



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

2017 年 3-4 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 04 月 17 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
1.1 WordCount 程序	1
1.2 提交 WordCount 程序	2
1.3 准备 WordCount 提交程序运行环境	3
1.4 进入 Spark-Submit 提交主类	5
1.5 Yarn 代理的启动	7
1.6 ApplicationMaster 的启动	11
1.7 Driver 的启动	12
第 2 章 SparkContext 的初始化	14
2.1 执行环境 SparkConf	14
2.2 SparkContext 综合概述	14
2.2.1 调用栈	15
2.2.2 监听主线 ListenerBus	15
2.2.3 元数据清理器	16
2.2.4 可视化监控服务 SparkUI	16
2.2.5 任务调度器	20
2.2.6 DAG 调度器	26
第 3 章 Spark RDD 创建及转换	29
3.1 Spark RDD 综述	29
3.2 Spark RDD 创建	30
3.3 Spark RDD 转换	30
3.4 DAG 的生成	30
3.4.1 RDD 中依赖	30
3.4.2 DAG 构建	34
3.4.3 WordCount 的 RDD 转换和 DAG 生成	34
第 4 章 Job 运行和调度器模块	37
4.1 DagScheduler 实现	38
4.1.1 Stage 的划分	38
4.1.2 实例:WordCount 划分 Stage	40
4.2 任务调度器实现	42

4.2.1 任务的生成	42
4.2.2 TaskScheduler 的创建	43
4.2.3 Task 的提交	43
4.2.4 Task 计算结果的处理	45
第 5 章 Executor 执行 Task	49
5.1 Executor 启动	49
5.2 Task 的执行	53
5.2.1 Task 执行前的准备	53
5.2.2 执行 Task	54
5.2.3 Task 结果的处理	58
5.3 Executor 生命周期	59
第 6 章 Spark Shuffle 分析	61
6.1 Shuffle Writer 历史版本	62
6.1.1 Hash Based Shuffle Write	62
6.1.2 Hash Based Shuffle Write 存在的问题	63
6.1.3 Shuffle Consolidate Writer	63
6.1.4 Sort Based Write	64
6.2 Spark1.6 版本 Shuffle Write	65
6.2.1 BaseShuffleHandle 对应 SortShuffleWriter	65
6.2.2 实例化 ExternalSorter	67
6.2.3 shuffle 写操作	67
6.2.4 shuffle 结果本地缓存	71
6.2.5 BypassMergeSortShuffleHandle 对应 BypassMergeSortShuffleWriter	75
6.2.6 SerializedShuffleHandle 对应 UnsafeShuffleWriter	75
6.2.7 Shuffle Write 流程图	76
6.3 Spark1.6 版本 Shuffle Read	77
6.3.1 shuffleClient	79
6.3.2 元数据获取	80
6.3.3 根据元数据信息抓取数据	82
6.3.4 reduce 端读取中间计算结果	83
6.3.5 本地数据的获取	85
6.3.6 远程数据的获取	86
6.3.7 BlockStoreShuffleReader.read 方法	88

6.3.8 Shuffle Read 流程图.....	88
第 7 章 存储体系	90
7.1 概述	90
7.2 ShuffleClient 模块	92
7.2.1 Block 的 RPC 服务	93
7.2.2 传输上下文 TransportContext.....	94
7.2.3 RPC 客户端 TransportClientFactory	95
7.2.4 Netty 服务器 TransportServer	96
7.2.5 获取远程 shuffle 文件	96
7.2.6 上传 shuffle 文件	97
7.3 BlockManagerMaster 对 BlockManager 的管理	98
7.3.1 BlockManagerMasterEndpoint.....	98
7.3.2 BlockManagerSlaveEndpoint	99
7.4 磁盘管理器 DiskBlockManager	100
7.5 内存存储 MemoryStore	101
7.6 磁盘存储 DiskStore	102

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)) .
16      ⏵ reduceByKey(_+_)
17    println(filtered.collect().mkString(","))
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：

