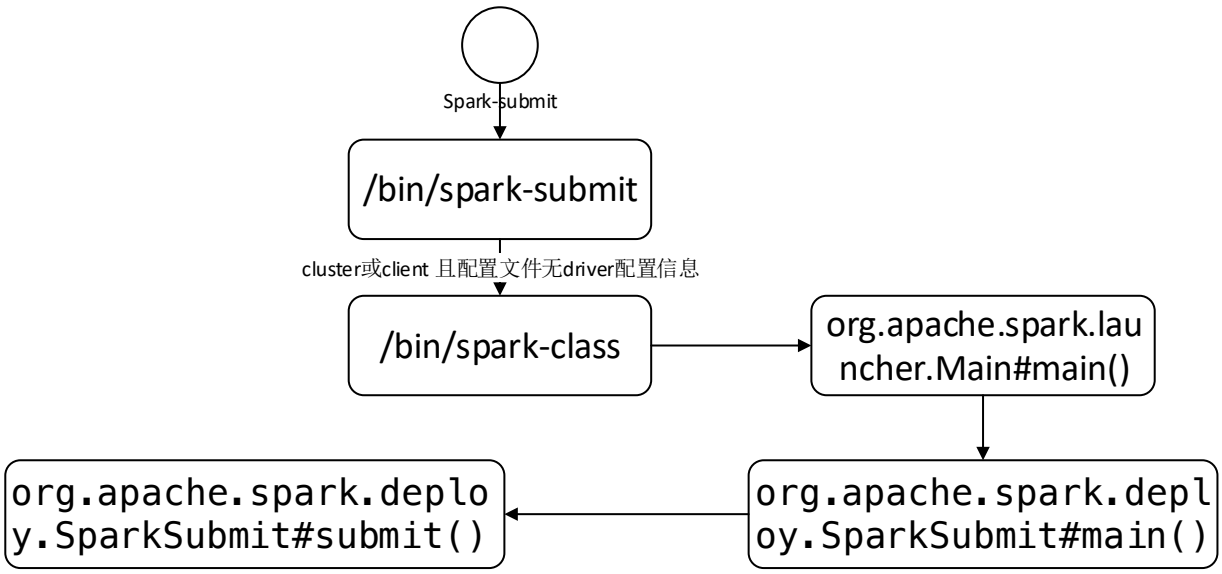


图 1.1　spark-submit 运行流程

在 master 中运行提交命令如下：

Shell 命令

```
/usr/bin/spark-submit
   HelloWorld-1.0-SNAPSHOT-jar-with-dependencies.jar
   /user/inputfile.txt 2
```

从这条命令可以看出使用的是 spark-submit shell 脚本，后面三个为参数，第一个为目录下的 WordCount jar 包，第二个为 hdfs 中的输入文件，第三个为 WordCount 中限定输出的 value 临界值。我们打开 spark-submit shell 脚本，脚本如所示。

spark-submit shell 脚本内容如程序1.2所示。

```
1 SPARK_HOME="$(cd "`dirname $(readlink -nf "$0")`"/..; pwd -P)"
2 export PYTHONHASHSEED=0
3 exec "SPARK_HOME"/bin/spark-class org.apache.spark.deploy.SparkSubmit "$@
    ↪ "
```

程序 1.2　spark-submit.sh 脚本

从程序1.2可以看出，spark-submit 执行流程是：首先找出 Spark 安装的根目录，然后再去执行 spark-class 这个 shell 脚本，参数为 org.apache.spark.deploy.SparkSubmit 以及提交命令后跟的所有参数。

spark-class 脚本先找到 master 的 java，遍历 Spark 安装目录下 lib 目录下的 jar 包，加入到 CLASSPATH 中，具体内容如下：

spark-class 的 CLASSPATH 值

```
/usr/hdp/current/spark-historyserver/conf/
/usr/hdp/2.4.0.0-169/spark/lib/spark-assembly-1.6.0.2.4.0.0-169-
hadoop2.7.1.2.4.0.0-169.jar
/usr/hdp/2.4.0.0-169/spark/lib/datanucleus-api-jdo-3.2.6.jar
/usr/hdp/2.4.0.0-169/spark/lib/datanucleus-core-3.2.10.jar
/usr/hdp/2.4.0.0-169/spark/lib/datanucleus-rdbms-3.2.9.jar
/usr/hdp/current/hadoop-client/conf/
```

spark-class 最后通过 Java -cp ⟨*classpath*⟩ org.apache.spark.launcher.Main 参数列表，开启 jvm 进程，执行的主类为 org.apache.spark.launcher.Main，传过来的参数有：

— org.apache.spark.deploy.SparkSubmi
— HelloWorld-1.0-SNAPSHOT-jar-with-dependencies.jar
— /user/inputfile.txt

最后启动的 Java 程序的命令如下：

spark-class 创建的 Java 虚拟机命令

```
java -Dhdp.version=2.4.0.0-169
-cp <classpath>
-Xms1g -Xmx1g
org.apache.spark.deploy.SparkSubmit
–master yarn
–deploy-mode client
HelloWorld-1.0-SNAPSHOT-jar-with-dependencies.jar
/user/inputfile.txt 2
```

1.3　准备 WordCount 提交程序运行环境

org.apache.spark.launcher.Main 代码片段如程序1.3所示。传入此类的 main 方法的参数有：

- org.apache.spark.deploy.SparkSubmit
- –master yarn-client

- HelloWorld-1.0-SNAPSHOT-jar-with-dependencies.jar
- /user/inputfile.txt

代码通过 SparkSubmitCommandBuilder 方法生成相应的 Java 命令，然后返回给 cmd，最后程序判断运行的主机是 Windows 还是 bash，打印到控制台，之后 spark-class 脚本重定位到这里并读取输出的字符流[①]，通过 exec 执行 java 命令。

```
1  public static void main(String[] argsArray) throws Exception {
2    List<String> args = new ArrayList<>(Arrays.asList(argsArray));
3    String className = args.remove(0);
4    AbstractCommandBuilder builder;
5    if (className.equals("org.apache.spark.deploy.SparkSubmit")) {
6    try {
7      builder = new SparkSubmitCommandBuilder(args);
8    } catch (IllegalArgumentException e) { ... }
9
10   Map<String, String> env = new HashMap<>();
11   List<String> cmd = builder.buildCommand(env);
12   if (isWindows()) {
13     System.out.println(prepareWindowsCommand(cmd, env));
14   } else {
15     // In bash, use NULL as the arg separator since it cannot be used in
         ↪ an argument.
16     List<String> bashCmd = prepareBashCommand(cmd, env);
17     for (String c : bashCmd) {
18       System.out.print(c);
19       System.out.print('\0');
20     }
21   }
22 }
```

程序 1.3　launcher.Main 程序代码片段

① launcher.Main 函数在 spark-class 脚本中执行，spark-class 脚本会读取 Main 的标准输出，将标准输出解析为启动程序命令。

1.4　进入 Spark-Submit 提交主类

前面的准备工作工作后最后调用 org.apache.spark.deploy.SparkSubmit 类的 main 方法，程序代码如程序1.4所示

```
1  def main(args: Array[String]): Unit = {
2  val appArgs = new SparkSubmitArguments(args)
3  if (appArgs.verbose) {
4  // scalastyle:off println
5     printStream.println(appArgs)
6  // scalastyle:on println
7  }
8  appArgs.action match {
9      case SparkSubmitAction.SUBMIT => submit(appArgs)
10     case SparkSubmitAction.KILL => kill(appArgs)
11     case SparkSubmitAction.REQUEST_STATUS => requestStatus(appArgs)
12   }
13 }
```

程序 1.4　SparkSubmit.scala 中 main 函数

首先第一步是构造参数；第二步是根据提供的参数匹配相应的操作。

1 构造提交参数方法

(a) 将提交脚本中的参数做个转换，像—master 等；

(b) 合并—conf 中和 spark-default.conf 中的参数；

(c) 删掉不是是 spark 系统的配置参数；

(d) 从环境脚本中读入默认配置参数并合并；

(e) 验证配置参数的有效性；

2 提交参数中的 action 默认为 SUBMIT，所以这步会执行 submit 方法。该方法中有两个较为重要的方法

SparkSubmit 调用的两个重要的方法

val (childArgs, childClasspath, sysProps, childMainClass) = prepareSubmitEnvironment(args)

runMain(childArgs, childClasspath, sysProps, childMainClass, args.verbose)

(a) 对于 prepareSubmitEnvironment 此方法返回一个较为重要的参数即 childMainClass，其会根据 deploy-mode 对应不同的实现，具体如图1.2所示

表 1.1　Yarn-Cluster 和 Yarn-Client 模式下的参数对比

参数名	Yarn-Cluster	Yarn-Client
childArgs		ArrayBuffer(/user/root/inputfile.txt, 2)
childClasspath	ArrayBuffer()	ArrayBuffer(file:/root/HelloWorld-1.0-SNAPSHOT-jar-with-dependencies.jar)
sysProps	很多，基本都是系统属性	spark.master->yarn-client
childMainClass	org.apache.spark.deploy.yarn.Clientcom.wtx.HelloWorld	

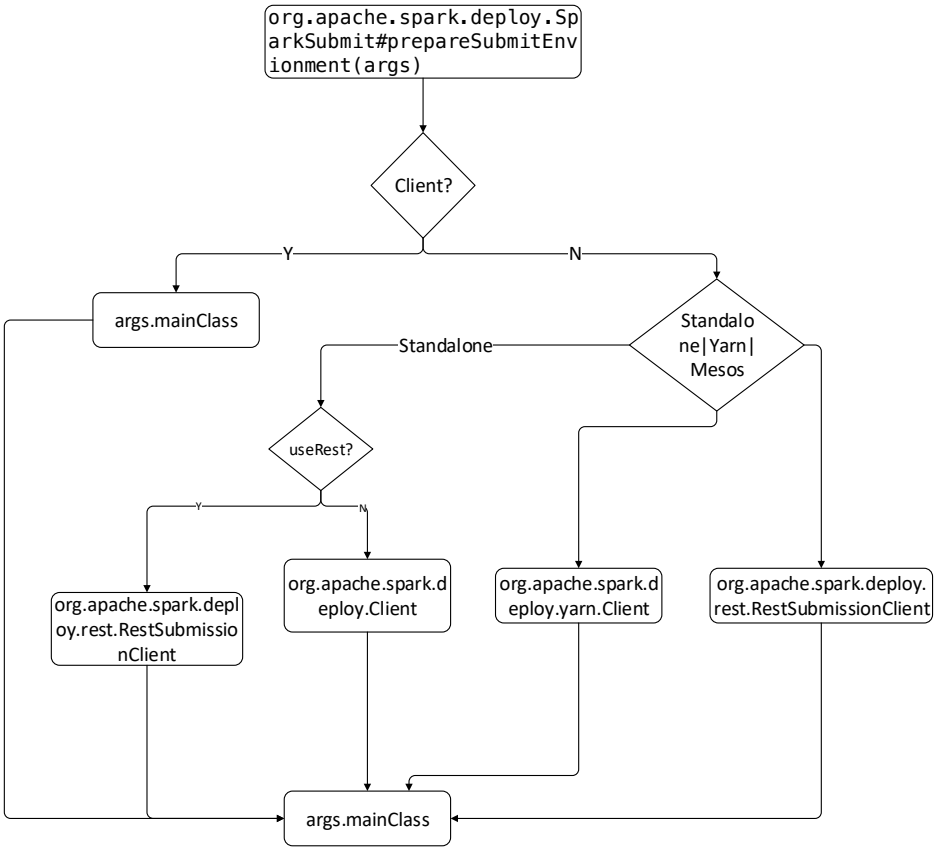


图 1.2　prepareSubmit 运行流程

为了验证自己分析的正确性，通过中间输出调试的方式，分别输出了在 yarn-cluster 和 yarn-client 模式下该方法返回的四元组值，命令为最前面的命令，其结果如表1.1所示 由结果可知，我们的分析是正确的。这里我们使用的 master 为 yarn，deploy-mode 为 cluster，所以 childMainClass 为 org.apache.spark.deploy.yarn.Client。

(b) 对于 runMain

i. 先检查查是否可见，是的话打印出各个参数的配置信息；

ii. 跟据 spark.driver.userClassPathFirst 这个属性确定类加载器，这个属性可以降低 Spark 依赖和用户依赖的冲突。它现在还是一个实验性的特征；

iii. 将 jar 包加入到类加载器的 classpath；

iv. 通过反射射的方式找到 mainClass，然后在获得其 main 方法，并通过 invoke 加载 main 方法，这里就是 org.apache.spark.deploy.yarn.Client 的 main 方法。

到这里 spark-submit 就执行完了，接下来将分析 org.apache.spark.deploy.yarn.Client 的 main 方法执行过程。

1.5　Yarn 代理的启动

由1.4节我们知道，其最后就是反射加载 org.apache.spark.deploy.yarn.Client 的 main 方法，其执行过程如图1.3所示

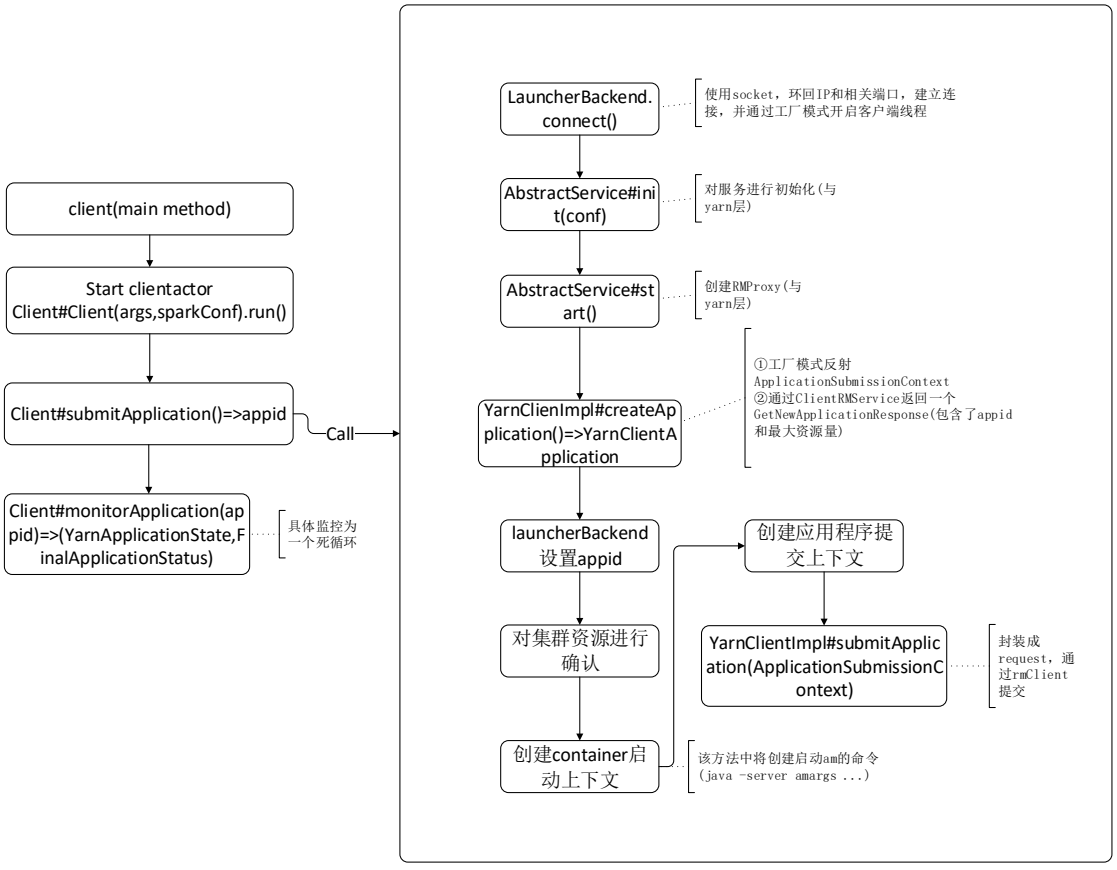


图 1.3　YarnClientMain 流程

main 函数代码如程序1.5所示

```
/**
* --name, com.wtx.HelloWorld,
* --jar, file:/root/HelloWorld-1.0-SNAPSHOT-jar-with-dependencies.jar,
* --class, com.wtx.HelloWorld,
* --arg, /user/root/inputfile.txt,
* --arg, 2
* */
def main(argStrings: Array[String]) {
  if (!sys.props.contains("SPARK_SUBMIT")) {
  logWarning("WARNING: This client is deprecated and will be removed in a
      ↪  " +
    "future version of Spark. Use ./bin/spark-submit with \"--master yarn
       ↪ \"")
  }
  System.setProperty("SPARK_YARN_MODE", "true")
  val sparkConf = new SparkConf
  val args = new ClientArguments(argStrings, sparkConf)
  if (!Utils.isDynamicAllocationEnabled(sparkConf)) {
    sparkConf.setIfMissing("spark.executor.instances", args.numExecutors.
       ↪ toString)
  }
  new Client(args, sparkConf).run()
}
```

程序 1.5　YarnClient 中 main 函数

这里面第一个执行的是 ClientArguments 方法，该方法中

— 对传入的参数进行解析，通过—标识符识别，像 jar、class、arg 之类的

— 获取 spark 环境中的默认配置参数，就是获取 spark.yarn.dist.files 和 spark.yarn.dist.archives 这两个

— 对已经解析出来的参数进行校验

之后就是 Client 中的主要方法，new Client(args, sparkConf).run()。

创建 Client 对象时，对一些变量如 yarnClient、amMemoryOverhead、executorMemoryOverhead、isClusterMode 等，其中 yarnClient 初始化为 YarnClientImpl 对

象。接着就是运行 run 方法。其代码如程序1.6所示

```
def run(): Unit = {
  this.appId = submitApplication()
  if (!launcherBackend.isConnected() && fireAndForget) {
      val report = getApplicationReport(appId)
      val state = report.getYarnApplicationState
      logInfo(s"Application report for $appId (state: $state)")
      logInfo(formatReportDetails(report))
      if (state == YarnApplicationState.FAILED || state ==
          ↪ YarnApplicationState.KILLED) {
      throw new SparkException(s"Application $appId finished with status:
          ↪  $state")
    }
  } else {
      val (yarnApplicationState, finalApplicationStatus) =
          ↪ monitorApplication(appId)
      if (yarnApplicationState == YarnApplicationState.FAILED ||
          finalApplicationStatus == FinalApplicationStatus.FAILED) {
         throw new SparkException(s"Application $appId finished with
            ↪ failed status")
      }
      if (yarnApplicationState == YarnApplicationState.KILLED ||
      finalApplicationStatus == FinalApplicationStatus.KILLED) {
          throw new SparkException(s"Application $appId is killed")
      }
      if (finalApplicationStatus == FinalApplicationStatus.UNDEFINED) {
          throw new SparkException(s"The final status of application
              ↪ $appId is undefined")
      }
  }
}
```

程序 1.6　YarnClient 中 run 函数

run 里第一个调用的方法就是 Client#submitApplication() 方法，此方法最终返

回的是与 yarn 通信之后的 applicationId。Client#submitApplication() 方法主要作用如下

- launcher server 进行连接
- 对 sparkconf 初始化，并创建对 ResourceManager 的代理客户端 rmClient，以及开启 historyClient 和 timelineClient(具体怎么建立代理会在下一部分详述，这里我们只需要理解成客户端操作服务端的方法就像操作本地的方法一样)
- 调用 YarnClientImpl#createApplication() 与 yarn 上创建一个 YarnClientApplication 对象，其类图如图1.4所示，这里面包含了重要的 appid 等信息

图 1.4　YarnClientApplication 类图

- 核对集群上是否满足 ApplicationMaster 的资源要求
- 创建 ApplicationMaster 的 Container 启动上下文信息，这里面将上传三种文件到 hdfs 对应的目录，分别为 spark-hdp-assemably、我们编写的 spark 应用 jar 包以及 spark 配置信息，之后将为 amContainer 封装一个启动 AM[①]的命令，类似于 java -server 这样的
- 创建提交 AM 的上下文信息，包括 spark 应用名称、AM 内存以及扩展值、AM 虚拟核数等信息
- 将 AM 上下文信息提交到 ResourceManager

run 里面第二个调用就是 monitorApplication(ApplicationId)。此函数通过 Client#submitApplication() 方法返回的 appid 实现对提交 spark 应用的监控，在实现上通过一个死循环，让客户端线程每隔 spark.yarn.report.interval 中定义的时间进行应用执行状态的获取，然后显示在客户端控制台上。应用执行状态 (ACCEPTED 之后) 即为以下四种的一种，FINISHED, FAILED,KILLED 或者 RUNNING。程序最终执行状态定义为以下四种 UNDEFINED, SUCCEEDED, FAILED 或 KILLED。

① AM 为 ApplicationMaster 的简写，以后本文中的 AM 不做特殊说明，均指代 ApplicationMaster

1.6　ApplicationMaster 的启动

前面通过 yarn 的代理客户端提交文件和提交程序信息之后，由 yarn 来进行分配 container[①]给 am，并将 am 的上下文信息发送给包含 amContainer 的 nodeManager，由其来启动 am。am 的 main 函数如程序1.7所示

```
1 def main(args: Array[String]): Unit = {
2   SignalLogger.register(log)
3   val amArgs = new ApplicationMasterArguments(args)
4   SparkHadoopUtil.get.runAsSparkUser { () =>
5       master = new ApplicationMaster(amArgs, new YarnRMClient(amArgs))
6       System.exit(master.run())
7   }
8 }
```