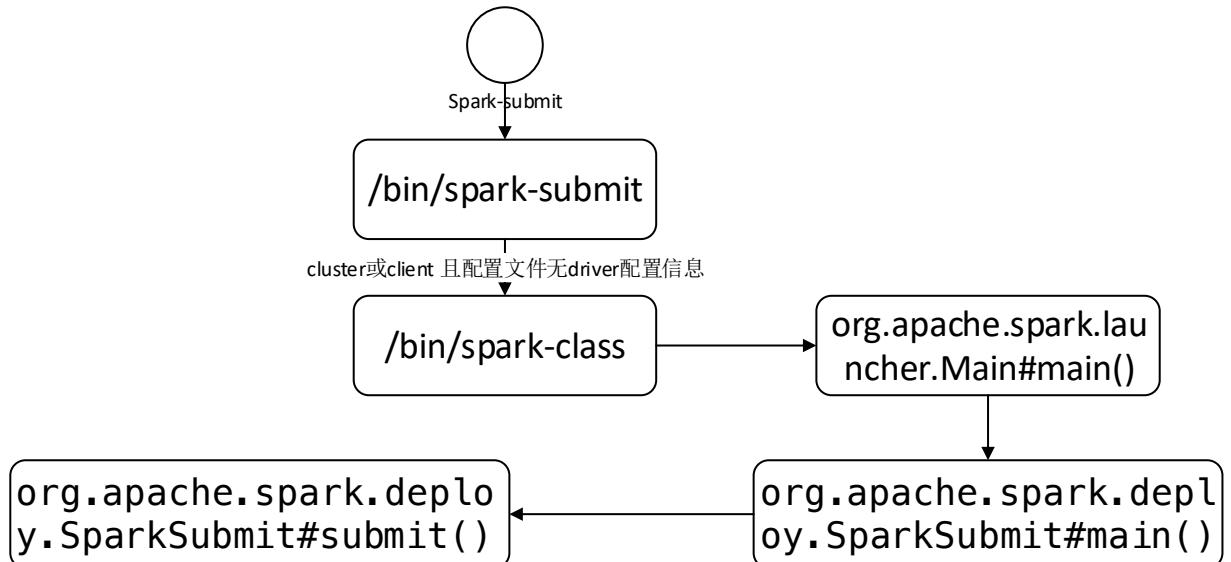


图 1.1 spark-submit 运行流程

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.SparkSubmit
- 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
16        // ↪ an argument.
17        List<String> bashCmd = prepareBashCommand(cmd, env);
18        for (String c : bashCmd) {
19            System.out.print(c);
20            System.out.print('\0');
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	

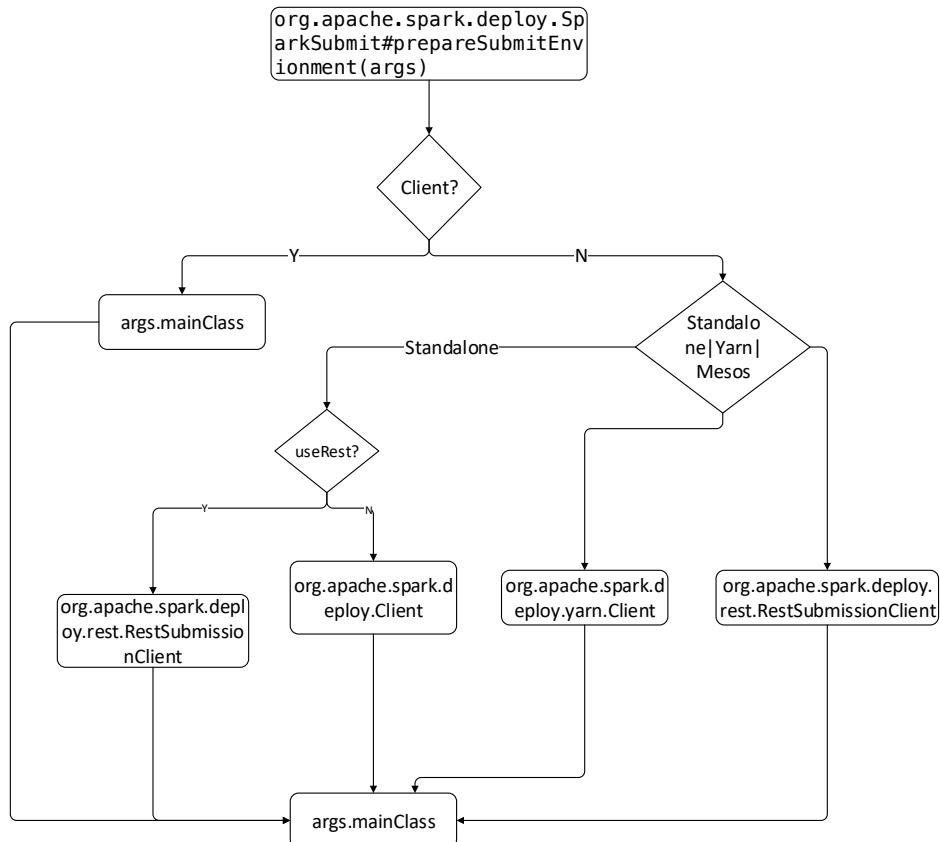


图 1.2 prepareSubmit 运行流程

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

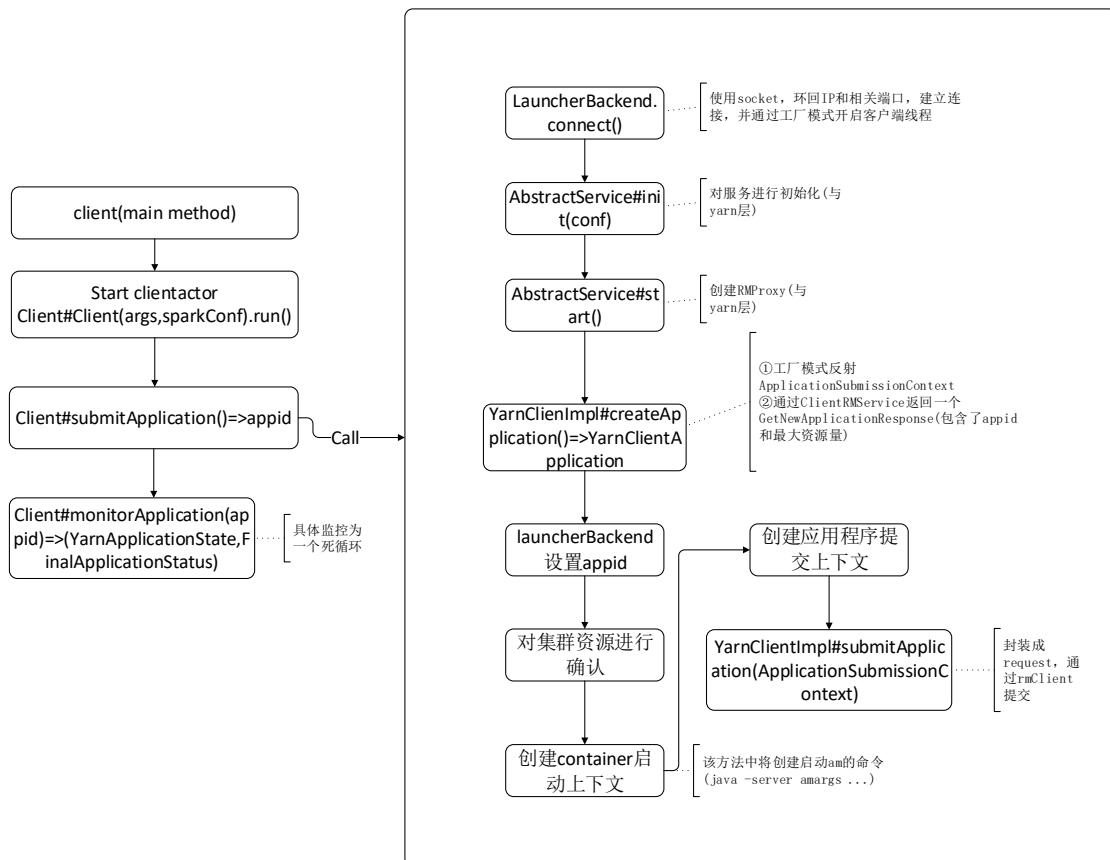


图 1.3 YarnClientMain 流程

main 函数代码如程序1.5所示

```

1 /**
2 * --name, com.wtx.HelloWorld,
3 * --jar, file:/root/HelloWorld-1.0-SNAPSHOT-jar-with-dependencies.jar,
4 * --class, com.wtx.HelloWorld,
5 * --arg, /user/root/inputfile.txt,
6 * --arg, 2
7 */
8 def main(argStrings: Array[String]) {
9   if (!sys.props.contains("SPARK_SUBMIT")) {
10     logWarning("WARNING: This client is deprecated and will be removed in a
11     ↪ " +
12     "future version of Spark. Use ./bin/spark-submit with \"--master yarn
13     ↪ \")")
14   }
15   System.setProperty("SPARK_YARN_MODE", "true")
16   val sparkConf = new SparkConf
17   val args = new ClientArguments(argStrings, sparkConf)
18   if (!Utils.isDynamicAllocationEnabled(sparkConf)) {
19     sparkConf.setIfMissing("spark.executor.instances", args.numExecutors.
20     ↪ toString)
21   }
22   new Client(args, sparkConf).run()
23 }
```