程序 1.7　ApplicationMaster 中 main 函数

在 spark 源码中 am 由单例对象和伴生类组成，main 函数的处理过程如下

- 对传入 am 的参数进行解析封装成 ApplicationMasterArguments，其类图如图1.5所示

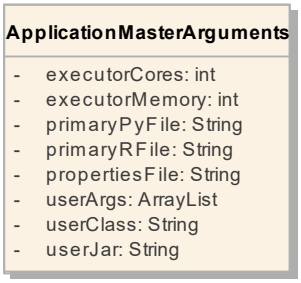


图 1.5　ApplicationMasterArguments 类图

- 实例化 ApplicationMaster，这里实例的是 ApplicationMaster 的伴生类，因为单例对象是不能实例化的，且我们要注意实例化 ApplicationMasterArguments ApplicationMaster 传入的第二个参数为 YarnRMClient，这个负责在 yarn 上注册或者取消 Application
- 执行 ApplicationMaster 的 run 方法，此方法首先获取 yarn 的配置信息，然后对 Application 进行注册，接着判断是否为 cluster 模式，是的话就是启动 driver，否则启动 executor

① container 为资源的抽象，包含 CPU、内存、网络资源和硬盘，Yarn 中负责分配 CPU 和内存

1.7　Driver 的启动

Yarn-Cluster 模式下 Driver 的启动过程如图1.6所示

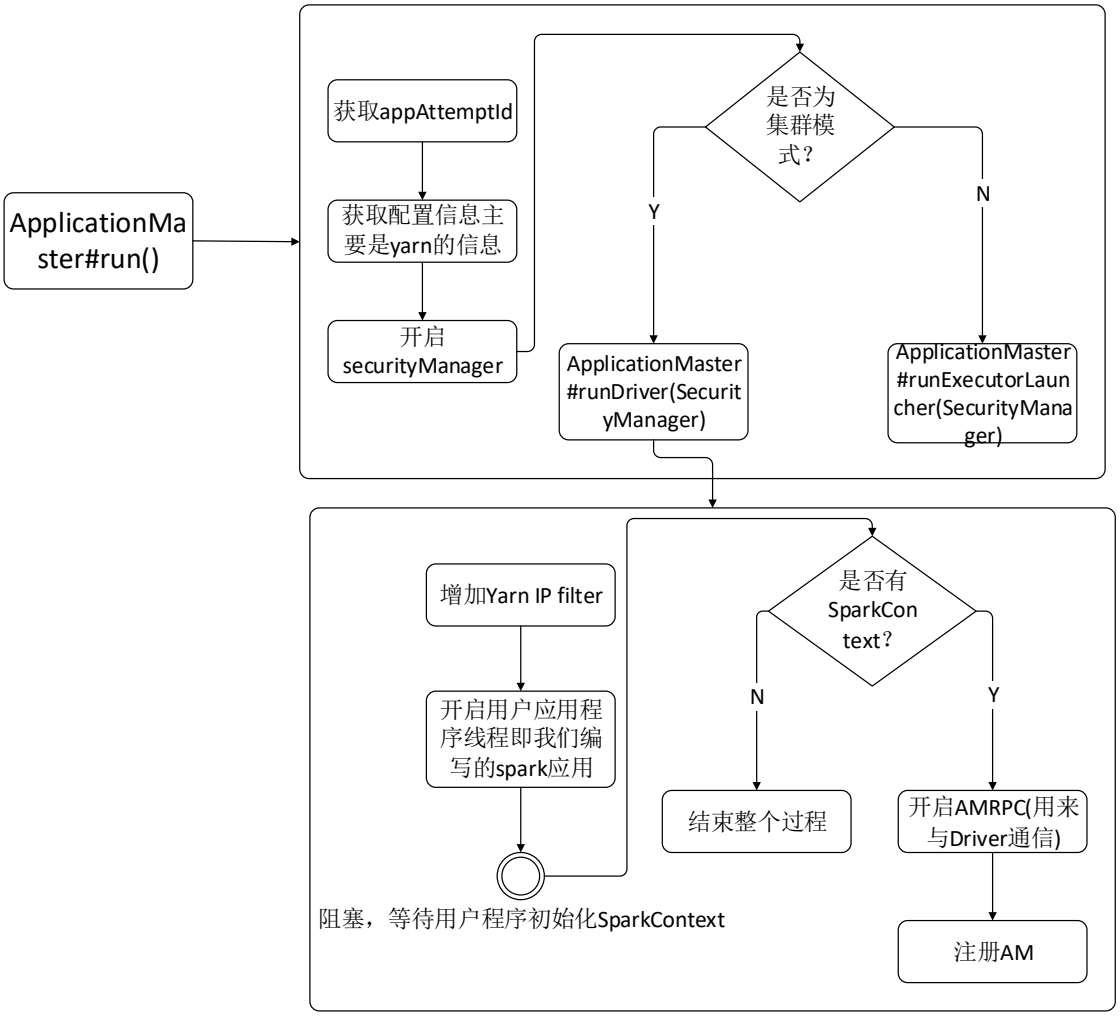


图 1.6　Driver 启动过程

启动 Dirver 的程序如程序1.8所示

```
1 private def runDriver(securityMgr: SecurityManager): Unit = {
2     addAmIpFilter()
3     userClassThread = startUserApplication()
4     val sc = waitForSparkContextInitialized()
5     if (sc == null) {
6         finish(FinalApplicationStatus.FAILED,
7         ApplicationMaster.EXIT_SC_NOT_INITED,
8         "Timed out waiting for SparkContext.")
```

```
9       } else {
10          rpcEnv = sc.env.rpcEnv
11          val driverRef = runAMEndpoint(
12          sc.getConf.get("spark.driver.host"),
13      sc.getConf.get("spark.driver.port"),
14      isClusterMode = true)
15      registerAM(rpcEnv, driverRef, sc.ui.map(_.appUIAddress).getOrElse("")
            ↪ , securityMgr)
16      userClassThread.join()
17    }
18  }
```

程序 1.8　启动 Driver

从源码中可以看到启动 Driver 的第一步为启动用户程序[①]，这里面对启动用户程序做了配置，具体如下

- 获得用户程序所需要的 ClassPath
- 获取类加载器信息
- 获得用户 jar 包路径
- 获得用户程序主类参数列表
- 开启一个新的线程，通过反射机制，以用户程序主类主函数为入口，加载用户程序

在用户程序主类的运行过程中，此时主线程[②]会阻塞，等待用户程序 SparkContext 初始化完成。

① 这里的用户程序即为我们所编写的 WorldCount 程序
② 指 AM 进程中运行 Driver 的线程

第 2 章　SparkContext 的初始化

Driver 线程启动子线程来加载用户程序主类，主类中第一步是对 SparkContext 的初始化，SparkContext 可以算得上是所有用户程序的启动引擎。而 SparkContext 初始化的配置参数由 SparkConf 负责。

2.1　执行环境 SparkConf

SparkConf 的构造很简单，主要包含的是提交用户程序的参数信息，包括 setMaster、setAppName 等。Spark 的配置属性都是以“Spark.”开头的字符串。下面介绍 SparkConf 中重要的方法

1 registerKryoClasses

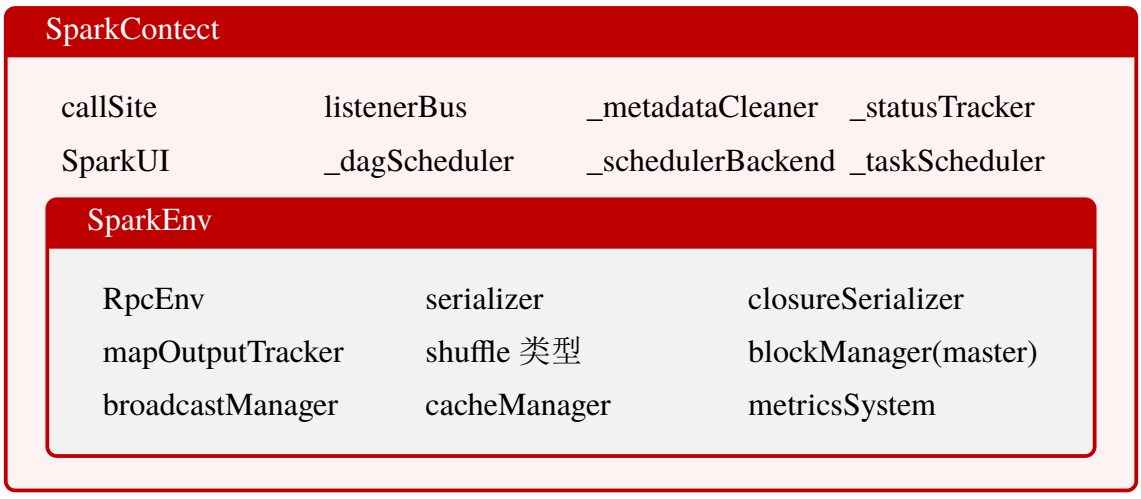
注册时使用 Kryo 序列化的类，指定该类的序列化器为 Kryo，关于 Kryo 的使用是 spark 性能优化的一个重要的组成部分。其中一个简单的应用是图计算。

2 registerAvroSchemas

序列化前先对 Schema 进行注册，影响的是压缩效率。

2.2　SparkContext 综合概述

配置完的 SparkConf 传入 SparkContext，将 SparkContext 比作汽车发动机的引擎，SparkConf 就是我们的控制面板，SparkContext 中的 SparkEnv 创建一些 Spark 应用运行必备的组件，像 Rpc、序列化器、块管理器、缓存管理器、测量系统、中间数据追踪器及文件服务器等，可以理解为汽车的空调、电路、油门线等环境条件。SparkContext 和 SparkEnv 关系可以表示为



2.2.1　调用栈

CallSite 中存储了线程栈中最靠近栈顶的用户类以及最靠近栈底的 Scala 或者 Spark 核心类，这个变量其实主要是给开发者看的，对于具体程序的运行没有必然的影响。当然它的信息也会反应在 SparkUI 的界面上的。

2.2.2　监听主线 ListenerBus

Spark 消息系统的具体实现为后台会自动开启一个线程来监听事件，现在基本上所有的分布式系统都是通过消息驱动的方式来进行并发执行的。具体的线程执行如程序2.1所示，从源码中可以看出，它其实是一个守护进程，是通过一个死循环，对事件队列里所有的事件进行遍历，再将不同的事件发送给不同的监听器 Listener。

</>　程序 **2.1:** *listenerBus 程序代码片段*　</>

```
private val listenerThread = new Thread(name) {
  setDaemon(true)
  override def run(): Unit = Utils.tryOrStopSparkContext(sparkContext)
   ↪  {
    AsynchronousListenerBus.withinListenerThread.withValue(true) {
      while (true) {
        eventLock.acquire()
          self.synchronized {
            processingEvent = true
          }
        try {
          val event = eventQueue.poll
          if (event == null) {
            if (!stopped.get) {
              ......
            }
            return
          }
          postToAll(event)
        } finally {
```

```
            self.synchronized {
              processingEvent = false
            }
          }
        }
      }
    }
}
```

2.2.3　元数据清理器

SparkContext 中 _metadataCleaner 为元数据清理器，SparkContext 为了保持对所有持久化的 RDD 的跟踪，使用类型是 TimeStampped-WeakValueHashMap 的 persistentRdds 缓存。元数据清理器的功能就是清除过期的持久化的 RDD。

2.2.4　可视化监控服务 SparkUI

Spark UI 提供监控功能，以具有样式和布局的网页形式来提供丰富的监控数据，通过浏览器就能进行访问。SparkUI 的创建代码如程序2.2所示

</>　程序 **2.2:** *SparkUI 声明*　</>

```
_ui =
if (conf.getBoolean("spark.ui.enabled", true)) {
  Some(SparkUI.createLiveUI(this, _conf, listenerBus,
    ↪  _jobProgressListener,
  _env.securityManager, appName, startTime = startTime))
} else {
  // For tests, do not enable the UI
  None
}
_ui.foreach(_.bind())
```

这其中的 createLiveUI，实际是调用了 SparkUI#create 方法，create 方法如程序2.3所示

</>　程序 2.3: *SparkUI* 创建方法　</>

```
private def create(
  sc: Option[SparkContext],
  conf: SparkConf,
  listenerBus: SparkListenerBus,
  securityManager: SecurityManager,
  appName: String,
  basePath: String = "",
  jobProgressListener: Option[JobProgressListener] = None,
  startTime: Long): SparkUI = {
    val _jobProgressListener: JobProgressListener =
     ↪  jobProgressListener.getOrElse {
    val listener = new JobProgressListener(conf)
    listenerBus.addListener(listener) listener
  }
  val environmentListener = new EnvironmentListener
  val storageStatusListener = new StorageStatusListener
  val executorsListener = new ExecutorsListener(storageStatusListener)
  val storageListener = new StorageListener(storageStatusListener)
  val operationGraphListener = new RDDOperationGraphListener(conf)
  listenerBus.addListener(environmentListener)
  listenerBus.addListener(storageStatusListener)
  listenerBus.addListener(executorsListener)
  listenerBus.addListener(storageListener)
  listenerBus.addListener(operationGraphListener)
  new SparkUI(sc, conf, securityManager, environmentListener,
    ↪  storageStatusListener,
  executorsListener, _jobProgressListener, storageListener,
    ↪  operationGraphListener,
  appName, basePath, startTime)
}
```

可以看出这里除了 JobProgressListener 是外部传入的之外，又增加了 Spark-

Listener。最后创建 SparkUI，SparkUI 服务默认是可以被杀掉的，可以修改属性 spark.ui.killEnabled 为 false 可以保证不被杀死。类 SparkUI 构造方法通过调用 initialize 方法会组织前端各个 Tab 和 Page 的展示及布局。SparkUI 的页面布局和展示使用的是 JobsTab。它可以展示所有的 Job 进度、状态信息，这里我们以它为例，JobsTab 会复用 SparkUI 的 killEnabled、SparkContext、jobProgressListener，包括 AllJobsPage 和 JobPage 两个页面,JobsTab 类实现代码程序2.4所示

</>　程序 **2.4:** *SparkUI JobsTab*　</>

```
private[ui] class JobsTab(parent: SparkUI) extends SparkUITab(parent,
  ↪  "jobs") {
  val sc = parent.sc
  val killEnabled = parent.killEnabled
  val jobProgresslistener = parent.jobProgressListener
  val executorListener = parent.executorsListener
  val operationGraphListener = parent.operationGraphListener
  def isFairScheduler: Boolean =
    jobProgresslistener.schedulingMode.exists(_ ==
      ↪  SchedulingMode.FAIR)
      attachPage(new AllJobsPage(this))
      attachPage(new JobPage(this))
  }
```

第一个就是 AllJobsPage，它由 render 方法进行渲染，利用 jobProgressListener 中统计监控数据生成激活、完成、失败等状态 Job 摘要信息，并调用 jobsTable 方法生成 html 各元素，最终使用 UIUtils 的 headerSparkPage 封装好 css、js、header 及页面布局等。渲染方法部分代码如程序2.5所示

</>　程序 **2.5:** 渲染方法实现　</>

```
def render(request: HttpServletRequest): Seq[Node] = {
  progressListener.synchronized {
    ......
    val stageHeader = s"Details for Stage $stageId (Attempt
      ↪  $stageAttemptId)"
    if (stageDataOption.isEmpty) {
```

```
6       val content =
7       <div id="no-info">
8       <p>No information to display for Stage {stageId} (Attempt
          ↪  {stageAttemptId})</p>
9       </div>
10      return UIUtils.headerSparkPage(stageHeader, content, parent)
11    }
12    if (stageDataOption.get.taskData.isEmpty) {
13      val content =
14      <div>
15      <h4>Summary Metrics</h4> No tasks have started yet
16      <h4>Tasks</h4> No tasks have started yet
17      </div>
18      return UIUtils.headerSparkPage(stageHeader, content, parent)
19  }
20  val stageData = stageDataOption.get
21  val tasks =
     ↪  stageData.taskData.values.toSeq.sortBy(_.taskInfo.launchTime)
22  val numCompleted = tasks.count(_.taskInfo.finished)
23  val allAccumulables = progressListener.stageIdToData((stageId,
     ↪  stageAttemptId)).accumulables
24  val externalAccumulables = allAccumulables.values.filter { acc =>
     ↪  !acc.internal }
25  val hasAccumulators = externalAccumulables.size > 0
26  val summary =
27  <div>
28  <ul class="unstyled">
29  <li>
30  <strong>Total Time Across All Tasks: </strong>
31  {UIUtils.formatDuration(stageData.executorRunTime)}
32  </li>
33  <li>
34  <strong>Locality Level Summary: </strong>
```

```
35  {getLocalitySummaryString(stageData)}
36  </li>
37  ......
```

SparkUI 创建好后，需要调用父类的 WebUI 的 bind 方法，绑定服务和端口，绑定实现代码如程序2.6所示

</>　程序 **2.6:** *SparkUI* 绑定实现　</>

```
1 /** Bind to the HTTP server behind this web interface. */
2 def bind() {
3   assert(!serverInfo.isDefined, "Attempted to bind %s more than
      ↪  once!".format(className))
4   try {
5     var host =
        ↪  Option(conf.getenv("SPARK_LOCAL_IP")).getOrElse("0.0.0.0")
6     serverInfo = Some(startJettyServer(host, port, handlers, conf,
        ↪  name))
7     logInfo("Bound %s to %s, and started at
        ↪  http://%s:%d".format(className, host,
8       publicHostName, boundPort))
9     } catch {
10      case e: Exception =>
11        logError("Failed to bind %s".format(className), e)
12        System.exit(1)
13    }
14 }
```

2.2.5　任务调度器

任务调度器组件由 _schedulerBackend 和 _taskScheduler 组成，在 Spark 框架中占据重要的地位，它们决定着 Spark 程序的整体运行流程，对应不同的资源管理方法和部署模式会有不同的实现。本文以 Yarn-Cluster 模式进行叙述。

SparkContext 调用 createTaskScheduler 方法，该方法返回值为一个二元组，其元素由 Scheduler 和 SchedulerBackend 组成，此方法执行时通过模式匹配的方式进

行，Yarn-Cluster 模式下的匹配代码如程序2.7所示

</>　程序 **2.7:** 匹配 *Yarn-Cluster*　</>

```
case "yarn-standalone" | "yarn-cluster" =>
  if (master == "yarn-standalone") {
    logWarning(
    "\"yarn-standalone\" is deprecated as of Spark 1.0. Use
      ↪  \"yarn-cluster\" instead.")
  }
  val scheduler = try {
    val clazz =
      ↪  Utils.classForName("org.apache.spark.scheduler.cluster.
    YarnClusterScheduler")
    val cons = clazz.getConstructor(classOf[SparkContext])
    cons.newInstance(sc).asInstanceOf[TaskSchedulerImpl]
  } catch {
  case e: Exception => {
    throw new SparkException("YARN mode not available ?", e)
  }
}
val backend = try {
  val clazz =Utils.classForName("org.apache.spark.scheduler.cluster.
  YarnClusterSchedulerBackend")
  val cons = clazz.getConstructor(classOf[TaskSchedulerImpl],
    ↪  classOf[SparkContext])
  cons.newInstance(scheduler,
    ↪  sc).asInstanceOf[CoarseGrainedSchedulerBackend]
} catch {
  ......
  }
}
scheduler.initialize(backend)
(backend, scheduler)
```

Yarn-Cluster 模式下两个组件分别得到下列值

Yarn-Cluster 模式

_schedulerBackend=YarnClusterSchedulerBackend(extends YarnSchedulerBackend)

_taskScheduler=YarnClusterScheduler(extends YarnScheduler)