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，通过一标识符识别，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

```

1 def run(): Unit = {
2   this.appId = submitApplication()
3
4   if (!launcherBackend.isConnected() && fireAndForget) {
5     val report = getApplicationReport(appId)
6     val state = report.getYarnApplicationState
7     logInfo(s"Application report for $appId (state: $state)")
8     logInfo(formatReportDetails(report))
9
10    if (state == YarnApplicationState.FAILED || state ==
11        YarnApplicationState.KILLED) {
12      throw new SparkException(s"Application $appId finished with status:
13          $state")
14    }
15  } else {
16    val (yarnApplicationState, finalApplicationStatus) =
17      monitorApplication(appId)
18
19    if (yarnApplicationState == YarnApplicationState.FAILED ||
20        finalApplicationStatus == FinalApplicationStatus.FAILED) {
21      throw new SparkException(s"Application $appId finished with
22          failed status")
23    }
24    if (yarnApplicationState == YarnApplicationState.KILLED ||
25        finalApplicationStatus == FinalApplicationStatus.KILLED) {
26      throw new SparkException(s"Application $appId is killed")
27    }
28    if (finalApplicationStatus == FinalApplicationStatus.UNDEFINED) {
29      throw new SparkException(s"The final status of application
30          $appId is undefined")
31    }
32  }
33}
34
35}
```