1 YarnClusterScheduler 调度器

其类图如图2.1所示

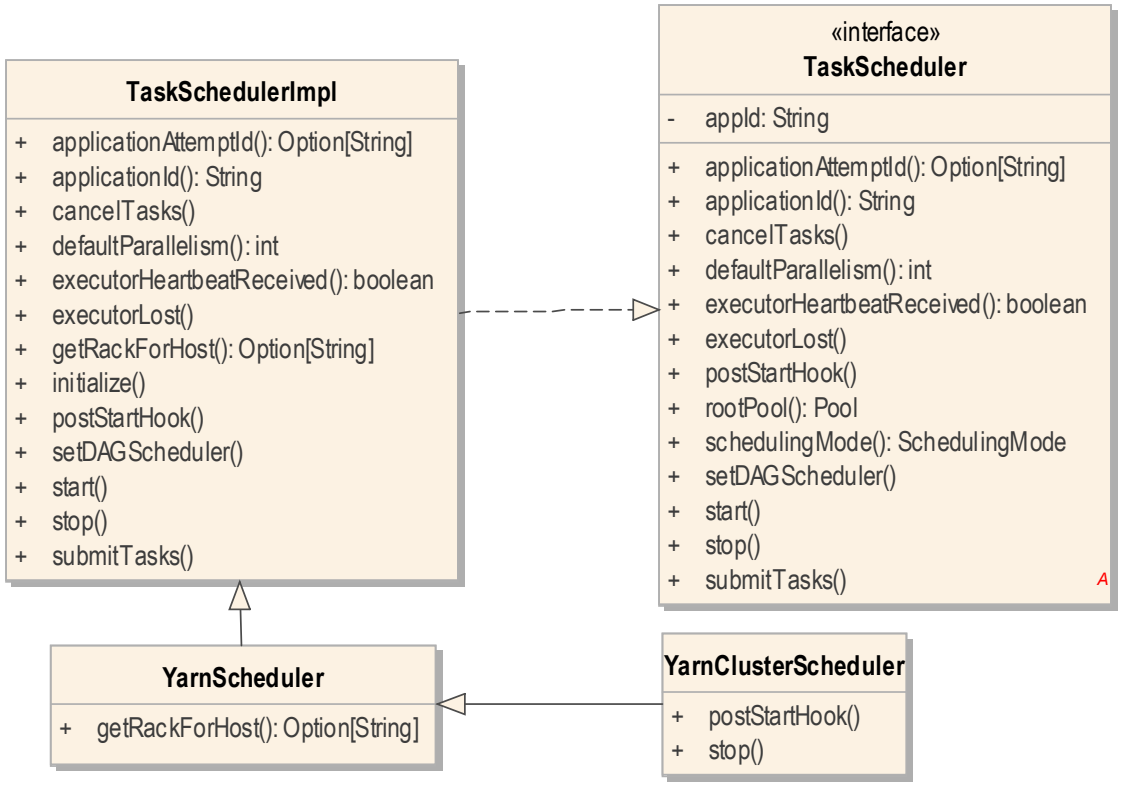


图 2.1　YarnScheduler 类图

程序2.7中倒数第二行代码，Scheduler 调用 initialize 方法对 backend 进行绑定，我们可以从类图2.1看到实际上是调用了 TaskSchedulerImpl#initialize() 方法，此方法代码很简短，指定 JOb 的调度模式，如程序2.8所示

</>　程序 2.8: 调度器初始化实现　</>

```
def initialize(backend: SchedulerBackend) {
  this.backend = backend
  rootPool = new Pool("", schedulingMode, 0, 0)
```

```
  schedulableBuilder = {
    schedulingMode match {
      case SchedulingMode.FIFO =>
      new FIFOSchedulableBuilder(rootPool)
      case SchedulingMode.FAIR =>
        new FairSchedulableBuilder(rootPool, conf)
      }
    }
    schedulableBuilder.buildPools()
}
```

此方法最重要的作用就是将 Scheduler 和 SchedulerBackend 进行绑定，并制定 Job 的调度模式，当然默认情况下为 FIFO。

接着执行 _taskScheduler.start()，由类图2.1可知实际上调用的是 TaskSchedulerImpl#start() 方法，此方法代码如程序2.9所示，此方法首先调用 YarnClusterSchedulerBackend#start()，之后判断是否需要检查慢服务。

</>　程序 2.9: 开启任务调度器　</>

```
override def start() {
  //Yarn-cluster 模式下为 YarnClusterSchedulerBackend
  backend.start()
    if (!isLocal && conf.getBoolean("spark.speculation", false))
      ↪  {
      speculationScheduler.scheduleAtFixedRate(new Runnable {
        override def run(): Unit =
          ↪  Utils.tryOrStopSparkContext(sc) {
          checkSpeculatableTasks()
        }
      }, SPECULATION_INTERVAL_MS, SPECULATION_INTERVAL_MS,
        ↪  TimeUnit.MILLISECONDS)
    }
}
```

2 YarnClusterSchedulerBackend 相关

与其相关的类及接口如图2.2所示

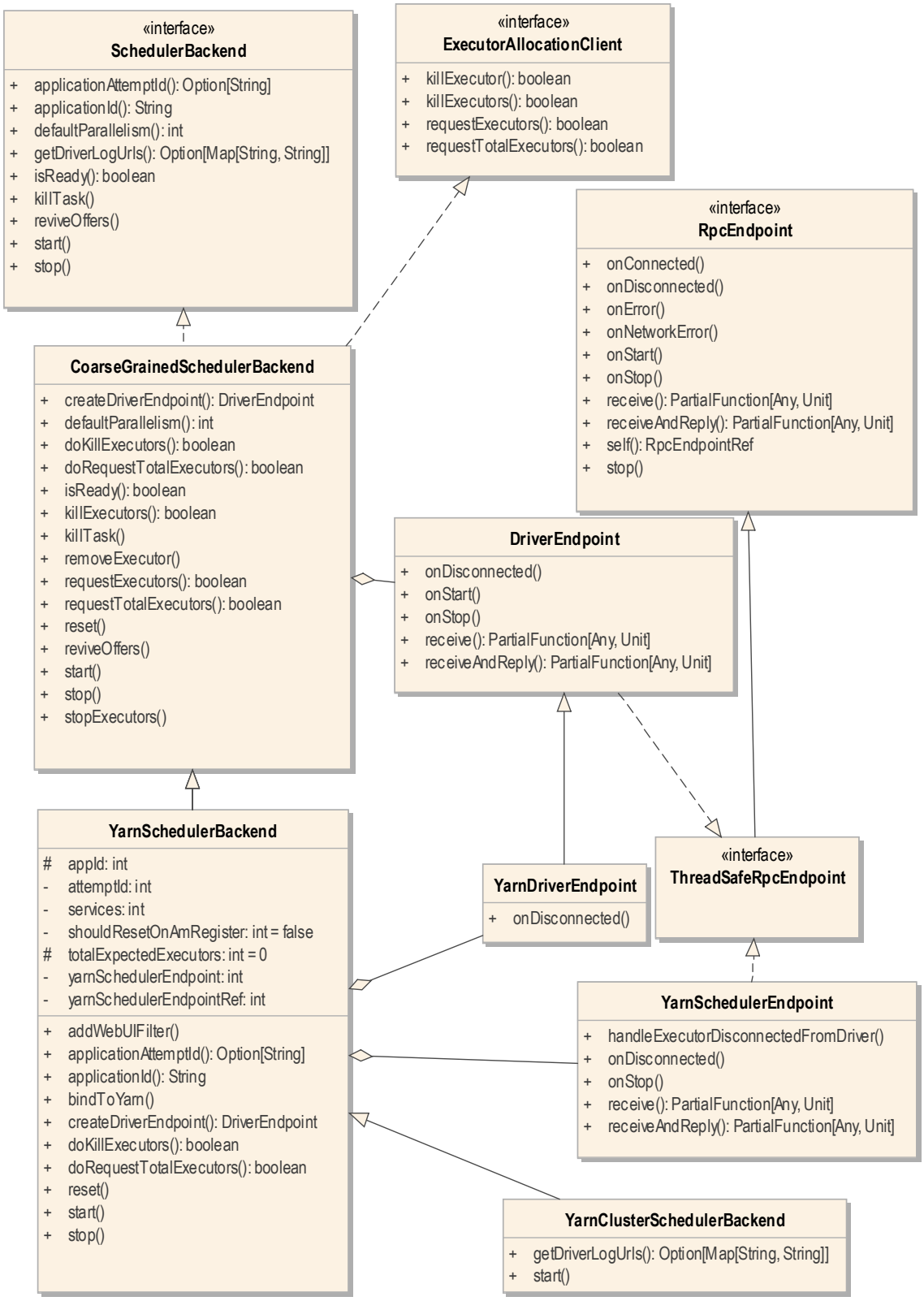


图 2.2　YarnSchedulerBackend 类图

接着从 TaskSchedulerImpl#start() 方法，该方法中第一行即执行 YarnClusterSchedulerBackend#start()，作用为，将 appid 和 attemptId 绑定到 Yarn 上；调用父类的 YarnSchedulerBackend#start() 方法。

YarnSchedulerBackend#start() 方法：验证 appid 是否已经定义；Yarn-Cluster 模式下，绑定 SchedulerExtensionServiceBinding；调用父类 CoarseGrainedSchedulerBackend#start() 方法；

CoarseGrainedSchedulerBackend#start() 方法如程序2.10所示

</>　程序 **2.10:** 开启 *CoarseGrainedScheduler*　</>

```
override def start() {
  val properties = new ArrayBuffer[(String, String)]
  for ((key, value) <- scheduler.sc.conf.getAll) {
    if (key.startsWith("spark.")) {
      properties += ((key, value))
    }
  }
  // TODO (prashant) send conf instead of properties
  driverEndpoint = rpcEnv.setupEndpoint(ENDPOINT_NAME,
    ↪ createDriverEndpoint(properties))
}
```

程序2.10中 RPC 在 SparkContext#createSparkEnv 函数调用里，具体函数实现流程如图2.3所示。

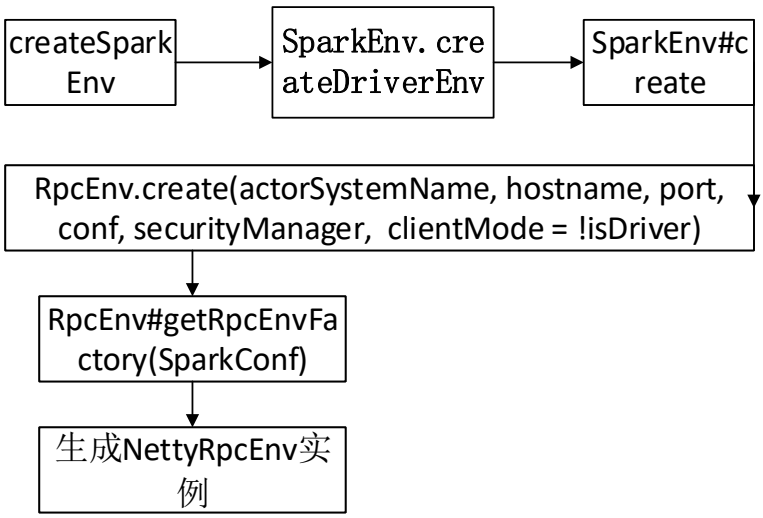


图 2.3　RPC 建立过程

流程图2.3可以看出 SparkRPC 是通过工厂模式反射生成 NettyRpcEnv 实例。那么 CoarseGrainedSchedulerBackend#start() 最后一行就执行的是 NettyRpcEnv#setupEndpoint()，接着执行 Dispatcher#registerRpcEndpoint()，最后就返回了 NettyRpcEndpointRef。

2.2.6　DAG 调度器

DAGScheduler 用于构建 Job 阶段，将用户应用的 DAG 划分为不同的 Stage，其中 Stage 由可以并发控制的一组 Task 构成，这些 Task 逻辑上完全相同，只是运行在不同的数据分区上。具体的内部实现原理会在 Job 构建章节中进行讲述。DAGScheduler 的数据结构主要维护 jobId 和 stageId 的关系、Stage、ActiveJob，以及缓存的 RDD 的 partitions 的位置信息。DAGScheduler 的数据结构如程序2.11所示

</>　程序 **2.11:** *DAG 调度器数据结构*　</>

```
1 private[scheduler] val nextJobId = new AtomicInteger(0)
2 private[scheduler] def numTotalJobs: Int = nextJobId.get()
3 private val nextStageId = new AtomicInteger(0)
4 private[scheduler] val jobIdToStageIds = new HashMap[Int,
  ↪  HashSet[Int]]
5 private[scheduler] val stageIdToStage = new HashMap[Int, Stage]
6 private[scheduler] val shuffleToMapStage = new HashMap[Int,
  ↪  ShuffleMapStage]
7 private[scheduler] val jobIdToActiveJob = new HashMap[Int, ActiveJob]
8 // Stages we need to run whose parents aren't done
9 private[scheduler] val waitingStages = new HashSet[Stage]
10 // Stages we are running right now
11 private[scheduler] val runningStages = new HashSet[Stage]
12 // Stages that must be resubmitted due to fetch failures
13 private[scheduler] val failedStages = new HashSet[Stage]
14 private[scheduler] val activeJobs = new HashSet[ActiveJob]
```

在初始化 DAGScheduler 的时候，会调用 eventProcessLoop#start()，这个用于创建子线程一直运行的事件监听线程。其调用过程如下图2.4所示

图 2.4　事件监听开启流程

事件线程运行代码如程序2.12所示

程序 **2.12:** 事件监听线程

```
private val eventThread = new Thread(name) {
  setDaemon(true)
  override def run(): Unit = {
    try {
      while (!stopped.get) {
        val event = eventQueue.take()
          try {
            onReceive(event)
          } catch {
            case NonFatal(e) => {
              ......
                }
              }
            }
          } catch {
            ......
        }
    }
}
```

从程序2.12中可以看出，事件监听器是个后台守护进程，其中运行的线程是一个由 while 循环组成的结构，其不断的从事件队列中取出事件，然后调用 onReceive(event) 方法对不同的事件进行模式匹配。这里的 onReceive(event) 是一个由子类实现的方法，如程序2.13所示

</>　程序 **2.13:** *DAGSchedulerEventProcessLoop.onReveive*　</>

```
override def onReceive(event: DAGSchedulerEvent): Unit = {
  val timerContext = timer.time()
  try {
    doOnReceive(event)
  } finally {
    timerContext.stop()
  }
}
```

程序中运行的是 doOnReceive(DAGSchedulerEvent) 方法，而这个方法中就包含了对不同事件的模式匹配以及对应着不同的处理方式。

第 3 章　Spark RDD 创建及转换

经过第2章 SparkContext 的初始化，Driver 端会立即进行 RDD 的生成和转换。本章将按照 WordCount 实例对 RDD 生成和转换进行详解。

3.1　Spark RDD 综述

RDD 即为弹性数据集，它可以由存储在物理中的数据来生成，也可以通过其他 RDD 进行转换，RDD 是 Spark 平台上的最基础的数据结构，分布式下可并行处理，并且对数据分区也可以根据业务不同进行改变，同时在系统内存资源足够的情况下，允许用户将计算中间结果缓存在内存中，后续的计算可以直接使用这个结果，这会极大的提高计算效能。

每个 RDD 内部，有下面 5 个主要的属性：

1 数据分区列表

为数据的基本组成单元，对于 RDD 来说，一个分区上将会对应一个计算任务，分区的个数会决定 Job 的并发度。在使用时可以由用户显示的指定分区数，没有指定的情况下，集群会采用默认的分区数，即集群工作节点上一个 cpu core 对应一个分区。分区的管理为 BlockManager。

RDD 分区的个数如下公式所示

$$partitions = \max\{blockSize, defaultMinPartition\} \tag{3-1}$$

$$defaultMinPartition = \begin{cases} textFile & \text{if 定义并行度} \\ \min\{\max\{totalCores, 2\}, 2\} \end{cases} \tag{3-2}$$

2 每个分区上的计算函数

Spark 中每个 RDD 都会实现一个 computer 函数，computer 函数会对迭代器进行连接，但不会保存每次连接运算的中间结果。

3 对其他 RDD 的依赖列表

RDD 的每次转换都会生成一个新的 RDD，新生成 RDD（子 RDD）会将旧 RDD（父 RDD）加入依赖，并将依赖加入列表中，这样就会形成一条线连接

起各个 RDD，这条线也称作生命线（lineage）。

4（可选）一个分区器

针对 key-value 的 RDD，分区器按照 RDD 的键值对进行处理，父 RDD 中的键值对在子 RDD 中分到哪个分区，分区器有两种，分变为 HashPartitioner 和 RangePartitioner。分区器不但决定了 RDD 本身的分区数量，也决定了子 RDD 中输出的分区数量。

5（可选）计算每个分区优先位置列表

对于数据源存储在 HDFS 中的，此列表存储的就是每个分区所在的块的位置。

3.2　Spark RDD 创建

创建 RDD 的方式有两种，第一种从存储在物理设备上的数据集创建，常见的有本地文件系统、HDFS 和 HBase，第二种通过已经存储在 RDD 进行转换。WordCount 实例中数据存放在 HDFS 中，通过 textFile 读取 HDFS 文件内容并创建 HadoopRDD。

3.3　Spark RDD 转换

Spark RDD 中的所有转换都是惰性的，在触发动作之前都只做单纯的连接。WordCount 实例中，在 RDD 创建后通过 flatmap 生成一个新的数据集，并传入 map 生成新的数据集，接着传入 reduceByKey，再接着传入 filter 生成新的数据集，最后返回给 Driver 的是 saveAsText 处理后的数据集，而不是整个数据集。

3.4　DAG 的生成

Spark 根据用户提交的计算逻辑中 RDD 的转换和动作来生成 RDD 之间的依赖关系，同时这个计算链条也就成了逻辑上的 DAG。

3.4.1　RDD 中依赖

RDD 中每个 Dependency 子类内部都会存储一个 RDD 对象，对应一个父 RDD，如果一次转换操作有多个父 RDD，就会对应产生多个 Dependency 对象，使用 List 存储，所有的 Dependency 对象存储在子 RDD 内部，通过遍历 RDD 内部的 Dependency 对象，就能获取该 RDD 所有依赖的父 RDD。RDD 中主构造函数声明了 Dependency 的类型，如程序3.1所示，可见 RDD 的 deps 为 Seq 类型，且其中的元素为 Dependency。

程序 3.1: *RDD 主构造函数*

```
abstract class RDD[T: ClassTag](
  @transient private var _sc: SparkContext,
  @transient private var deps: Seq[Dependency[_]]
  ) extends Serializable with Logging {
    ......
}
```

子 RDD 和父 RDD 的依赖关系有两种，即宽依赖和窄依赖。两种依赖最简单的理解如图3.1所示

1 宽依赖是指子 RDD 中的一个分区依赖于父 RDD 的多个分区

2 窄依赖是指子 RDD 中的一个分区依赖于父 RDD 的一个分区

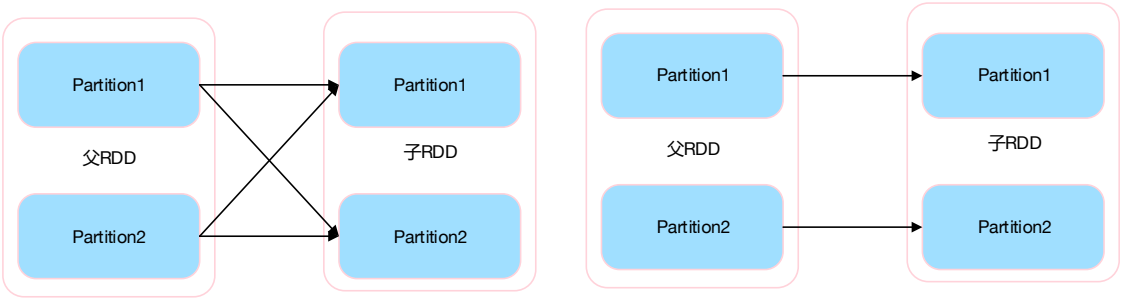


图 3.1　宽依赖（左）窄依赖（右）

RDD 不同的转换操作对应分区的映射规则不一样，但对于窄依赖而言计算任务作用于不同的数据分区，它们之间相互没有影响，所以这些计算任务也就可以并发的执行了。宽依赖中子 RDD 的单个分区要依赖父 RDD 的多个分区，因此这两个 RDD 不能通过一个计算任务来完成，这里需要进行 shuffle。

RDD 中依赖的层次图如图3.2所示

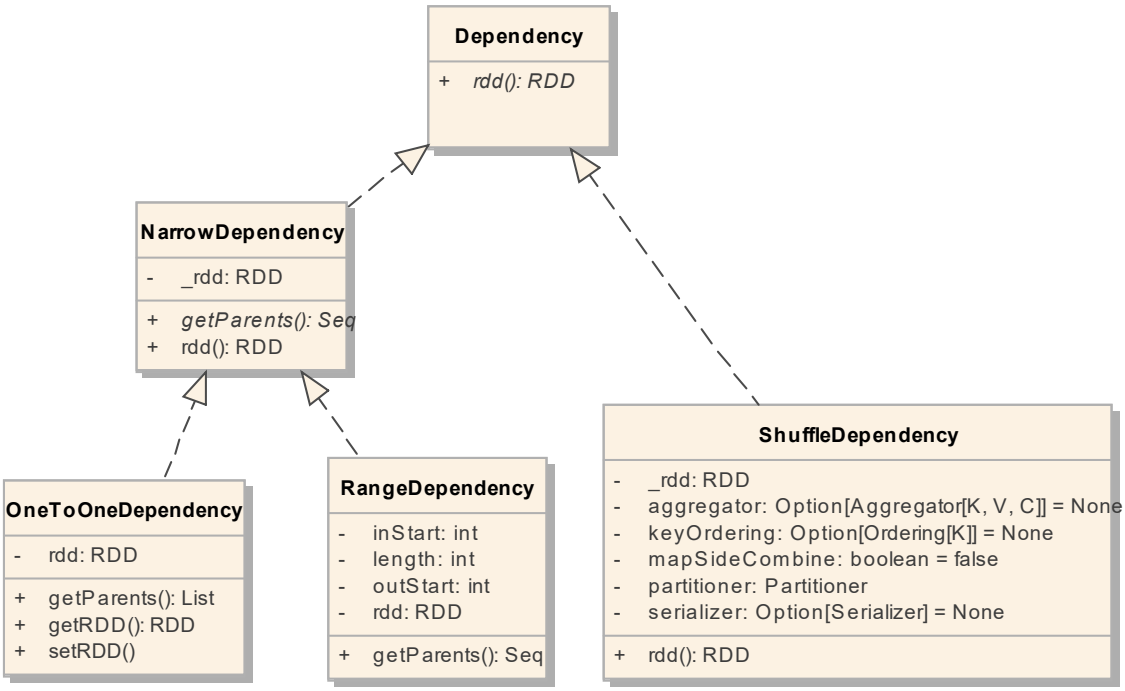


图 3.2　Spark RDD 依赖类层次图

所有的依赖都要实现抽象类 Dependency，其代码如程序3.2所示

程序 3.2: *抽象类 Dependency*

```
abstract class Dependency[T] extends Serializable {
  def rdd: RDD[T]
}
```