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

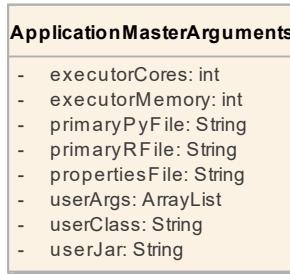


图 1.5 ApplicationMasterArguments 类图

- 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

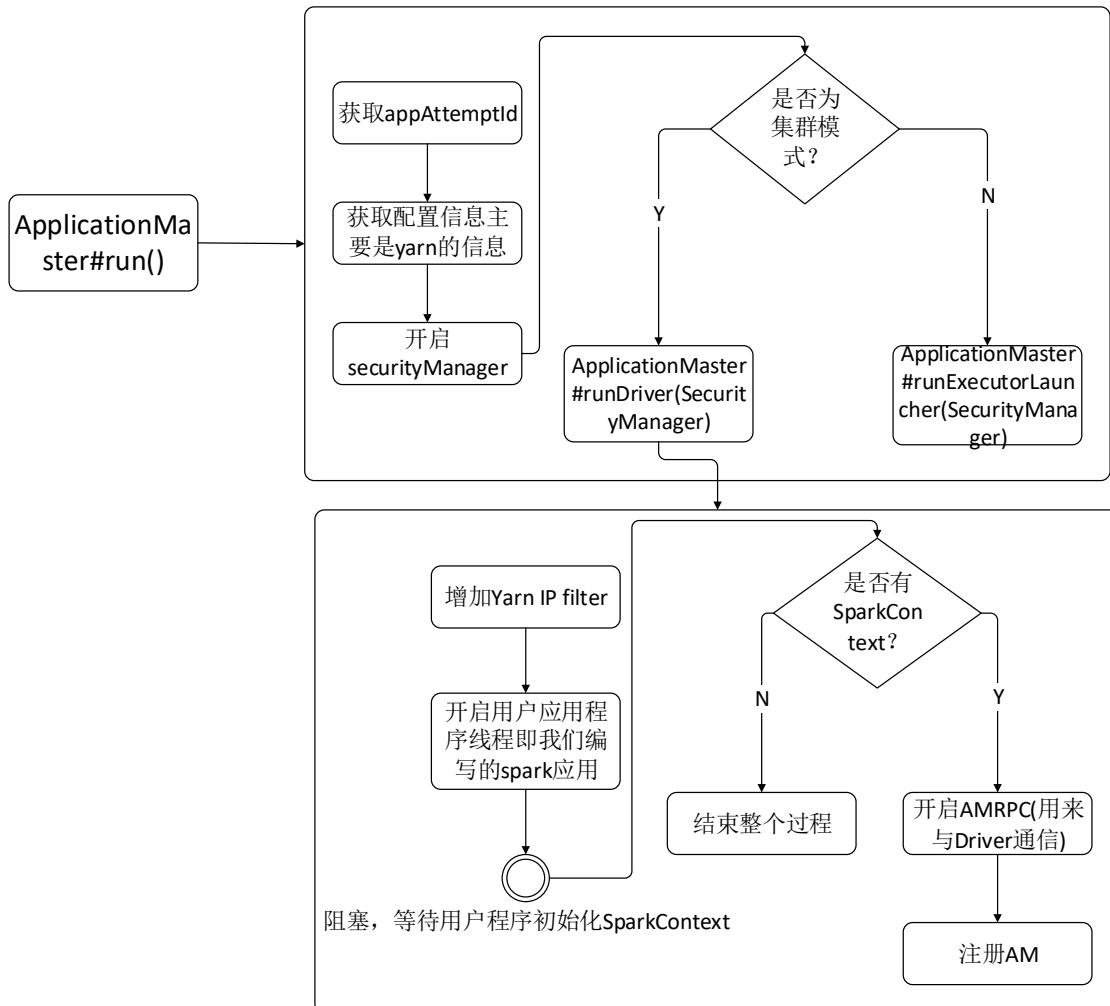


图 1.6 Driver 启动过程

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 }

```

```

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(""))
16     ↪ , securityMgr)
17     userClassThread.join()
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 WordCount 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

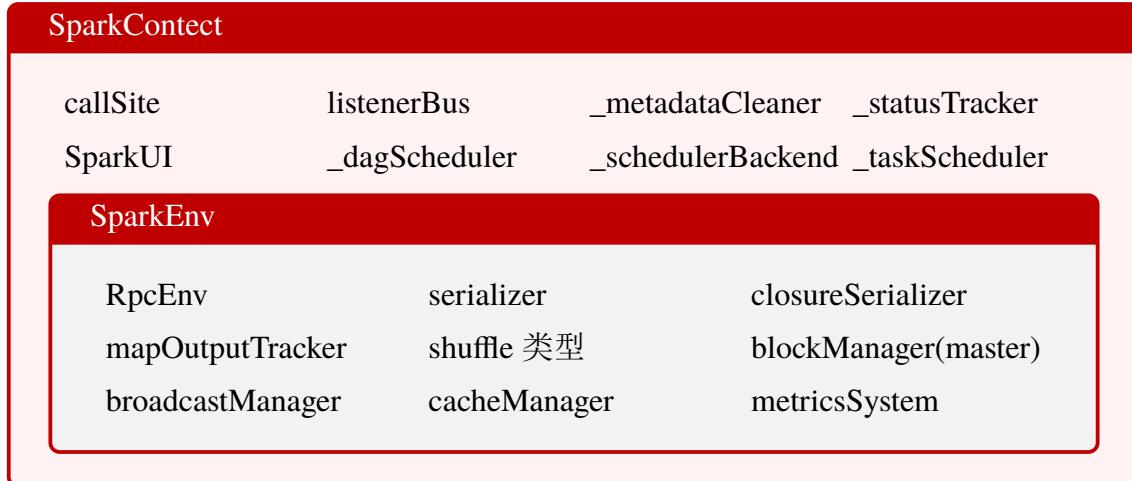
注册时使用 Kryo 序列化的类，指定该类的序列化器为 Kryo，关于 Kryo 的使用是 spark 性能优化的一个重要的组成部分。其中一个简单应用是图计算。

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 </>

1 private val listenerThread = new Thread(name) {
2   setDaemon(true)
3   override def run(): Unit = Utils.tryOrStopSparkContext(sparkContext)
4   →  {
5     AsynchronousListenerBus.withinListenerThread.withValue(true) {
6       while (true) {
7         eventLock.acquire()
8         self.synchronized {
9           processingEvent = true
10          }
11          try {
12            val event = eventQueue.poll
13            if (event == null) {
14              if (!stopped.get) {
15                .....
16              }
17            }
18            postToAll(event)
19          } finally {
```