这其中的 rdd 即为参数此 RDD 的父 RDD，此为抽象方法，在子类中都必须有其实现。

窄依赖的实现如程序3.3所示

程序 3.3: *NarrowDependency*

```
abstract class NarrowDependency[T](_rdd: RDD[T]) extends Dependency[T]
  ↪ {
  def getParents(partitionId: Int): Seq[Int]
  override def rdd: RDD[T] = _rdd
}
```

这其中实现了对父 RDD 中 rdd 方法的实现，同时定义了一个抽象方法 getParents，获取该依赖对应的父 RDD 的分区。

对于窄依赖的实现有两种，一种是 OneToOneDependency，另一种是 RangeDependency，这两种依赖都实现了 getParents 方法，只不过 OneToOneDependency 中父 RDD 和子 RDD 中的分区 ID 是相同的，RangeDependency 仅仅被 org.apache.spark.UnionRDD 所使用，针对的是子 RDD 有多个父 RDD 的情形，但一对父子 RDD 中，分区任然是一对一，这种情形下依赖如图3.3所示

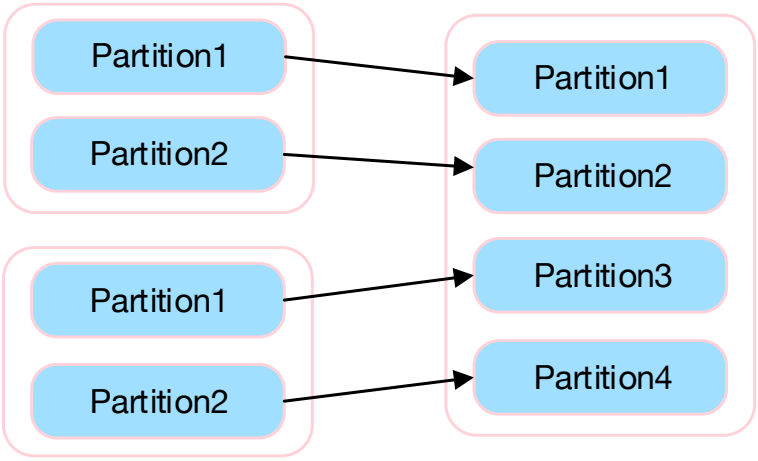


图 3.3　RangeDependency 示例

宽依赖只有一种实现：ShuffleDependency，其代码如程序3.4所示

</>　程序 **3.4:** *ShuffleDependcy*　</>

```
class ShuffleDependency[K: ClassTag, V: ClassTag, C: ClassTag](
  @transient private val _rdd: RDD[_ <: Product2[K, V]],
  val partitioner: Partitioner,
  val serializer: Option[Serializer] = None,
  val keyOrdering: Option[Ordering[K]] = None,
  val aggregator: Option[Aggregator[K, V, C]] = None,
  val mapSideCombine: Boolean = false)
    extends Dependency[Product2[K, V]] {
      override def rdd: RDD[Product2[K, V]] =
        ↪  _rdd.asInstanceOf[RDD[Product2[K, V]]]
      private[spark] val keyClassName: String =
        ↪  reflect.classTag[K].runtimeClass.getName
      private[spark] val valueClassName: String =
        ↪  reflect.classTag[V].runtimeClass.getName
```

```
12      private[spark] val combinerClassName: Option[String] =
13      Option(reflect.classTag[C]).map(_.runtimeClass.getName)
14      val shuffleId: Int = _rdd.context.newShuffleId()
15      val shuffleHandle: ShuffleHandle =
16       _rdd.context.env.shuffleManager.registerShuffle(
17        shuffleId,_rdd.partitions.size,this)
18      _rdd.sparkContext.cleaner.foreach(
19        _.registerShuffleForCleanup(this))
20  }
```

这里处理了一些变量，为 map 端本地聚合做准备。shuffle 的具体过程会在以后章节进行分析。

3.4.2　DAG 构建

RDD 通过一系列转换，会形成自己的依赖列表，列表中的每个依赖都会包含一个父 RDD 的对象，ShuffleRDD 依赖中另外包含了分区器、序列化器、map 端聚合等参数。这些信息提供了子 RDD 由那些父 RDD 转换而来以及它所依赖的父 RDD 的分区信息。逻辑上，各 RDD 通过一条线连接，且由子 RDD 指向父 RDD，这样就形成了一条 Lineage，也可看做这些 RDD 形成了 DAG。

Spark 根据由 DAG 来划分 Stage，进而生成计算任务。在同一个分区上进行计算的任务可以放在一个线程中计算，而宽依赖中子 RDD 要从父 RDD 的各个分区中拉取数据，并且放到不同的分区中进行计算，所以得新开一个线程进行。由此可得出划分 Stage 的依据就是 ShuffleDependency。划分阶段在 RDD 触发 Action 之后，具体分析会在调度章节进行说明。

3.4.3　WordCount 的 RDD 转换和 DAG 生成

本实例中 WordCount 运行在三台主机构成的集群中，两台为 NodeManager，一台作为 ResourceManager，输入数据保存在 HDFS 上，且存在 DataNode 中的某一台上，RDD 转换的细节如图3.4所示

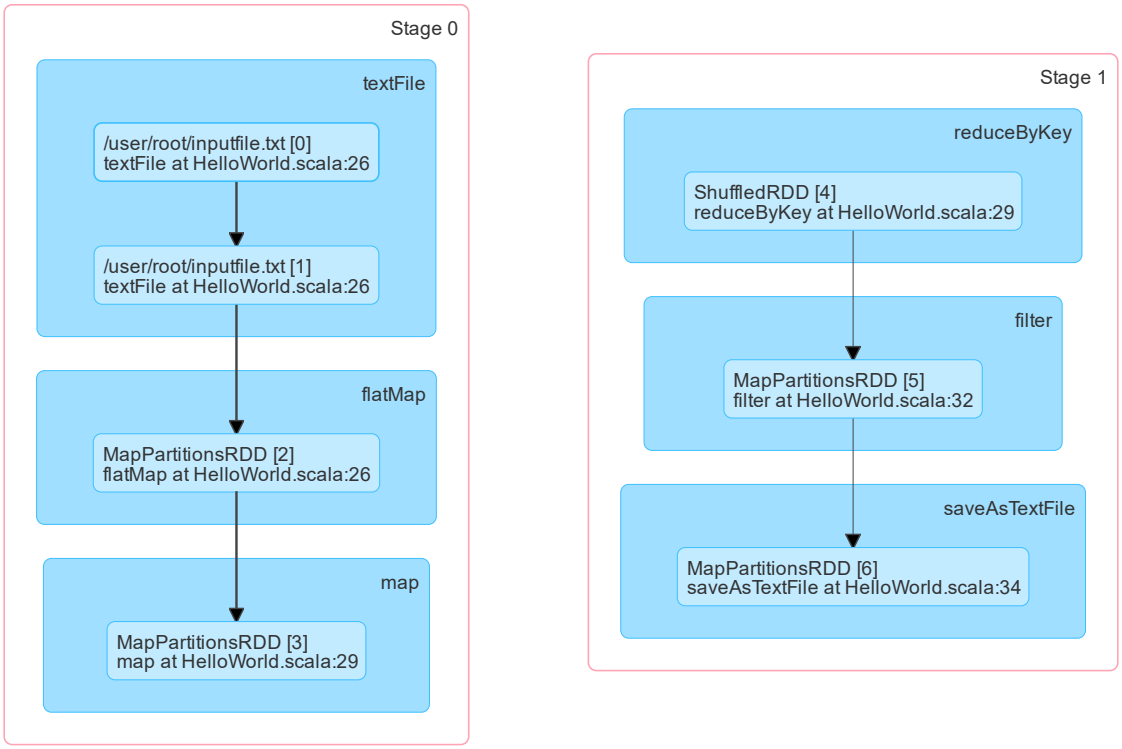


图 3.4　WordCount 的 RDD 转换

通过图3.4，可以清晰的看到数据被分成了两个分区，程序使用的默认分区数，即一个 cpu core 对应一个分区。用户对应的每一个操作会对应生成一个 RDD，但对应 shuffle 操作如图中的 reduceByKey 会隐式的读在 map 端和 reduce 端分别产生一个 MapPartitionRDD，中间生成一个 shuffldedRDD，不过这些都是 Spark 内部进行的，用户不用关心。

为了加深对图3.4中 RDD 的转换关系的理解，图3.5描述了在本 WordCount 实例中真实数据下的转换和计算过程。

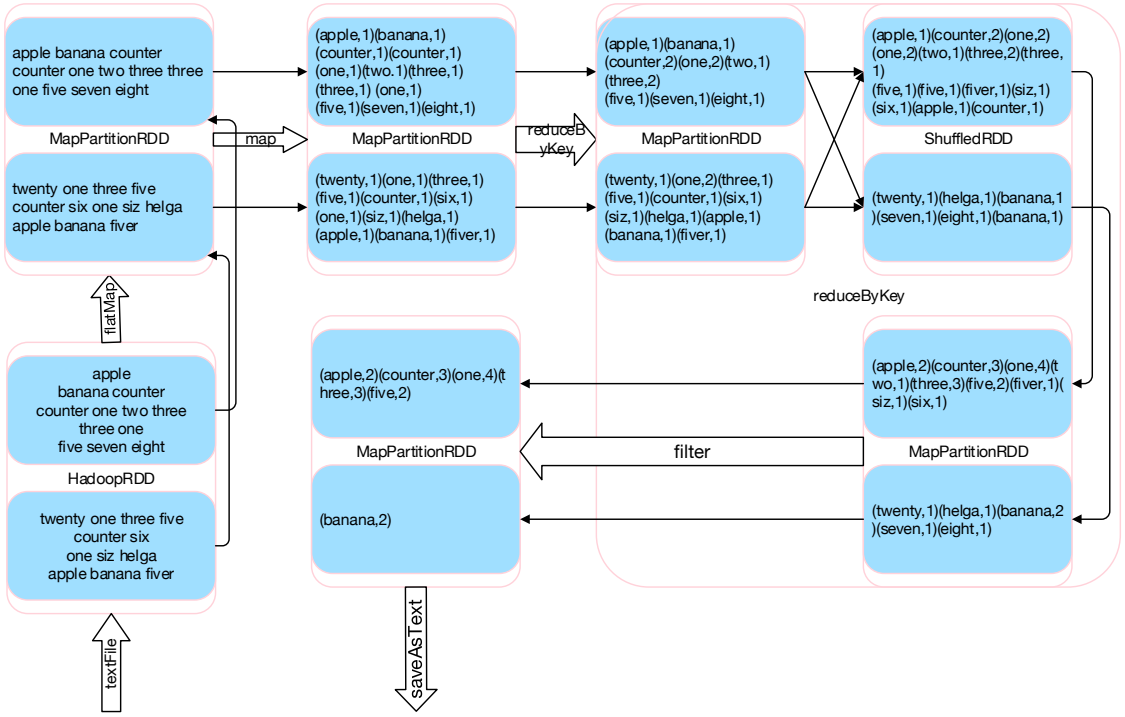


图 3.5　WordCount 的数据逻辑图

源数据存放在 HDFS 中，通过 texFile 读取后形成两个分区的 HadoopRDD，这里是因为测试集群中总共两个 cpu core，分区数设置为默认分区。这里也应该注意到在 reduceByKey 操作中，Spark 产生了三个 RDD，第一个为本地聚合后形成的 MapPartitionRDD，第二个为通过网络操作按照分区器所设置的记录和子 RDD 中分区对应关系，将父 RDD 中相关记录拉取到子 RDD 中，此部分需要网络，也需要跨分区，这部分为性能调优的重点。最后一个 RDD 为 reduce 端聚合。

第 4 章　Job 运行和调度器模块

RDD 上的操作分为两种类型转换和动作。第3章中 RDD 的转换会构成 DAG，最后 RDD 会触发 action，WordCount 例子中触发 action 的算子为 saveAsTextFile。此算子会将数据集中的元素以 textFile 的形式保存到传入的路径中，本地文件或者 HDFS。RDD 中的 saveAsTextFile 经过一系列 runJob 的调用，最后调用了 DagScheduler 中的 runJob。DagScheduler 在第2章中已被初始化。其代码如程序4.1所示。

</>　程序 **4.1:** *DagScheduler.runJob*　</>

```
def runJob[T, U](rdd: RDD[T],func: (TaskContext, Iterator[T]) =>
  ↪  U,partitions: Seq[Int],callSite: CallSite,resultHandler: (Int, U)
  ↪  => Unit,properties: Properties): Unit = {
 ......
 val waiter = submitJob(rdd, func, partitions, callSite,
   ↪  resultHandler, properties)
 waiter.awaitResult() match {
   case JobSucceeded =>
     (waiter.jobId, callSite.shortForm, (System.nanoTime - start) /
       ↪  1e9))
   case JobFailed(exception: Exception) =>
     (waiter.jobId, callSite.shortForm, (System.nanoTime - start) /
       ↪  1e9))
     val callerStackTrace = Thread.currentThread().getStackTrace.tail
     exception.setStackTrace(exception.getStackTrace ++
       ↪  callerStackTrace)
     throw exception
 }
}
```

DagScheduler.runJobz 主要调用了 submitJob 方法和 waiter.awaitResult()，这里可以看出任务的运行是异步的。submitJob 方法如程序4.2所示, 代码中只保留了重要部分。

</> 程序 **4.2:** *DAGScheduler.submitJob* </>

```
def submitJob[T, U](
rdd: RDD[T],
func: (TaskContext, Iterator[T]) => U,
partitions: Seq[Int],
callSite: CallSite,
resultHandler: (Int, U) => Unit,
properties: Properties): JobWaiter[U] = {
......
val jobId = nextJobId.getAndIncrement()
......
val waiter = new JobWaiter(this, jobId, partitions.size,
  ↪  resultHandler)
eventProcessLoop.post(JobSubmitted(
  jobId, rdd, func2, partitions.toArray, callSite, waiter,
  SerializationUtils.clone(properties)))
  waiter
}
```

由程序4.2可以看出 submitJob 会想生成一个 jobId，并且生成 jobwaiter 来监听 job 的运行状态，最后通过 DAGScheduler 的事件处理线程进行处理。dag-scheduler-event-loop 在 SparkContex 中初始化且作为守护线程存在。DAGSchedulerEventProcessLoop 收到 JobSubmitted 事件后会立即调用 DAGScheduler.handleJobSubmitted 方法进行处理，这方法中会进行 Stage 的划分以及提交 Stage。

4.1　DagScheduler 实现

4.1.1　Stage 的划分

在 Spark 中，一个 Job 会被分成多个 Stage，各个直接存在着依赖关系，其中最下游的 Stage 也成为 finalStage 或 resultStage。并行在不同分区进行相同的逻辑计算且互不影响的任务属于同一个 Stage，而 shuffle 过程要从父 RDD 的多个分区中拉取数据，必须要等待多个分区任务完成才能开始计算，故其不能并行化，属于不同的 Stage，DAG 中划分 Stage 就是以 Shuflle 为依据进行的。

handleJobSubmitted 通过调用 org.apache.spark.DAGScheduler#newResultStage 来创建 finalStage，

```
finalStage = newResultStage(finalRDD, func, partitions, jobId, callSite)
```

newResultStage 会根据传入的 finalRDD 来获取父 Stage 和 jobId，然后开始创建 ResultStage 即当前的 Stage，这里父 RDD 使用列表类型。代码如程序4.3所示

</> 程序 **4.3:** *DAGScheduler.newResultStage* </>

```
1 private def newResultStage(rdd: RDD[_],func: (TaskContext,
  ↪  Iterator[_]) => _,partitions: Array[Int],
2 jobId: Int,callSite: CallSite): ResultStage = {
3   val (parentStages: List[Stage], id: Int) = getParentStagesAndId(rdd,
    ↪  jobId)
4   val stage = new ResultStage(id, rdd, func, partitions, parentStages,
    ↪  jobId, callSite)
5   stageIdToStage(id) = stage
6   updateJobIdStageIdMaps(jobId, stage)
7   stage
8 }
```

程序4.3中 getParentStagesAndId 方法会进一步调用 getParentStages 方法，getParentStages 方法为划分 Stage 的核心。它的返回类型为 List[Stage]，每遇到 DAG 中 RDD 的 ShuffleDependency 就会参数一个新的 Stage，其代码如程序4.4所示。

</> 程序 **4.4:** *DAGScheduler.getParentStages* </>

```
1 private def getParentStages(rdd: RDD[_], firstJobId: Int): List[Stage]
  ↪  = {
2   val parents = new HashSet[Stage]
3   val visited = new HashSet[RDD[_]]
4   val waitingForVisit = new Stack[RDD[_]]
5   def visit(r: RDD[_]) {
6     if (!visited(r)) {
7       visited += r
8       for (dep <- r.dependencies) {
```

```
9           dep match {
10            case shufDep: ShuffleDependency[_, _, _] =>
11              parents += getShuffleMapStage(shufDep, firstJobId)
12            case _ =>waitingForVisit.push(dep.rdd)
13          }
14        }
15      }
16    }
17    waitingForVisit.push(rdd)
18    while (waitingForVisit.nonEmpty) {
19      visit(waitingForVisit.pop())
20    }
21    parents.toList
22  }
```

getParentStages 方法中定义了一个 visit 方法，此方法会对方法外的变量 parents、visited 及 waitingForVisit 产生影响，这里用到 Scala 函数的闭包特性。getParentStages 变量说明如下

getParentStages 变量说明

parents：存放各个 ShuffleDependency 所依赖的 Stage
visited：存放 DAG 中已经遍历过的 RDD
waitingForVisit：存放即将要遍历的 RDD，为栈结构

4.1.2　实例:WordCount 划分 Stage

以本 WordCount 样例展示如何划分 Stage。WordCount DAG 如图3.4所示，提前三个重要的信息组成新的 RDD 数据结构，定义如下

新定义 RDD 数据结构

数据结构：RDD(deps:List[Dependency],partitions:List(partitionId))
deps：代表当前 RDD 的依赖列表
partitions: 指当前 RDD 的分区列表
textFile：
HadoopRDD[1](Nil,List(1,2))
flatMap:
MapPartitionRDD[2](List(onoToOneDependency(HadoopRDD[1]),List(1,2))
map:
MapPartitionRDD[3](List(onoToOneDependency(MapPartitionRDD[2])),List(1,2))
reduceByKey:
ShuffledRDD[4](List(ShuffleDependency(MapPartitionRDD[3])),List(3,4))
filter:
MapPartitionRDD[5](List(onoToOneDependency(ShuffledRDD[4])),List(3,4))
saveTextFile:
MapPartitionRDD[6](List(onoToOneDependency(MapPartitionRDD[5])),List(3,4))

这里定义 RDD 中包含了两种数据元素，分别为当前 RDD 的依赖列表和当前 RDD 的分区列表。依赖列表中存放有当前 RDD 所依赖的父 RDD 的对象，分区列表中存放着分区的 Id，要获取该分区数据可以通过 BlockManager 来获取。

DAG 以 finalRDD 即 MapPartitionRDD[6] 为起始点，依次进入函数 getParentStages, 各变量参数的值变化情况如表4.1所示

表 4.1　getParentStages 变量值

visited:HashSet	parents:HashSet	waittingForVisit:Stack
6	Nil	6
6	Nil	5
6 5	Nil	4
6 5 4	ShuffleMapStage(3,shuffleDep(3))	Nil

这里函数生成了 ShuffleMapStage，其中存放了 RDD[3] 的信息, 另外 ShuffleMapStage 中 Task 计算的结果会传给 Driver 端的 mapOutputTracker，其他任 Task 可以通过查询它来获取这些结果，不过这些结果并不是真实的数据，而是保存真

实数据所在位置、大小等元数据信息，其他 Task 通过这些元数据信息获取其需要处理的数据。接下来回到程序4.3，进入 newResultStage 代码块。最后返回 ResultStage(RDD[6],List(ShuffleMapStage(3,shuffleDep(3))))。最后函数 newResultStage 将 jobId 和 ResultStage 进行绑定，返回 ResultStage，也就是 finalStage。

4.2　任务调度器实现

4.2.1　任务的生成

现在回到 handleJobSubmitted，上节生成了 finalStage，此方法后面部分会调用 submitStage 来提交这个 Stage，如果当前提交的 Stage 还有父 Stage 没有提交，那么就递归提交父 Stage，只有父 Stage 提交完了，才能提交当前 Stage，submitStage 如程序4.5所示

</>　程序 **4.5:** *DAGScheduler.submitStage*　</>

```
1 private def submitStage(stage: Stage) {
2   val jobId = activeJobForStage(stage)
3   if (jobId.isDefined) {
4     if (!waitingStages(stage) && !runningStages(stage) &&
      ↪  !failedStages(stage)) {
5       val missing = getMissingParentStages(stage).sortBy(_.id)
        ↪  //List(ShuffleMapStage[0])
6       if (missing.isEmpty) {
7         submitMissingTasks(stage, jobId.get)
8       } else {
9         for (parent <- missing) {
10          submitStage(parent)
11        }
12        waitingStages += stage//此时 waitingStages=set(ResultStage 1)
13      }
14    }
15  } else {
16    abortStage(stage, "No active job for stage " + stage.id, None)
17  }
18}
```

经过上节 DAGScheduler 对 DAG Stage 的划分，WordCount 实例逻辑上可以形成图3.4所示的 Stage。Stage1 作为 submitStage 的参数进入，函数先判断其是否有父 Stage，有父 Stage 就递归提交父 Stage，最后会由 submitMissingTasks 来完成最后的提交任务工作。submitMissingTasks 函数会首先分析其包含的分区数，对于每一个分区会为其生成一个 Task，然后同属于一个 Stage 的 Task 会被封装为 TaskSet，最后提交给 TaskScheduler。TaskSet 中包含了一组处理逻辑完全相同的 Task，只是作用数据不同。

4.2.2　TaskScheduler 的创建

对于每个 TaskScheduler 都会对应一个 SchedulerBackend。其中 TaskScheduler 负责程序的不同 job 之间的调度，SchedulerBackend 负责与集群资源管理器进行通信，取得应用所需要的资源，并且将资源传递给 TaskScheduler，由 TaskScheduler 为其最后分配计算资源。

TaskScheduler 和 SchedulerBackend 在 SparkContext 初始化的时候创建，不同的资源管理模式和部署模式下其对应的值不相同，在 SparkContext 初始化章节中分析了 Yarn-Cluster 模式下 TaskScheduler 和 SchedulerBackend 分别对应 YarnClusterScheduler 和 YarnClusterSchedulerBackend。这里 TaskScheduler 和 SchedulerBackend 都是接口，SchedulerBackend 负责分配当前可用的资源，具体就是向当前等待分配计算资源的 Task 分配计算资源。

4.2.3　Task 的提交

submitMissingTasks 中会通过 TaskScheduler.submitTasks 来提交 Tasks，TaskScheduler 提供的是一个接口，在不同模式下会有不同实现。在 Yarn-Cluster 模式下会调用 TaskSchedulerImpl.submitTasks(TaskSet)，其作用首先将保存这组任务的 TaskSet 加入到一个 TaskManager 中，TaskManager 会根据数据的 locality aware 为 Task 分配资源，并且也对 Task 的执行状态监控。接着为应用分配调度策略，主要包括 FIFO 和 FAIR。默认情况下为 FIFO，调度的单位为 TaskSet，优先调度 jobId 小的以及先到 rootPool 的 TaskSet。最后将调用 SchedulerBackend.reviveOffers()，实际调用 CoarseGrainedSchedulerBackend.reviveOffers(), 接着调用 CoarseGrainedSchedulerBackend.DriverEndpoint.makeOffers(), 遍历出每个 Executor 的资源封装之后，传入 TaskSchlerImpl.resourceOffers() 为每个 Task 分配具体的 Executor，分配资源部分代码如程序4.6所示

</> 程序 **4.6:** *TaskSchedulerImpl.resourceOffers* </>

```
//为了避免 Task 集中分配到某些机器上，随机打乱
val shuffledOffers = Random.shuffle(offers)
//存储分配好的 Task
val tasks = shuffledOffers.map(o => new
  ↪  ArrayBuffer[TaskDescription](o.cores))
val availableCpus = shuffledOffers.map(o => o.cores).toArray
val sortedTaskSets = rootPool.getSortedTaskSetQueue
for (taskSet <- sortedTaskSets) {
  taskSet.parent.name, taskSet.name, taskSet.runningTasks))
  if (newExecAvail) {
    taskSet.executorAdded()
  }
}
var launchedTask = false
for (taskSet <- sortedTaskSets; maxLocality <-
  ↪  taskSet.myLocalityLevels) {
  do {
    launchedTask = resourceOfferSingleTaskSet(taskSet, maxLocality,
      ↪  shuffledOffers, availableCpus, tasks)
  } while (launchedTask)
}
if (tasks.size > 0) {
  hasLaunchedTask = true
}
return tasks
```

程序返回的是 TaskDescription 的二维数组，TaskDescription 包含的是 TaskId、ExecutorId 和 Task 执行所需要的依赖环境信息等。

最后调用 CoarseGrainedSchedulerBackend.DriverEndPoint.launchTasks，它的输入为一个 TaskDescription 的二维数组，即上面程序段的返回值。此函数的作用先对 TaskDescription 进行序列化，然后更新 Executor 资源状况，空闲 cpu core 减去每个 Task 需要的 cpu core，最后将 Task 通过 Rpc 发送到 Executor，Executor 收到

这个消息后就会开始执行 Task。相应的代码如程序4.7所示

</>　程序 **4.7:**　</>

CoarseGrainedSchedulerBackend.DriverEndPoint.launchTasks

```
private def launchTasks(tasks: Seq[Seq[TaskDescription]]) {
  for (task <- tasks.flatten) {
  val serializedTask = ser.serialize(task)
  if (serializedTask.limit >= akkaFrameSize -
    ↪  AkkaUtils.reservedSizeBytes) {
    scheduler.taskIdToTaskSetManager.get(task.taskId).foreach {
      ↪  taskSetMgr =>
    try {
      taskSetMgr.abort(msg)
    } catch {
      case e: Exception => logError("Exception in error callback", e)
    }
  }
  }
  else {
    val executorData = executorDataMap(task.executorId)
    executorData.freeCores -= scheduler.CPUS_PER_TASK
    executorData.executorEndpoint.send(LaunchTask(new
      ↪  SerializableBuffer(serializedTask)))
    }
  }
}
```

Executor 执行 Task 会在下一章中进行介绍。

4.2.4　Task 计算结果的处理

Executor 在执行完 Task 时会向 Driver 发送 StatusUpdate 的消息来通知 Driver 任务的状态更新未 TaskState.FINISHED。Driver 会通知 TaskScheduler，TaskScheduler 会执行 TaskSchedulerImpl.statusUpdate，其代码如程序4.8所示

</>　程序 **4.8:** *TaskSchedulerImpl.statusUpdate*　</>

```
taskIdToTaskSetManager.get(tid) match {
  case Some(taskSet) =>
    if (TaskState.isFinished(state)) {
    taskIdToTaskSetManager.remove(tid)
    taskIdToExecutorId.remove(tid).foreach { execId =>
      if (executorIdToTaskCount.contains(execId)) {
        executorIdToTaskCount(execId) -= 1
     }
    }
  }
  if (state == TaskState.FINISHED) {
    taskSet.removeRunningTask(tid)
    taskResultGetter.enqueueSuccessfulTask(taskSet, tid,
      ↪  serializedData)
  } else if (Set(TaskState.FAILED, TaskState.KILLED,
    ↪  TaskState.LOST).contains(state)) {
    taskSet.removeRunningTask(tid)
    taskResultGetter.enqueueFailedTask(taskSet, tid, state,
      ↪  serializedData)
  }
  case None =>
    logError(
    ("Ignoring update with state %s for TID %s because its task set is
      ↪  gone (this is " +
    "likely the result of receiving duplicate task finished status
      ↪  updates)")
    .format(state, tid))
}
```

Task 只有在 FINISHED 状态下，才会被标记为成功，其他状态为执行失败。对于成功的 Task，Driver 会将 Task 加入到成功任务队列。Executor 执行 Task 成功返回的结果可能有两种情形，

1 直接结果

这种情况下直接返回结果即可

2 非直接结果

需要向远程 worker 网络获取结果，获取之后在远程节点上删除结果，这通过 Spark 中的 BlockManager 完成。

这两种结果的产生会在下章 Executor 执行 Task 章节讲述。

接着通过 TaskScheduleImpl.handleSuccessfulTask 来负责处理获取到的计算结果。这里调用栈如下所示

handleSuccessfulTask 调用栈

(1)TaskScheduleImpl.handleSuccessfulTask
(2)TaskSetManager.handleSuccessfulTask
(3)DAGScheduler.taskEnded
(4)DAGScheduler.handleTaskCompletion

其中最核心的为第 4 个，它会对不同的 Task 进行模式匹配，进行不同的处理，对于 ShuffleMapTask，程序代码如程序5.10所示

</>　程序 **4.9:** *ShuffleMapTask* 结果处理　</>

```
case smt: ShuffleMapTask =>
  val shuffleStage = stage.asInstanceOf[ShuffleMapStage]
  updateAccumulators(event)
  val status = event.result.asInstanceOf[MapStatus]
  val execId = status.location.executorId
  if (failedEpoch.contains(execId) && smt.epoch <=
    ↪  failedEpoch(execId)) {
  } else {
    shuffleStage.addOutputLoc(smt.partitionId, status)
  }
  if (runningStages.contains(shuffleStage) &&
    ↪  shuffleStage.pendingPartitions.isEmpty) {
    markStageAsFinished(shuffleStage)
    mapOutputTracker.registerMapOutputs(
    shuffleStage.shuffleDep.shuffleId,
    shuffleStage.outputLocInMapOutputTrackerFormat(),
```

```
15      changeEpoch = true)
16      clearCacheLocs()
17      if (!shuffleStage.isAvailable) {
18        shuffleStage.findMissingPartitions().mkString(", "))
19        submitStage(shuffleStage)
20      } else {
21        if (shuffleStage.mapStageJobs.nonEmpty) {
22          val stats =
              ↪  mapOutputTracker.getStatistics(shuffleStage.shuffleDep)
23          for (job <- shuffleStage.mapStageJobs) {
24            markMapStageJobAsFinished(job, stats)
25          }
26        }
27      }
28  }
```

首先会将整体结果注册到 MapOutputTracker 中去，这样下一个 Stage 的 Task 就可以方便的获取所需数据的元数据信息。接着判断当前 Stage 中是否所有任务都已成功返回，如果有部分数据是空的，那证明执行该分区的 Task 执行失败，需要重新提交 Stage；如果当前 Stage 执行成功并且没有父 Stage，那么就通过 submitWaitingStages() 将 waitingStage 集合中的 Stage 提交。

至此一个 Stage 就已经执行完成，如果 Stage 是 ResultStage，那么一个 job 就执行完成了。

Spark 中有两种 Task，ShuffleMapTask 和 ResultTask。ShuffleMapTask 根据 Task 的 Partition 存放计算结果，而 ResultTask 将计算结果发送到 Driver。用户程序出发 action 后，会调用 SparkContext 的 runJob 方法开始进行任务的提交。最后会通过 DAG 的事件处理器传递到 DAGScheduler.handleJobSubmitted，它会首先划分 Stage，然后提交 Task。至此，Task 开始在集群上运行了。Stage 的开始就是从外部存储或者 shuffle 结果中读取数据；一个 Stage 的结束就是由于发生 shuflle 或者生成结果。

第 5 章　Executor 执行 Task

5.1　Executor 启动

Yarn-Cluster 模式下 AM 中 SparkContext 初始化完成之后，AM 就开始在集群中划分资源，启动 container。

AM 通过循环等待 spark.yarn.am.waitTime 中定义的时间监听 SparkContext 的初始化状态，最后获得 SparkContext 的初始化实例。如果初始化实例为空，则报错并将应用状态置为失败。SparkContext 不为空的情况下，将会注册 AM。注册 AM 核心代码如程序5.1所示

</>　程序 **5.1:** *Driver 端 am 注册*　</>

```
allocator = client.register(driverUrl,
driverRef,
yarnConf,
_sparkConf,
uiAddress,
historyAddress,
securityMgr)
allocator.allocateResources()
reporterThread = launchReporterThread()
```

程序5.1中 client 其实为 YarnRMClient，是与 RM[①]负责通信的。client.register 的作用就是向 RM 中注册 AM，并获得 YarnAllocator 实例对象，此对象的作用体现在下一行代码。

程序的调用栈如下所示

(1)YarnAllocator#allocateResources()
(2)YarnAllocator#handleAllocatedContainers
(3)YarnAllocator#runAllocatedContainers

调用栈 (1) 负责和 RM 通信确保资源能够满足请求的资源，如果满足将会生成和请求资源最大 Executor 数量相等的 Container。调用栈 (2) 将会匹配 RM 所提供

① RM 为 ResourceManager 的简称，以后提到 RM 均为 ResourceManage

的 Container，并决定在其上启动 Executor。调用栈 (3) 是负责启动的模块，也是本程序的重点，其代码段如程序5.2所示，流程图如图5.1所示。

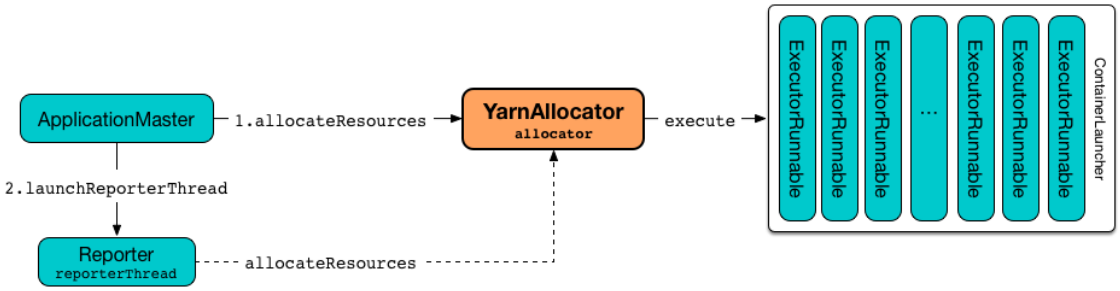


图 5.1　YarnAllocator 启动 Container 流程

</>　程序 **5.2:** *启动 Executor*　</>

```
for (container <- containersToUse) {
  numExecutorsRunning += 1
  val executorHostname = container.getNodeId.getHost
  val containerId = container.getId
  executorIdCounter += 1
  val executorId = executorIdCounter.toString
  executorIdToContainer(executorId) = container
  containerIdToExecutorId(container.getId) = executorId
  val containerSet =
    ↪  allocatedHostToContainersMap.getOrElseUpdate(executorHostname,
   new HashSet[ContainerId])
  containerSet += containerId
  allocatedContainerToHostMap.put(containerId, executorHostname)
  val executorRunnable = new
    ↪  ExecutorRunnable(container,conf,sparkConf,driverUrl,executorId,
  executorHostname,executorMemory,executorCores,
  appAttemptId.getApplicationId.toString,securityMgr)
  if (launchContainers) {
    launcherPool.execute(executorRunnable)
  }
}
```

targetNumExecutors 为 Executor 的总数，默认情况为 2，当用户没有开启动态分配并手动指定 targetNumExecutors 的数量时，系统会逐个 Executor 分配 Container。同时会将生成的 Executor 标识唯一的 ID，且与 Container 进行绑定，最后会通过 ExecutorRunnable 实例的 run 方法到 NM[①]中启动 Container。每个 Container 中包含了 Executor 的核数和内存大小，通过 Java 命令启动 Executor 实例即 CoarseGrainedExecutorBackend。CoarseGrainedExecutorBackend 的 main 方法首先会对启动它时传来的参数进行解析，只会会调用 run 方法，代码细节如程序5.3所示

</>　程序 **5.3:** *CoarseGrainedExecutorBackend.run*　</>

```
SignalLogger.register(log)
SparkHadoopUtil.get.runAsSparkUser { () =>
  val executorConf = new SparkConf
  val port = executorConf.getInt("spark.executor.port", 0)
  //这个 fetcher 通过 rpcenv 用来获得 driver 的 ref，之后就关闭了
  val fetcher = RpcEnv.create("driverPropsFetcher",hostname,port,
  executorConf,new SecurityManager(executorConf),clientMode = true)
  val driver = fetcher.setupEndpointRefByURI(driverUrl)
  //获得 Driver 端的配置信息
  val props = driver.askWithRetry[Seq[(String,
    ↪  String)]](RetrieveSparkProps) ++
  Seq[(String, String)](("spark.app.id", appId))
  fetcher.shutdown()
  // 使用从 Driver 端获取的配置信息创建 SparkEnv
  val driverConf = new SparkConf()
    for ((key, value) <- props) {
      if (SparkConf.isExecutorStartupConf(key)) {
        driverConf.setIfMissing(key, value)
      } else {
        driverConf.set(key, value)
      }
    }
  //这里如果是 yarn 模式下，他其实一直在跟新和 HDFS 交互的密钥信息
```

① NM 为 NodeManager 的简称，以后提到 NM 均为 NodeManage

```
23  if (driverConf.contains("spark.yarn.credentials.file")) {
24
      ↪  SparkHadoopUtil.get.startExecutorDelegationTokenRenewer(driverConf)
25  }
26  //创建 ExecutorEnv，代码和 Driver 端的基本一摸一样
27  val env = SparkEnv.createExecutorEnv(driverConf, executorId,
     ↪  hostname, port, cores, isLocal = false)
28  val sparkHostPort = env.conf.getOption("spark.executor.port").map {
     ↪  port =>
29   hostname + ":" + port}.orNull
30  //这里创建了一个 Endpoint 用于与 driver 交互执行 task。是由此可以
     ↪  看出 Executor 的真正实现 CoarseGrainedExecutorBackend
31  env.rpcEnv.setupEndpoint("Executor", new
     ↪  CoarseGrainedExecutorBackend(
32  env.rpcEnv, driverUrl, executorId, sparkHostPort, cores,
     ↪  userClassPath, env))
33  //WorkerWatcher 就是用来监视 worker 的，当与 worker 断开连接时，它
     ↪  将会关闭这个 jvm 进程
34  workerUrl.foreach { url =>
35   env.rpcEnv.setupEndpoint("WorkerWatcher", new
       ↪  WorkerWatcher(env.rpcEnv, url))
36  }
37  env.rpcEnv.awaitTermination()
38  SparkHadoopUtil.get.stopExecutorDelegationTokenRenewer()
39 }
```

Executor 创建于 Driver 交互的 RPC 环境后，会立刻执行 CoarseGrainedExecutorBackend 的 onStart() 方法，该方法最重要的作用是向 Driver 注册 Executor 的信息，主要为 executorId 和 cores 等。注册的消息会在 CoarseGrainedSchedulerBackend#receiveAndReply 中接收，在 Driver 端会通过 executorDataMap(HashMap[String,ExecutorData]) 数据结构进行存储，TaskScheduler 对将 task 封装为 TaskDescription 时会用到 executorDataMap 中的信息。

5.2　Task 的执行

第4章 Driver 通过 CoarseGrainedSchedulerBackend.DriverEndPoint.launchTasks 会将 Task 分配到 Executor 上，这里通过的是 nettyRpc。

CoarseGrainedExecutorBackend 收到 LaunchTask 消息后，会调用 Executor 的 launchTask 启动 Task，如程序5.4

</>　程序 **5.4:** *CoarseGrainedExecutorBackend 消息处理模块*　</>

```
case LaunchTask(data) =>
  if (executor == null) {
    logError("Received LaunchTask command but executor was null")
    System.exit(1)
  } else {
    val taskDesc = ser.deserialize[TaskDescription](data.value)
    logInfo("Got assigned task " + taskDesc.taskId)
    executor.launchTask(this, taskId = taskDesc.taskId, attemptNumber
     ↪  = taskDesc.attemptNumber,
    taskDesc.name, taskDesc.serializedTask)
}
```

Executor.launchTask 方法如程序块5.5所示

</>　程序 **5.5:** *Executor.launcherTask*　</>

```
val tr = new TaskRunner(context, taskId = taskId, attemptNumber =
  ↪  attemptNumber, taskName,
serializedTask)
runningTasks.put(taskId, tr)
threadPool.execute(tr)
```

Executor 会输入的 Task 生成一个 TaskRunner，最终会被放到一个 ThreadPool 中去。TaskRunner 为一个线程，重写其 run 方法，run 中会执行 Task。

5.2.1　Task 执行前的准备

Driver 端生成 Task 的时候会将 Task 依赖的文件和 jar 信息都封装到 Task 中去，在 Executor 端，得先恢复这些信息。

```
val (taskFiles,taskJars,taskBytes) = Task.deserializeWithDependencies(serializedTask)
updateDependencies(taskFiles, taskJars)
```

这里返回了三元组，信息包含了 taskFiles、taskJars 和 taskBytes，这些信息只是属于类似于元数据的信息，接下来 Executor 还需要调用 updateDependencies 下载这些依赖：

```
// Fetch file with useCache mode, close cache for local mode.
Utils.fetchFile(name, new File(SparkFiles.getRootDirectory()), conf,
env.securityManager, hadoopConf, timestamp, useCache = !isLocal)
```

依赖下载完成后，Executor 会通过 TaskRunner 来执行 Task。

5.2.2　执行 Task

准备工作做好后，Executor 就是执行 Task.run 方法来运行 Task，如程序5.6所示

</>　程序 **5.6:** *Execotor 执行 Task*　</>

```
//开始执行 Task
taskStart = System.currentTimeMillis()
var threwException = true
val (value, accumUpdates) = try {
  val res = task.run(
  taskAttemptId = taskId,
  attemptNumber = attemptNumber,
  metricsSystem = env.metricsSystem)
  threwException = false
  res
} finally {
  ......
}
val taskFinish = System.currentTimeMillis()
```

程序5.6中 task.runTask 会调用 Task.run 方法，此方法为 final 类型，首先会本次 Task 生成 TaskContext，也就是保留本 Task 的上下文信息。最后 TaskContext 还

会在 Task 执行完成后对次 Task 标记成功并通过回调函数来完成最终的处理，如程序5.7所示

</>　程序 **5.7:** *TaskContext* 任务成功时执行回调　</>

```
private[spark] def markTaskCompleted(): Unit = {
  completed = true
  val errorMsgs = new ArrayBuffer[String](2)
  onCompleteCallbacks.reverse.foreach { listener =>
  try {
    listener.onTaskCompletion(this)
  } catch {
    case e: Throwable =>
      errorMsgs += e.getMessage
    }
  }
  if (errorMsgs.nonEmpty) {
    throw new TaskCompletionListenerException(errorMsgs)
  }
}
```

整个 Task.run 的核心程序如5.8所示

</>　程序 **5.8:** *Task run* 方法核心部分　</>

```
//设置上下文信息，实际上会调用
  ↪  org.apache.spark.TaskContext#setTaskContext
TaskContext.setTaskContext(context)
//更新 metrics 信息
context.taskMetrics.setHostname(Utils.localHostName())
context.taskMetrics.setAccumulatorsUpdater(context.collectInternalAccumulators)
//当前线程，在被打断的时候可以通过它来停止该线程
taskThread = Thread.currentThread()
if (_killed) {//如果当前 Task 被杀死，那么需要退出 Task 的执行
  kill(interruptThread = false)
}
```

```
try {
  //执行本次 Task，runTask 真正执行代码为子类中重写的 runTask，对应 ShuffleMapTask 和 ResultTask
  (runTask(context), context.collectAccumulators())
} catch {
} finally {
  // Call the task completion callbacks.
  context.markTaskCompleted()
  try {
  }
  } finally {
    TaskContext.unset()
  }
}
```

这里的 runTask 会有两种不同的实现方法，分别为 ShuffleMapTask 和 ResultTask，两种 Task 前期阶段处理基本相同，如程序5.9所示。

</>　程序 **5.9:** 两种 *Task* 共同的处理方式　</>

```
// Deserialize the RDD using the broadcast variable.
val deserializeStartTime = System.currentTimeMillis()
//获取反序列化实例
val ser = SparkEnv.get.closureSerializer.newInstance()
//以下两端代码是其区别，上面的为 ShuffleMapTask，下面为 ResultTask
val (rdd, dep) = ser.deserialize[(RDD[_], ShuffleDependency[_, _, _])](
//获取 rdd 和作用于 rdd 结果的函数
val (rdd, func) = ser.deserialize[(RDD[T], (TaskContext, Iterator[T]) => U)](

ByteBuffer.wrap(taskBinary.value), Thread.currentThread.getContextClassLoader)
```

```
_executorDeserializeTime = System.currentTimeMillis() -
  ↪ deserializeStartTime
//Task 的测量信息
metrics = Some(context.taskMetrics)
```

1 对于 ShuffleMapTask

ShuffleMapTask 是根据 Stage 中分区数量来生成的，它会根据该 Stage 每个 RDD 连接操作作用于对应的 partition 上，并将结果生成文件供下游 Task 使用。ShuffleMapTask 核心代码段如程序5.10所示

</>　程序 **5.10:** *ShuffleMapTask* 核心部分实现分析　</>

```
//从 SparkEnv 中获得 shuffleManager，具体分析会在 shuffle 模块讲
  ↪ 述
val manager = SparkEnv.get.shuffleManager
//从 shuffleManager 获得写入器
writer = manager.getWriter[Any, Any](dep.shuffleHandle,
  ↪ partitionId, context)
//调用 rdd 开始计算，并且最后通过写入器写入文件系统
writer.write(rdd.iterator(partition,
  ↪ context).asInstanceOf[Iterator[_ <: Product2[Any, Any]]])
//关闭写入器，并将最后结果返回
writer.stop(success = true).get
```

2 对于 ResultTask

ResultStage 会根据生成结果的 Partition 来生成与 Partition 数量相同的 ResultTask，最后会将计算结果回传给 Driver 端，这里的结果不一定是真实的数据。ResultTask 的作用代码如程序5.11所示。

</>　程序 **5.11:** *ResultTask* 核心部分实现分析　</>

```
//调用 rdd 的迭代器执行传入函数的操作
func(context, rdd.iterator(partition, context))
```

5.2.3　Task 结果的处理

Executor 运行完 Task 后，就会将结果保存在 DirectTaskResult 里，如程序5.12所示

</>　程序 **5.12:** *Executor 对 Task 结果的处理*　</>

```
//任务运行结果的处理
val resultSer = env.serializer.newInstance()
val beforeSerialization = System.currentTimeMillis()
//value 中保存 Task 的计算结果
val valueBytes = resultSer.serialize(value)
val afterSerialization = System.currentTimeMillis()
//首先将结果直接放入 org.apache.spark.scheduler.DirectTaskResult
val directResult = new DirectTaskResult(valueBytes, accumUpdates,
  ↪  task.metrics.orNull)
val serializedDirectResult = ser.serialize(directResult)
val resultSize = serializedDirectResult.limit
```

但是 serializedDirectResult 并不是直接回传给 Driver，Executor 通过程序5.13的策略对结果进行不同处理。

</>　程序 **5.13:** *Executor 回传结果策略*　</>

```
// directSend = sending directly back to the driver
val serializedResult: ByteBuffer = {
if (maxResultSize > 0 && resultSize > maxResultSize) {
  //如果 resultSize 大于 spark.drive.driver.maxResultSize 设置的大小
    ↪  值，则直接丢弃
  ser.serialize(new IndirectTaskResult[Any](TaskResultBlockId(taskId),
    ↪  resultSize))
} else if (resultSize >= akkaFrameSize - AkkaUtils.reservedSizeBytes)
  ↪  {
  //如果不能通过 AKKA 的消息传递，那么放入 BlockManager 等待调用者以
    ↪  网络形式来获取
  //AKKA 的消息默认大小可以通过 spark.akka.frameSize 来设置
  val blockId = TaskResultBlockId(taskId)
```

```
  env.blockManager.putBytes(
  blockId, serializedDirectResult, StorageLevel.MEMORY_AND_DISK_SER)
  ser.serialize(new IndirectTaskResult[Any](blockId, resultSize))
} else {
  //结果回传 Driver
  serializedDirectResult
 }
}
//通过 AKKA 向 driver 汇报本次 Task 已经完成
execBackend.statusUpdate(taskId, TaskState.FINISHED, serializedResult)
```

而 execBackend.statusUpdate 实际上调用的 CoarseGrainedExecutorBackend.statusUpdate, 程序如5.14所示

</>　程序 **5.14:** *Executor* 执行回传消息　</>

```
val msg = StatusUpdate(executorId, taskId, state, data)
driver match {
  case Some(driverRef) => driverRef.send(msg)
  case None => logWarning(s"Drop $msg because has not yet connected to
    ↪ driver")
}
```

这里通过 DriverRef 通过 Rpc 机制实际会调用 TaskschedulerImpl.statusUpdate。Driver 端对结果的处理见小节4.2.4。

5.3 Executor 生命周期

本章主要介绍 Executor 启动的过程、Task 的分发、Task 的运行以及 Executor 对 Task 结果的处理。Executor 运行 Task 这个过程的时序图如图5.2所示

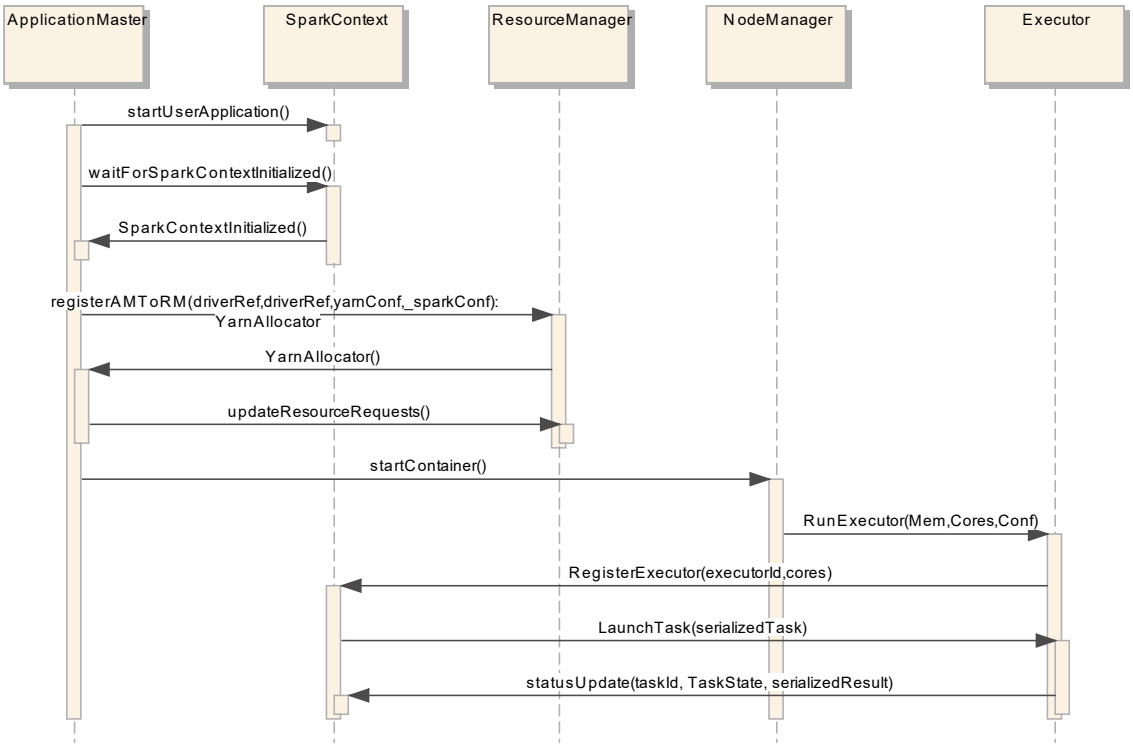


图 5.2　Executor 运行 Task 时序图

第 6 章　Spark Shuffle 分析

前面讲解了 map 端和 reduce 端对 Task 的不同处理方式，连接它们的桥梁即为 Shuffle 模块，shuffle 用于打通 map 任务的输出与 reduce 任务的输入，map 任务计算的中间输出结果按照 key 值哈希后分配给某一个 reduce 任务。

shuffle 过程数据的传输情况可能非常负责

1 数据量大；

2 单个 Executor 内存不足以放下处理完的数据时需要进行磁盘读写；

3 数据传输过程中需要压缩和解压缩；

4 跨节点传输时需要通过网络；

正因为这众多的因素，shuffle 无疑是调优的重点，理解 shuffle 的版本历史有助于了解 shuffle 优化的思路。

Shuffle 的类型是在 SparkContext 初始化时确定的，Spark1.6 版本如程序6.1所示

</>　程序 **6.1:** *ShuffleManager* 的初始化　</>

```
//   指定 shuffle 的类型
val shortShuffleMgrNames = Map(
"hash" -> "org.apache.spark.shuffle.hash.HashShuffleManager",
"sort" -> "org.apache.spark.shuffle.sort.SortShuffleManager",
"tungsten-sort" -> "org.apache.spark.shuffle.sort.SortShuffleManager")
val shuffleMgrName = conf.get("spark.shuffle.manager", "sort")
val shuffleMgrClass =
  ↪  shortShuffleMgrNames.getOrElse(shuffleMgrName.toLowerCase,
  ↪  shuffleMgrName)
val shuffleManager = instantiateClass[ShuffleManager](shuffleMgrClass)
```

可以看出 ShuffleManager 默认类型为 SortShuffleManager。

在 Executor 上执行 ShuffleMapTask 时，最终会调用 ShuffleMapTask.runTask。核心逻辑如程序6.2所示

</>　程序 **6.2:** *ShuffleMapTask 获取 ShuffleWriter*　</>

```
//从 SparkEnv 中获取 shuffleManager
val manager = SparkEnv.get.shuffleManager
//从 manager 中获取 Writer，这里是 SortShuffleWriter
writer = manager.getWriter[Any, Any](dep.shuffleHandle, partitionId,
  ↪ context)
//调用 RDD 开始运算，运算结果通过 Writer 进行持久化，之后将文件所有
  ↪ 记录写入并创建索引文件，通过 MapStatus 告知下游 Task
writer.write(rdd.iterator(partition, context).asInstanceOf[Iterator[_
  ↪ <: Product2[Any, Any]]])
writer.stop(success = true).get
```

6.1　Shuffle Writer

6.1.1　Hash Based Shuffle Write

在 Spark1.0 以前，由于不要求数据有序，shuffle write 的任务很简单：将数据 partition 好，并持久化。之所以要持久化，一方面是要减少内存存储空间压力，另一方面也是为了 fault-tolerance。

shuffle write 的任务很简单，那么实现也很简单：将 shuffle write 的处理逻辑加入到 ShuffleMapStage（ShuffleMapTask 所在的 stage）的最后，该 stage 的 final RDD 每输出一个 record 就将其 partition 并持久化。此种类型的 Writer 工作方式如图6.1所示, 从图中可以看出 4 个 ShuffleMapTask 要在同一个 worker node 上运行，CPU core 数为 2，可以同时运行两个 task。每个 task 的执行结果（该 stage 的 finalRDD 中某个 partition 包含的 records）被逐一写到本地磁盘上。每个 task 包含 R 个缓冲区，R = reducer 个数（也就是下一个 stage 中 task 的个数），缓冲区被称为 bucket[①]，其大小为 spark.shuffle.file.buffer.kb ，默认是 32KB（Spark 1.1 版本以前是 100KB）。

① 其实 bucket 是一个广义的概念，代表 ShuffleMapTask 输出结果经过 partition 后要存放的地方，这里为了细化数据存放位置和数据名称，仅仅用 bucket 表示缓冲区。

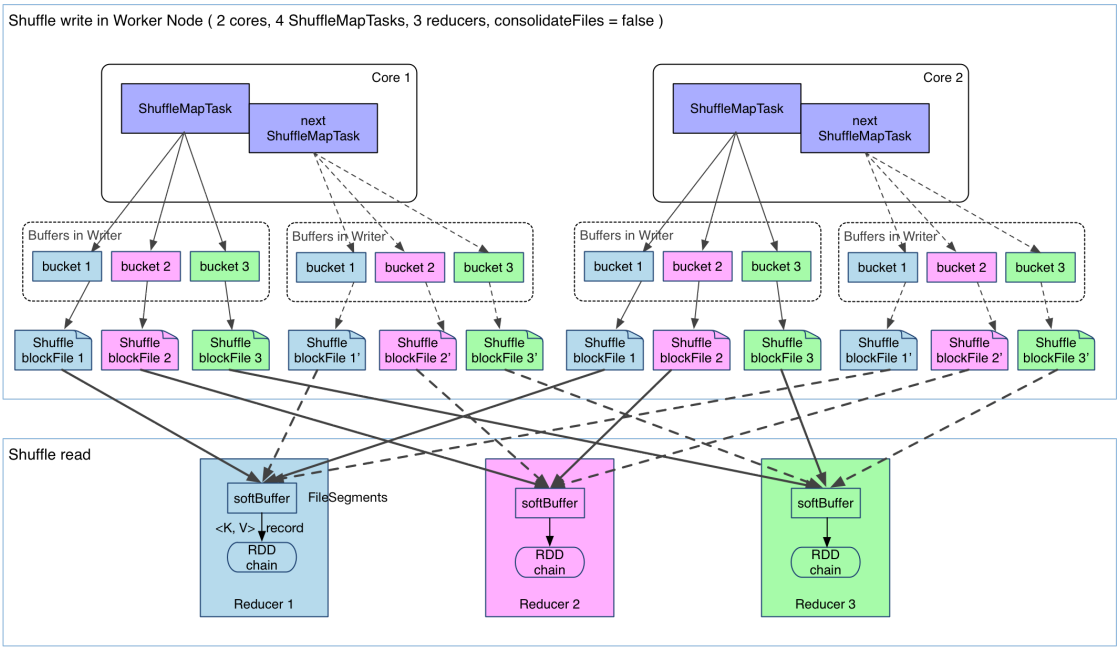


图 6.1　早期的 ShuffleWriter 工作原理

6.1.2　Hash Based Shuffle Write 存在的问题

由于 ShuffleMapTask 需要为每个下游的 Task 创建一个单独的文件，因此文件的数量就是 M*R[①]个。如果单机 28 个 cpu core，假设节点上 20 个 core 用于 ShuffleMapTask，R 数量为 1000 个，那么逻辑上会有 20000 个文件。在实际运行中，Task 的数量会更多，这种 ShuffleWriter 的实现会带来一下问题：

1 缓冲区占用内存空间大

每个节点都可能同时打开多个文件，每次打开文件都会占用一定内存。如上面的 20000 个文件，每个 writer Handler 默认需要 100KB 内存，单个节点就可能达到 2GB 的内存，当 Map 端和 Reduce 同时增加 10 倍时，整体的内存会更吓人。

2 系统随机读增多

当系统存在很多文件比较小但数量比较多的时候，而且机械硬盘在随机读方面的性能特别差，非常容易出现性能瓶颈。

6.1.3　Shuffle Consolidate Writer

为了解决上一小节中 Shuffle 产生文件较多的问题，之后的 Shuffle 加入了 Consolidate Files 机制，它的目标就是减少 Shuffle 过程文件产生过多的问题，当然第一个问题还是没有解决。它的思想是针对同一个核运行多个 ShuffleMapTask 的情况

① M 为 shuffleMapTask 的数量，R 为 reduce 端 Task 的数量

下，众多的 ShuffleMapTask 会将记录写到同一个文件中，下一个会以追加的方式写入而不是新建文件。这样文件数就变为 core[①]*R 个文件。此种模式下运行原理如图6.2所示

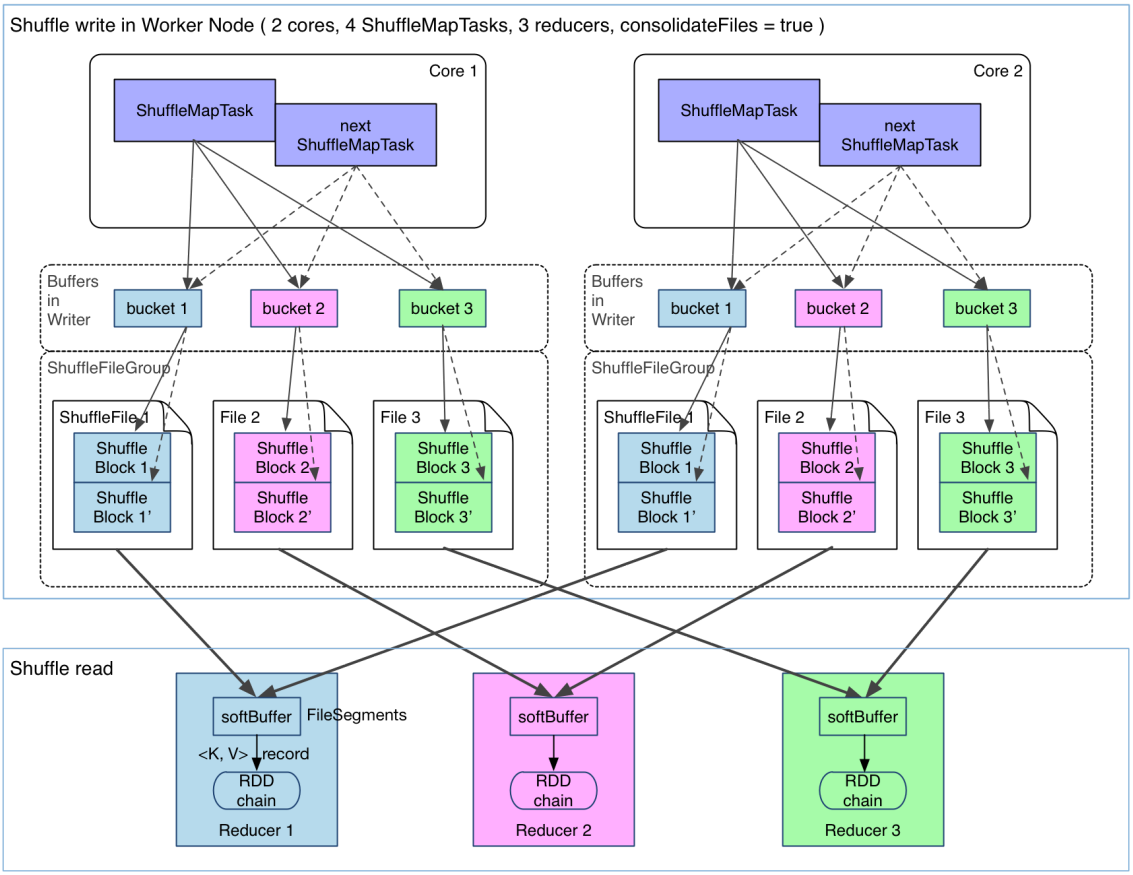


图 6.2　Shuffle Consolidate Files 工作原理

不过这种模式下当每个 core 只运行一个 ShuffleMapTask 时，那么就和原来的机制一样了。但是当 ShuffleMapTask 明显多于 core 数量时，这种模式下可以显著减少文件的数量。这里还剩下一个问题，下游的 Task 如何区分文件的不同部分？这部分在后面小节中讲述。

针对 Shuffle Consolidate 遗留的问题，Spark 重新建立了一套 Shuffle 机制。也就是下面讲述的 SortShuffle，这个已经是当前 Spark 版本的默认选项。

6.1.4　Sort Based Write

此方法的选择是在 org.apache.spark.SparkEnv 下完成的，本章开头部分已经说明。首先，每个 ShuffleMapTask 不会为每个 Reducer 生成一个单独的文件相反，它

① 这里指核的数量

会将所有的结果写到一个文件里，同时生成一个 Index 文件，Reducer 可以通过这个 Index 文件取得它需要处理的数据，减少了文件的数量。

Shuffle Map Task 会按照 key 相对应的 partition id 进行排序，对于属于同一个 partition 的 keys 可选的进行或不进行排序。因为对于不需要排序的操作来说，这个排序是有损性能的。对于那些需要 Sort 的操作，比如 sortByKey，这个排序是由 Reducer 完成的。Sort Base Shuffle 原理如图6.3所示

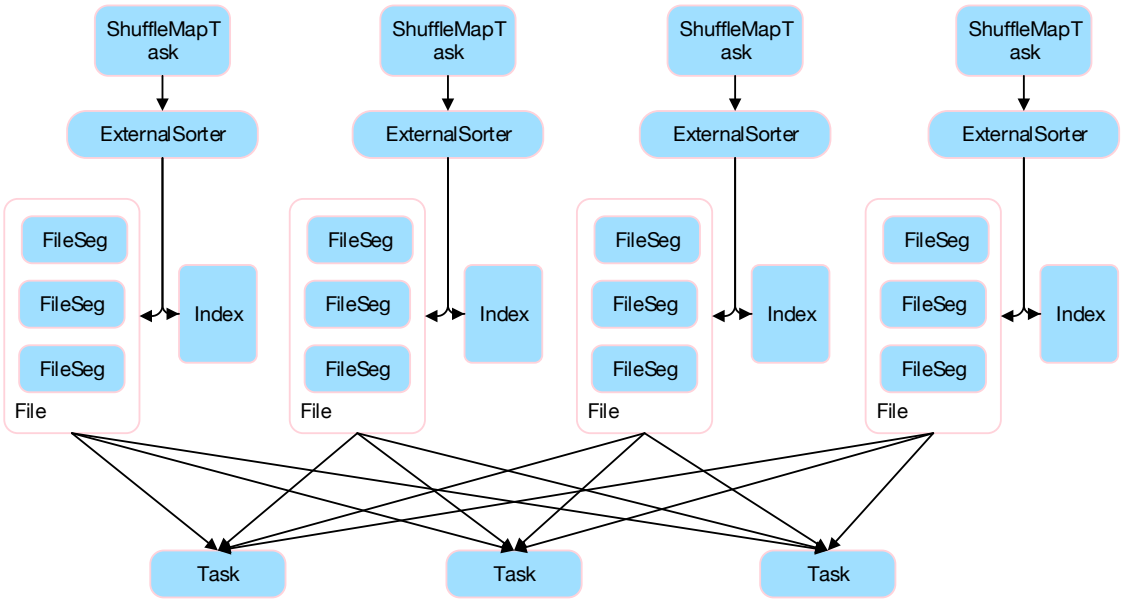


图 6.3　Sort Base Shuffle 工作原理

为了下游 Task 获取到其所需要的分区，生成文件的同时会附加 Index 文件，来记录不同分区的位置信息。

6.2　Shuffle Read

在 Stage 的边界，要计算 ShuffledRDD 中的数据，必须先把 MapPartitionsRDD 中的数据 fetch 过来。而下边界，要么将文件写入本地文件系统，以供子 Stage 读取，要么是最后一个 Stage，输出结果。

下游 Stage 的第一个 RDD 即为 ShuffledRDD，其 compute 方法中会调用 shuffleManager.getReader 来读取上游的分区。全部 ShuffleMapTasks 执行完再去 fetch。因为 fetch 来的 FileSegments 要先在内存做缓冲，所以一次 fetch 的 FileSegments 总大小不能太大。Spark 规定这个缓冲界限不能超过 ăspark.reducer.maxMbInFlight，这里用 ăsoftBufferă 表示，默认大小为 48MB。核心的 read 实现如程序6.3所示

</>　程序 **6.3:** *Shuffle Read* 的实现　</>

```
override def read(): Iterator[Product2[K, C]] = {
val blockFetcherItr = new
  ↪  ShuffleBlockFetcherIterator(context,blockManager.shuffleClient,
blockManager,mapOutputTracker.getMapSizesByExecutorId(handle.shuffleId,
  ↪  startPartition, endPartition),
 SparkEnv.get.conf.getSizeAsMb("spark.reducer.maxSizeInFlight",
   ↪  "48m") * 1024 * 1024)
 ......
 //获取序列化器
 val ser = Serializer.getSerializer(dep.serializer)
 ......
 val interruptibleIter = new InterruptibleIterator[(Any,
   ↪  Any)](context, metricIter)
 val aggregatedIter: Iterator[Product2[K, C]] = if
   ↪  (dep.aggregator.isDefined) {//需要聚合
   if (dep.mapSideCombine) {//需要 map 端聚合
     val combinedKeyValuesIterator =
       ↪  interruptibleIter.asInstanceOf[Iterator[(K, C)]]
     dep.aggregator.get.combineCombinersByKey(
     combinedKeyValuesIterator, context)
   } else {//只需在 reduce 端聚合
     val keyValuesIterator =
       ↪  interruptibleIter.asInstanceOf[Iterator[(K, Nothing)]]
     dep.aggregator.get.combineValuesByKey(keyValuesIterator,
       ↪  context)
   }
 } else {//不需要聚合
   interruptibleIter.asInstanceOf[Iterator[Product2[K, C]]]
 }
 dep.keyOrdering match {//判断是否需要排序
   case Some(keyOrd: Ordering[K]) =>
```

```
24    //对于需要排序的情况使用 ExternalSorter 进行排序，注意如果
        ↪ spark.shuffle.spill 是 false，那么数据不会写到磁盘
25      val sorter =new ExternalSorter[K, C, C](context, ordering =
          ↪ Some(keyOrd), serializer = Some(ser))
26      sorter.insertAll(aggregatedIter)
27      ......
28      CompletionIterator[Product2[K, C], Iterator[Product2[K,
          ↪ C]]](sorter.iterator, sorter.stop())
29    case None =>aggregatedIter
30    //无需排序
31  }
32 }
```

一个 ShuffleMapStage 形成后，会将该 stage 最后一个 final RDD 注册到 MapOutputTrackerMaster.registerShuffle(shuffleId, rdd.partitions.size)，这一步很重要，因为 shuffle 过程需要 MapOutputTrackerMaster 来指示 ShuffleMapTask 输出数据的位置”。因此，reducer 在 shuffle 的时候是要去 driver 里面的 MapOutputTrackerMaster 询问 ShuffleMapTask 输出的数据位置的，它会调用 MapOutputTracke.getStatuses 来获得数据的 meta 数据信息，获得 meta 数据信息后，它会将这些数据存入 Seq[(BlockManagerId, Seq[(BlockId, Long)])] 中，然后调用 ShuffleBlockFetcherIterator 最终发起请求。

6.2.1　reduce 端读取中间计算结果

ShuffleBlockFetcherIterator 会调用 splitLocalRemoteBlocks 来划分数据的读取方式，主要是本地数据和远程节点数据。这里 Spark 限制每次最多启动 5 个线程到最多 5 个节点上读取数据，同时通过 spark.reducer.maxMbInFlight 来适应网络带宽。该部分具体情况如程序6.4所示

</>　程序 **6.4:** *reduce* 读取数据策略划分　</>

```
1 private[this] def splitLocalRemoteBlocks():ArrayBuffer[FetchRequest]={
2   val targetRequestSize = math.max(maxBytesInFlight / 5, 1L)
3   val remoteRequests = new ArrayBuffer[FetchRequest]
4   var totalBlocks = 0
```

```
  for ((address, blockInfos) <- blocksByAddress) {
    totalBlocks += blockInfos.size
    if (address.executorId == blockManager.blockManagerId.executorId)
      ↪  {//本地获取 local
    //Block 在本地，需要过滤大小为 0 的 Block
    localBlocks ++= blockInfos.filter(_._2 != 0).map(_._1)
    numBlocksToFetch += localBlocks.size
    } else {//远程获取 remote
      val iterator = blockInfos.iterator
      var curRequestSize = 0L
      var curBlocks = new ArrayBuffer[(BlockId, Long)]
      while (iterator.hasNext) {//blockId 是 ShuffleBlockId
        val (blockId, size) = iterator.next()
        if (size > 0) {
          curBlocks += ((blockId, size));remoteBlocks += blockId
          numBlocksToFetch += 1;curRequestSize += size
        } else if (size < 0) {
          throw new BlockException(blockId, "block size " + size)
        }
        if (curRequestSize >= targetRequestSize) {
          //当前总的 Size 已经可以批量放入一次 Request 中
          remoteRequests += new FetchRequest(address, curBlocks)
          curBlocks = new ArrayBuffer[(BlockId, Long)]
          curRequestSize = 0
        }
      }
      if (curBlocks.nonEmpty) {//剩余的请求组成一次 Request
        remoteRequests += new FetchRequest(address, curBlocks)
      }
    }
  }
  remoteRequests
}
```

程序6.4是用于划分哪些 Block 从本地获取，哪些需要远程拉取，是获取中间计算结果的关键。程序中下列变量比较重要

1 targetRequestSize 每个远程请求块的最大尺寸

2 totalblock 统计 Block 总数

3 localBlocks:ArrayBuffer[BlockId], 缓存本地获取的 Block 的 BlockId 序列

4 remoteBlocks:HashSet[BlockId]，缓存远程获取的 Block 的 BlockId 序列

5 curBlocks:ArrayBuffer[(BlockId, Long)]，远程获取的累加缓存，用于保证每个远程请求的尺寸不超过 targetRequestSize。

6 curRequestSize 当前 curBlocks 中的所有 Block 的大小之和，用于保证每个远程请求尺寸不超过 targetRequestSize

7 remoteRequests:ArrayBuffer[FetchRequest]，缓存需要远程请求的 FetchRequest 对象

8 numBlocksToFetch，一共要获取的 Block 数量

9 maxBytesInFlight，单次航班请求的最大字节数。这批请求的总字节数不能超过 maxBytesInFlight，并且每一个请求的字节数不能超过 maxBytesInFlight 的五分之一。

6.2.2　本地数据的获取

上一小节中已经将本地数据块放入 localBlocks，对于本地块中的数据将通过 fetchLocalBlocks 进行获取。具体过程如程序6.5所示

</>　程序 **6.5:** 获取本地块　</>

```
val iter = localBlocks.iterator
while (iter.hasNext) {
  val blockId = iter.next()
  try {
    val buf = blockManager.getBlockData(blockId)
    shuffleMetrics.incLocalBlocksFetched(1)
    shuffleMetrics.incLocalBytesRead(buf.size)
    buf.retain()
    results.put(new SuccessFetchResult(blockId,
      ↪  blockManager.blockManagerId, 0, buf))
  } catch {
```

```
  }
}
```

这里调用了 BlockManager.getBlockData 获取数据，对于 shuffle 数据其会调用对应的 ShuffleManager.shuffleBlockResolver.getBlockData 方法获取数据，其实现如程序6.6所示

</>　程序 **6.6:** *SortShuffleManager 读取 shuffle 数据实现*　</>

```
//根据 ShuffleId 和 MapId 获得索引文件
val indexFile = getIndexFile(blockId.shuffleId, blockId.mapId)
val in = new DataInputStream(new FileInputStream(indexFile))
try {
  ByteStreams.skipFully(in,blockId.reduceId*8)//本次 Block 的数据区
  val offset = in.readLong()//开始
  val nextOffset = in.readLong()//结束
  new FileSegmentManagedBuffer(transportConf,getDataFile(
  blockId.shuffleId, blockId.mapId),offset,nextOffset - offset)
}
```

可以看出它是根据索引文件来获得数据块在数据文件中的具体位置信息的。

6.2.3　远程数据的获取

splitLocalRemoteBlocks　返回值为远程数据块，ShuffleBlockFetcherIterator.fetchUpToMaxBytes 来发送远程获取请求。通过 sendRequest 发送读取 Block 的请求，其中包含中两个回调函数，分别对应着请求成功和请求失败，具体如程序6.7所示

</>　程序 **6.7:** *向远程节点发送读取 Block 请求*　</>

```
shuffleClient.fetchBlocks(address.host, address.port,
  ↪  address.executorId, blockIds.toArray,
new BlockFetchingListener {
  override def onBlockFetchSuccess(blockId: String, buf:
    ↪  ManagedBuffer): Unit = {//请求成功
    if (!isZombie) {
```

```
      buf.retain()
      results.put(new SuccessFetchResult(BlockId(blockId), address,
        ↪ sizeMap(blockId), buf))
    }
  }
  override def onBlockFetchFailure(blockId: String, e: Throwable):
    ↪ Unit = {
    results.put(new FailureFetchResult(BlockId(blockId), address, e))
  }
})
```

程序中的 shuffleClient 就是 blockTransferService，如程序6.8所示

</>　程序 **6.8:** *BlockManager 获取 shuffleClient*　</>

```
val externalShuffleServiceEnabled =
  ↪ conf.getBoolean("spark.shuffle.service.enabled", false)
//externalShuffleServiceEnabled 默认为 false
private[spark] val shuffleClient = if (externalShuffleServiceEnabled)
  ↪ {
  val transConf = SparkTransportConf.fromSparkConf(conf, "shuffle",
    ↪ numUsableCores)
  new ExternalShuffleClient(transConf, securityManager,
    ↪ securityManager.isAuthenticationEnabled(),
  securityManager.isSaslEncryptionEnabled())
} else {
  blockTransferService
}
//此部分在 SparkEnv 里定义，直接通过 netty 的方式
val blockTransferService = new NettyBlockTransferService(conf,
  ↪ securityManager, numUsableCores)
```

6.2.4　Shuffle Read 流程图

整个 shuffle read 的执行过程如图6.4所示

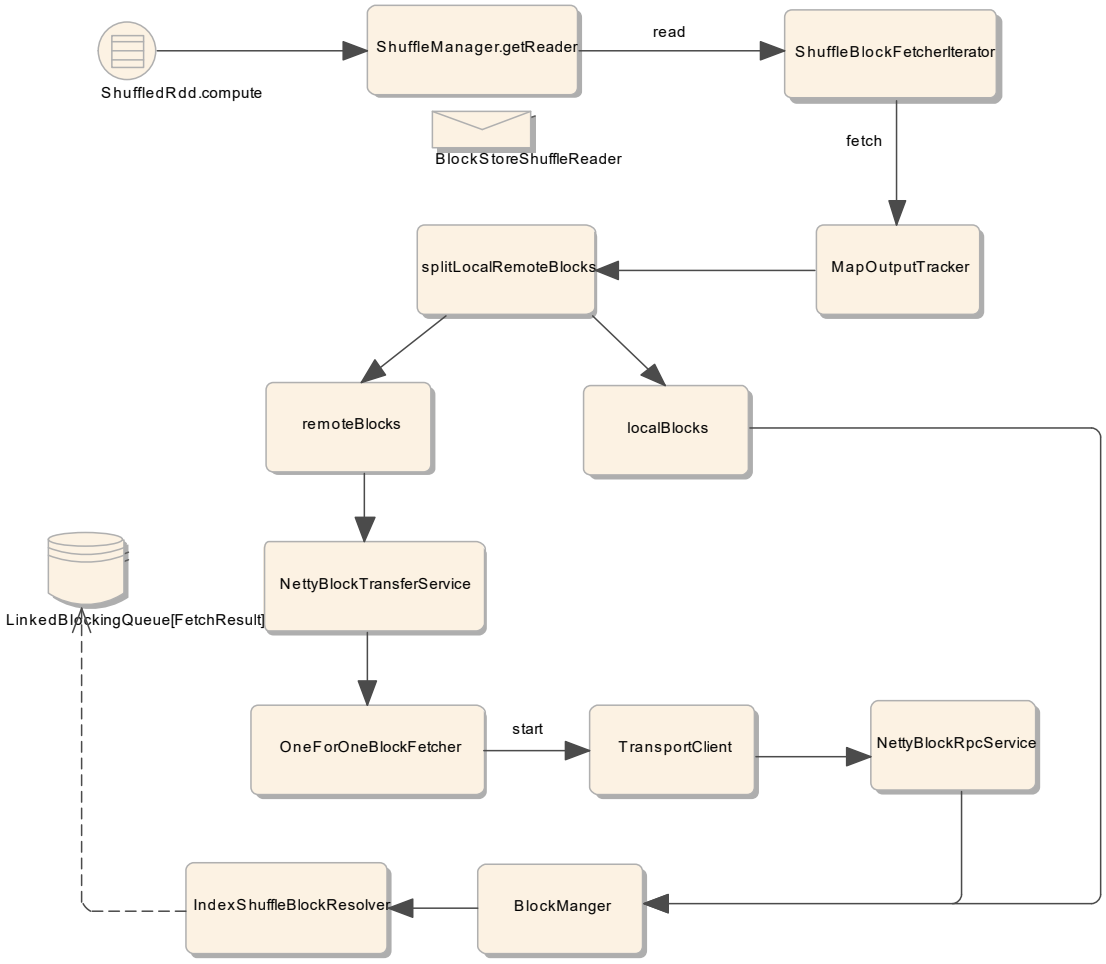


图 6.4　shuffleread 详细实现

第 7 章　存储体系

7.1　概述

块管理器 BlockManager 是 Spark 存储体系中的核心组件，其初始化在 SparkEnv 创建过程中，会首先创建 BlockManagerMaster，之后再创建 BlockManager，最后会在 SparkContext 中 taskscheduler 初始化完成后，调用 _env.blockManager.initialize(_applicationId) 完成初始化工作如程序7.1所示

</>　程序 **7.1:** *BlockManager* 的初始化　</>

```
//管理所有数据，无论在哪的数据，创建 BlockManager
val blockManager = new BlockManager(executorId, rpcEnv,
  ↪ blockManagerMaster,serializer, conf, memoryManager,
  ↪ mapOutputTracker, shuffleManager,blockTransferService,
  ↪ securityManager, numUsableCores)
//blockTransferService 的初始化和 shuffleClient 的初始化，
  ↪ shuffleClient 默认是 NettyBlockTransferService
blockTransferService.init(this)
shuffleClient.init(appId)
blockManagerId = BlockManagerId(
executorId, blockTransferService.hostName, blockTransferService.port)
//当有外部的 ShuffleService 时，创建新的 BlockManagerId，否则就利用现
  ↪ 有的 blockManagerId
shuffleServerId = if (externalShuffleServiceEnabled) {
  BlockManagerId(executorId, blockTransferService.hostName,
    ↪ externalShuffleServicePort)
} else {
  blockManagerId
}
//向 BlockManagerMaster 注册 blockManagerId slaveEndpoint
  master.registerBlockManager(blockManagerId, maxMemory,
    ↪ slaveEndpoint)
if (externalShuffleServiceEnabled && !blockManagerId.isDriver) {
```

```
17   registerWithExternalShuffleServer()
18 }
```

程序中在向 BlockManagerMaster 注册的时候会向判断当前进程是否为 Driver。这是因为 Driver 和 Executor 中都会对 SparkEnv 进行初始化，且执行代码相同，

Spark 存储体系架构图如7.1所示

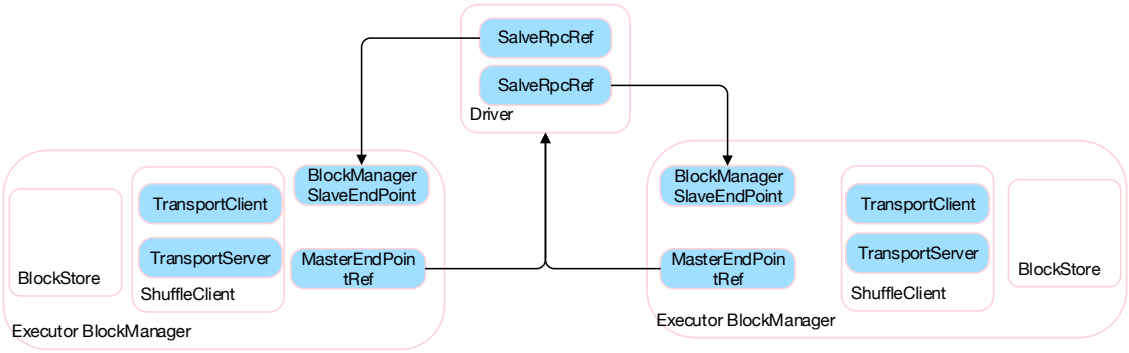


图 7.1　Spark 存储体系架构

Spark 定义了抽象类 BlockStore，用于指定所有的存储类型规范。Spark1.6 版本中定义了三种实现 MemoryStore、DiskStore 和 ExternalBlockStore，其类图如图7.2所示

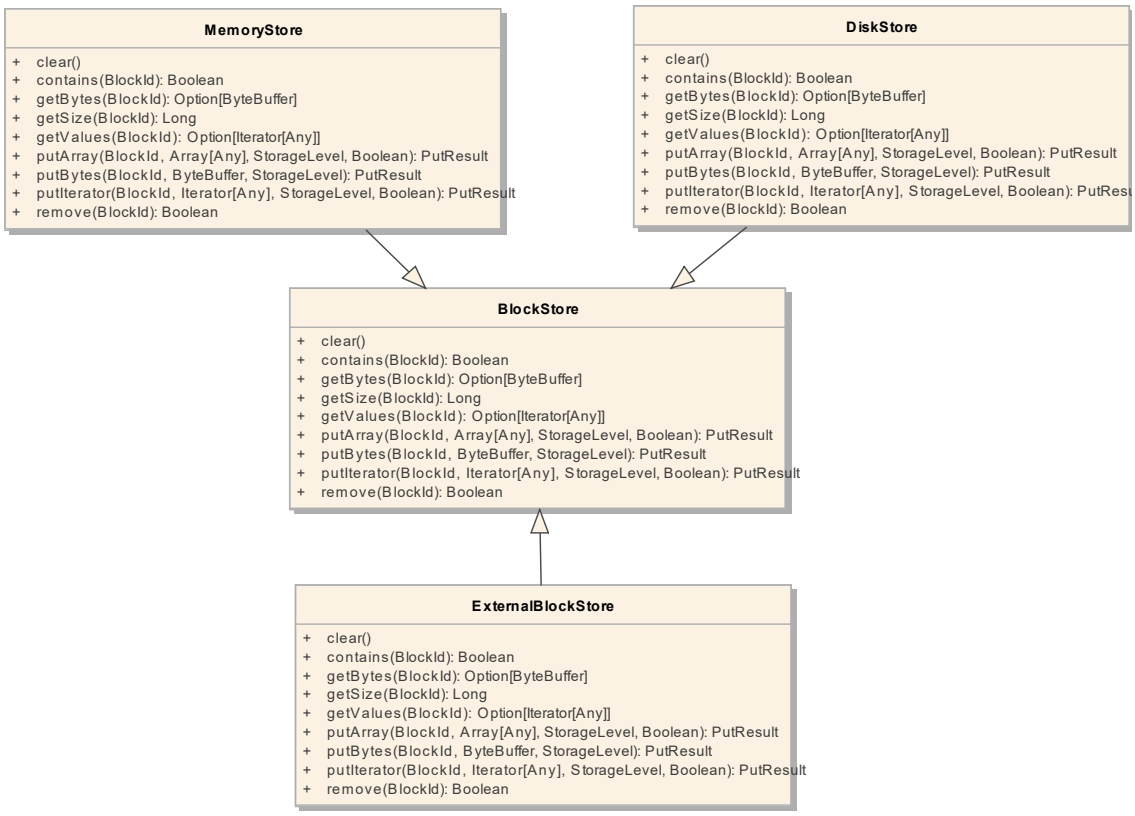


图 7.2　BlockStore 类图结构

7.2　ShuffleClient 模块

由于 Spark 是分布式部署的，每个 Task 最终都运行在不同的机器节点上，map 任务的输出结果直接存储在 map 任务所在的机器的存储体系中，reduce 任务极有可能不在同一机器上运行，所以需要远程下载 map 任务的中间结果，这时存储体系中的 ShuffleClient 就可以发挥作用了。

ShuffleClient 并不只是 shuffle 的客户端，它即包含了 client，可以将 shuffle 文件上传到其他 Executor 或者从其他 Executor 下载文件到本地，同时也为其他 Executor 提供 shuffle 服务。在 spark1.6 中 shuffleClient 在没有开启 externalShuffleService 时即为 NettyBlockTransferService。其初始化内容如程序7.2所示

</>　程序 **7.2:** *nettyBlockTransferService 初始化*　</>

```
override def init(blockDataManager: BlockDataManager): Unit = {
  //创建 rpc 服务
  val rpcHandler = new NettyBlockRpcServer(conf.getAppId, serializer,
    ↪  blockDataManager)
```

```
  var serverBootstrap: Option[TransportServerBootstrap] = None
  var clientBootstrap: Option[TransportClientBootstrap] = None
  if (authEnabled) {
    serverBootstrap = Some(new SaslServerBootstrap(transportConf,
      ↪  securityManager))
    clientBootstrap = Some(new SaslClientBootstrap(transportConf,
      ↪  conf.getAppId, securityManager,
    securityManager.isSaslEncryptionEnabled()))
  }
  //创建 TransportContext
  transportContext = new TransportContext(transportConf, rpcHandler)
  //创建 RPC 客户端工厂
  clientFactory =
    ↪  transportContext.createClientFactory(clientBootstrap.toSeq.asJava)
  //创建 Netty 服务器 TransportServer，端口由 spark.blockManager.port
    ↪  控制
  server = createServer(serverBootstrap.toList)
  appId = conf.getAppId
}
```

7.2.1　Block 的 RPC 服务

在 map 和 reduce 两端，当 reduce 任务需要从远处节点 Executor 上下载 map 中间输出时，NettyBlockRpcServer 需要提供下载 Block 文件的功能；同时，为了容错，也需要将 Block 的数据上传到其他节点，所以 NettyBlockRpcServer 还提供上传功能。NettyBlockRpcServer 核心功能代码如程序7.3

</>　程序 **7.3:** *NettyBlockRpcServer 核心实现*　</>

```
message match {
  case openBlocks: OpenBlocks =>
    val blocks:
      ↪  Seq[ManagedBuffer]=openBlocks.blockIds.map(BlockId.apply).map(blockManager.get
```

```
    val streamId = streamManager.registerStream(appId,
      ↪ blocks.iterator.asJava)
    logTrace(s"Registered streamId $streamId with ${blocks.size}
      ↪ buffers")
    responseContext.onSuccess(new StreamHandle(streamId,
      ↪ blocks.size).toByteBuffer)
  case uploadBlock: UploadBlock =>
    val level: StorageLevel
      ↪ =serializer.newInstance().deserialize(ByteBuffer.wrap(uploadBlock.metadata))
    val data = new
      ↪ NioManagedBuffer(ByteBuffer.wrap(uploadBlock.blockData))
    blockManager.putBlockData(BlockId(uploadBlock.blockId), data,
      ↪ level)
    responseContext.onSuccess(ByteBuffer.allocate(0))
```

7.2.2　传输上下文 TransportContext

TransportContext 用 Java 编写，TransportContext 既可以创建 Netty 服务，也可以创建 Netty 客户端。其构造器代码如程序7.4所示

</>　程序 **7.4:** *TransportContext* 构造器　</>

```
public TransportContext(
TransportConf conf,
RpcHandler rpcHandler,
boolean closeIdleConnections) {
  //主要控制 Netty 框架提供的 shuffle 的 I/O 交互的客户端和服务器线
    ↪ 程数量
  this.conf = conf;
  //负责 shuffle 的 I/O 服务端在接收到客户端 RPC 请求后，提供打开
    ↪ Block 或者上传 Block 的 RPC 处理，此处即为 NettyBlockRpcServer
  this.rpcHandler = rpcHandler;
  //在 shuffle 的 I/O 客户端对消息进行编码
  this.encoder = new MessageEncoder();
```

```
    //在 shuffle 的 I/O 服务端对客户端传来的 ByteBuf 进行解析，防止丢包和解析出错
    this.decoder = new MessageDecoder();
    this.closeIdleConnections = closeIdleConnections;
  }
```

7.2.3　RPC 客户端 TransportClientFactory

TransportClientFactory 是创建 Netty 客户端 TransporClient 的工厂类，TransportClient 用于向 Netty 服务端发送 RPC 请求。TransportClientFactory 主要部分如程序7.5所示

</>　程序 **7.5:** *TransportClientFactory* 构造器　</>

```
public TransportClientFactory(
TransportContext context,
List<TransportClientBootstrap> clientBootstraps) {
  this.context = Preconditions.checkNotNull(context);
  this.conf = context.getConf();
  //缓存客户端列表
  this.clientBootstraps =
    Lists.newArrayList(Preconditions.checkNotNull(clientBootstraps));
  //缓存客户端连接
  this.connectionPool = new ConcurrentHashMap<SocketAddress,
    ClientPool>();
  //节点之间去数据的连接数，可以使用属性
    spark.shuffle.io.numConnectionsPerPeer 来设置，默认为 1
  this.numConnectionsPerPeer = conf.numConnectionsPerPeer();
  this.rand = new Random();
  IOMode ioMode = IOMode.valueOf(conf.ioMode());
  //客户端 channel 被创建时使用的类，spark.shuffle.io.mode 来配置，默认为 NioSocketChannel
  this.socketChannelClass = NettyUtils.getClientChannelClass(ioMode);
  // TODO: Make thread pool name configurable.
```

```
17  //根据 Netty 规范，客户端只有 Work 组，所以此处创建 workerGroup，实
      ↪  际上是 NioEventLoopGroup
18  this.workerGroup = NettyUtils.createEventLoop(ioMode,
      ↪  conf.clientThreads(), "shuffle-client");
19  //汇集 ByteBuf 但对本地线程缓存禁用的分配器
20  this.pooledAllocator = NettyUtils.createPooledByteBufAllocator(
21  conf.preferDirectBufs(), false /* allowCache */,
      ↪  conf.clientThreads());
22 }
```

7.2.4　Netty 服务器 TransportServer

TransportServer 提供了 Netty 实现的服务器端，用于提供 RPC 服务，上传、下载等。

7.2.5　获取远程 shuffle 文件

在上一节中可以看到在获取远程 shuffle 文件时调用的是 shuffle-Client.fetchBlocks，实际上是利用 NettyBlockTransferService 中创建 Netty 服务。具体的函数实现如程序7.6所示

</>　程序 **7.6:** 获取远端节点上的 *shuffle* 文件　</>

```
1 override def fetchBlocks(
2 host: String,
3 port: Int,
4 execId: String,
5 blockIds: Array[String],
6 listener: BlockFetchingListener): Unit = {
7     val blockFetchStarter = new RetryingBlockFetcher.BlockFetchStarter
        ↪  {
8       override def createAndStart(blockIds: Array[String], listener:
          ↪  BlockFetchingListener) {
9         val client = clientFactory.createClient(host, port)
```

```
10          new OneForOneBlockFetcher(client, appId, execId,
              ↪  blockIds.toArray, listener).start()
11        }
12      }
13      val maxRetries = transportConf.maxIORetries()
14      if (maxRetries > 0) {
15        new RetryingBlockFetcher(transportConf, blockFetchStarter,
            ↪  blockIds, listener).start()
16      } else {
17        blockFetchStarter.createAndStart(blockIds, listener)
18      }
19  }
```

7.2.6　上传 shuffle 文件

NettyBlockTransferService 的上传功能也是利用了 NettyBlockTransferService 中创建的 Netty 服务。具体实现如程序7.7所示

</>　程序 **7.7:** *uploadBlock* 实现　</>

```
override def uploadBlock(
hostname: String,
port: Int,
execId: String,
blockId: BlockId,
blockData: ManagedBuffer,
level: StorageLevel): Future[Unit] = {
  val result = Promise[Unit]()
  //创建 Netty 服务客户端
  val client = clientFactory.createClient(hostname, port)
  //将 Block 的存储级别 StorageLevel 序列化
  val levelBytes = serializer.newInstance().serialize(level).array()
  //将 Block 中的 ByteBuffer 转化为数组，便于序列化
  val nioBuffer = blockData.nioByteBuffer()
```

```
  val array = if (nioBuffer.hasArray) {
    nioBuffer.array()
  } else {
    val data = new Array[Byte](nioBuffer.remaining())
    nioBuffer.get(data)
    data
  }
  //调用 Netty 客户端的 snedRpc 方法将字节数组上传，回调
    ↪ RpcResponseCallback
  client.sendRpc(new UploadBlock(appId, execId, blockId.toString,
    ↪ levelBytes, array).toByteBuffer,
    new RpcResponseCallback {
      override def onSuccess(response: ByteBuffer): Unit = {
        result.success((): Unit)
      }
      override def onFailure(e: Throwable): Unit = {
        result.failure(e)
      }
  })
  result.future
}
```

7.3　BlockManagerMaster 对 BlockManager 的管理

Driver 上的 BlockManagerMaster 对存在于 Executor 上的 BlockManager 统一管理，比如 Executor 注册 BlockManger、更新 Executor 上 Block 的最新信息。Spark 作为分布式处理框架，在管理上是 Driver 端持有 BlockManagerMasterEndpoint，所有的 Executor 上会有个 BlockManagerMasterEndpointRef，所有的 Executor 与 Driver 在 BlockManger 的交互都依赖于它。

7.3.1　BlockManagerMasterEndpoint

BlockManagerMasterEndpoint 只存在于 Driver 上，在 Executor 启动时会加载和 Driver 中一样的 SparkEnv，这里会完成对 Driver 中 BlockManagerMasterEndpoint 的

引用。然后通过其给 BlockManagerMasterEndpoint 发消息，实现与 Driver 的交互。BlockManagerMasterEndpoint 维护了很多缓存数据结构，重点看下面三个

1 blockManagerInfo

其类型为 HashMap[BlockManagerId, BlockManagerInfo]，缓存所有的 BlockManagerId 及其 BlockManager 的信息

2 blockManagerIdByExecutor

其类型为 HashMap[String, BlockManagerId]，缓存 executorId 与其拥有的 BlockManagerId 之间的映射关系

3 blockLocations

其类型为 JHashMap[BlockId, mutable.HashSet[BlockManagerId]]，缓存 Block 与 BlockManagerId 直接的映射关系

7.3.2　BlockManagerSlaveEndpoint

在每个 Executor 上都会运行一个 BlockManagerSlaveEndpoint，用来接收 Driver 中 BlockManagerMaster 的消息。在 Executor 对 BlockManager 初始化时会调用 BlockManagerMaster.registerBlockManager 来向 Dirver 中的 BlockManagerMaster 注册 BlockManger 的信息，如程序7.8所示

</>　程序 **7.8:** *registerBlockManager*　</>

```
def registerBlockManager(
blockManagerId: BlockManagerId, maxMemSize: Long, slaveEndpoint:
  ↪  RpcEndpointRef): Unit = {
 logInfo("Trying to register BlockManager")
 tell(RegisterBlockManager(blockManagerId, maxMemSize,
   ↪  slaveEndpoint))
 logInfo("Registered BlockManager")
}
```

由上面的代码可以看出，Executor 端会将 blockManagerId、最大内存和 BlockManagerSlaveEndpoint 封装为 RegisterBlockManager 消息，通过 Executor 端持有的 BlockManagerMasterEndpointRef 向 Driver 端运行的 BlockManagerMaster 注册。Driver 收到 Executor 注册的消息后会对传来的消息内容进行处理，具体的处理如程序7.9所示，注册的时候会对 blockManagerIdByExecutor 和 blockManagerInfo 两个 HashMap 进行数据更新，此时 Driver 中的 BlockManagerMaster 就拥有了 Executor

中 BlockManager 的信息，且持有了对应的 BlockManagerSlaveEndpointRef。

</>　程序 7.9: *Driver 注册 BlockManager 实现*　</>

```
private def register(id: BlockManagerId, maxMemSize: Long,
  ↪  slaveEndpoint: RpcEndpointRef) {
 val time = System.currentTimeMillis()
 //确保 blockManagerInfo 持有消息中的 BlockManagerId 及其对应信息
 if (!blockManagerInfo.contains(id)) {
   //确保一个 Executor 只有一个 BlockManagerId
   blockManagerIdByExecutor.get(id.executorId) match {
   //旧的 BlockManagerId 会被移除
     case Some(oldId) =>
       removeExecutor(id.executorId)
     case None =>
   }
 blockManagerIdByExecutor(id.executorId) = id
 blockManagerInfo(id) = new BlockManagerInfo(id,
   ↪  System.currentTimeMillis(), maxMemSize, slaveEndpoint)
 }
 //最后向 listenerBus 推送一个 SparkListenerBlockManagerAdded 消息
 listenerBus.post(SparkListenerBlockManagerAdded(time, id,
   ↪  maxMemSize))
}
```

7.4　磁盘管理器 DiskBlockManager

磁盘管理器的创建在新建 BlockManager 实例的时候，它将在 spark.local.dir 中配置的目录下创建二级目录，二级目录的个数由 spark.diskStore.subDirectories 指定，默认为 64. 这些目录里面将会保存 Block 数据，新建二级目录用于对文件进行散列存储，散列存储可以是所有的文件都随机存放，写入或删除文件都很方便，存取速度快，节省时间。另外一个重要的方法就是 getFile，程序要获取磁盘上的文件都是通过 getFile 进行的，其中 subDirs 是二维数组，用来缓存一级目录和二级目录。这里也使用了 Spark 磁盘散列文件存储的实现机制。如程序7.10所示

</>　程序 7.10: *getFile* 的实现　</>

```
def getFile(filename: String): File = {
  //根据文件名 blockId.name 计算哈希值
  val hash = Utils.nonNegativeHash(filename)
  //根据哈希值与本地文件一级目录的总数求余数，记为 dirId
  val dirId = hash % localDirs.length
  //根据哈希值与本地文件一级目录的总数求商，此商与二级目录的数据在
    ↪ 求余数，记为 subDirId
  val subDirId = (hash / localDirs.length) % subDirsPerLocalDir
  //如果 dirId/subDirId 目录存在，则获取 dirId/subDirId 目录下的文件
  val subDir = subDirs(dirId).synchronized {
    val old = subDirs(dirId)(subDirId)
    if (old != null) {
      old
    } else {
      //否则新建 dirId/subDirId 目录
      val newDir = new File(localDirs(dirId), "%02x".format(subDirId))
      if (!newDir.exists() && !newDir.mkdir()) {
        throw new IOException(s"Failed to create local dir in
          ↪ \$newDir.")
      }
      subDirs(dirId)(subDirId) = newDir
      newDir
    }
  }
  new File(subDir, filename)
}
```

7.5　内存存储 MemoryStore

MemoryStore 存储的是没有序列化的 Java 对象数组或者序列化的 ByteBuffer 存储到内存中，Spark 中 MemoryStore 的内存模型如图7.3所示

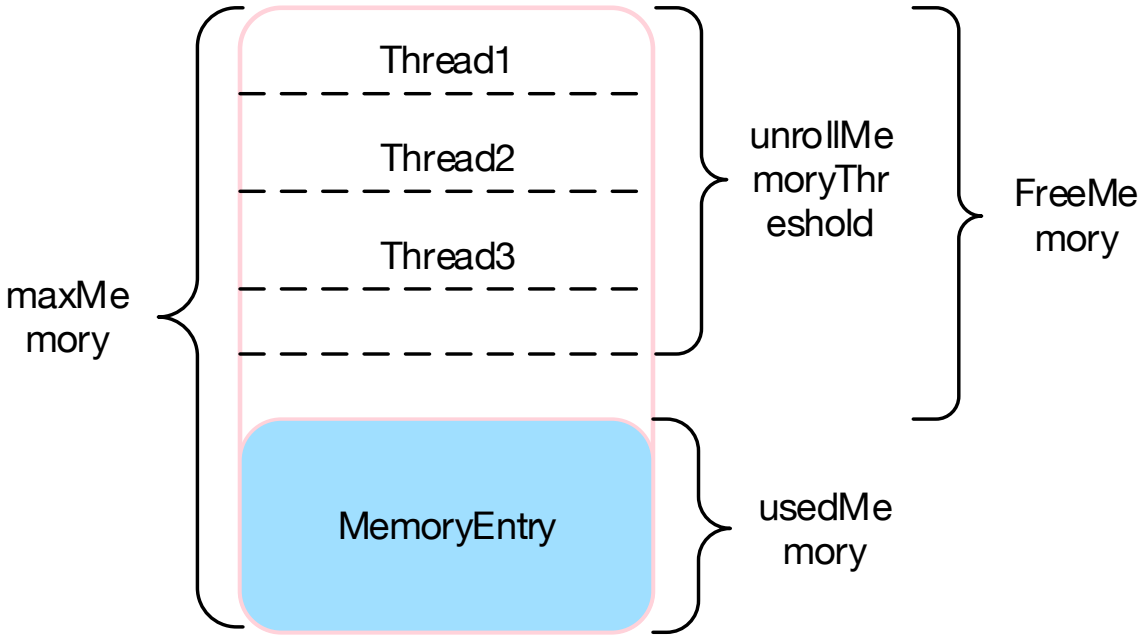


图 7.3　MemoryStore 的内存模型

如图7.3中看出，MemoryStore 的存储可以分为两部分

1 MemoryEntry 组成的 usedMemory，这些 Memory 实际是通过 entries 持有的

2 unrollMemoryMap 被各线程提前预定，就像占座一样

7.6　磁盘存储 DiskStore

当 MemoryStore 没有足够空间时，就会使用 DiskStore 将块存入磁盘。