```

20         self.synchronized {
21             processingEvent = false
22         }
23     }
24 }
25 }
26 }
27 }

```

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 </>

```

1 _ui =
2 if (conf.getBoolean("spark.ui.enabled", true)) {
3   Some(SparkUI.createLiveUI(this, _conf, listenerBus,
4     ← _jobProgressListener,
5     _env.securityManager, appName, startTime = startTime))
6 } else {
7   // For tests, do not enable the UI
8   None
9 }
9 _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           </>
 1 private def create(
 2   sc: Option[SparkContext],
 3   conf: SparkConf,
 4   listenerBus: SparkListenerBus,
 5   securityManager: SecurityManager,
 6   appName: String,
 7   basePath: String = "",
 8   jobProgressListener: Option[JobProgressListener] = None,
 9   startTime: Long): SparkUI = {
10   val _jobProgressListener: JobProgressListener =
11     jobProgressListener.getorElse {
12       val listener = new JobProgressListener(conf)
13       listenerBus.addListener(listener) listener
14     }
15   val environmentListener = new EnvironmentListener
16   val storageStatusListener = new StorageStatusListener
17   val executorsListener = new ExecutorsListener(storageStatusListener)
18   val storageListener = new StorageListener(storageStatusListener)
19   val operationGraphListener = new RDDOperationGraphListener(conf)
20   listenerBus.addListener(environmentListener)
21   listenerBus.addListener(storageStatusListener)
22   listenerBus.addListener(executorsListener)
23   listenerBus.addListener(storageListener)
24   listenerBus.addListener(operationGraphListener)
25   new SparkUI(sc, conf, securityManager, environmentListener,
26     storageStatusListener,
27     executorsListener, _jobProgressListener, storageListener,
28     operationGraphListener,
29     appName, basePath, startTime)
30 }

```

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 类实现代码如程序2.4所示

```
</>                                         程序 2.4: SparkUI JobsTab                                         </>
1 private[ui] class JobsTab(parent: SparkUI) extends SparkUITab(parent,
2   ← "jobs") {
3   val sc = parent.sc
4   val killEnabled = parent.killEnabled
5   val jobProgresslistener = parent.jobProgressListener
6   val executorListener = parent.executorsListener
7   val operationGraphListener = parent.operationGraphListener
8   def isFairScheduler: Boolean =
9     jobProgresslistener.schedulingMode.exists(_ ==
10       ← SchedulingMode.FAIR)
11     attachPage(new AllJobsPage(this))
12     attachPage(new JobPage(this))
13 }
```

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                                         </>
1 def render(request: HttpServletRequest): Seq[Node] = {
2   progressListener.synchronized {
3     .....
4     val stageHeader = s"Details for Stage $stageId (Attempt
5       ← $stageAttemptId)"
6     if (stageDataOption.isEmpty) {
```

```
6     val content =
7         <div id="no-info">
8             <p>No information to display for Stage {stageId} (Attempt
9                 ↳ {stageAttemptId})</p>
10            </div>
11
12            return UIUtils.headerSparkPage(stageHeader, content, parent)
13        }
14
15        if (stageDataOption.get.taskData.isEmpty) {
16
17            val content =
18                <div>
19                    <h4>Summary Metrics</h4> No tasks have started yet
20                    <h4>Tasks</h4> No tasks have started yet
21                </div>
22
23            return UIUtils.headerSparkPage(stageHeader, content, parent)
24        }
25
26        val stageData = stageDataOption.get
27
28        val tasks =
29            ↳ stageData.taskData.values.toSeq.sortBy(_.taskInfo.launchTime)
30
31        val numCompleted = tasks.count(_.taskInfo.finished)
32
33        val allAccumulables = progressListener.stageIdToData((stageId,
34            ↳ stageAttemptId)).accumulables
35
36        val externalAccumulables = allAccumulables.values.filter { acc =>
37            ↳ !acc.internal }
38
39        val hasAccumulators = externalAccumulables.size > 0
40
41        val summary =
42
43            <div>
44                <ul class="unstyled">
45                    <li>
46                        <strong>Total Time Across All Tasks: </strong>
47                        {UIUtils.formatDuration(stageData.executorRunTime)}
48                    </li>
49                    <li>
50                        <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

```

</>                               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
4     ← once!".format(className))
5   try {
6     var host =
7       ← Option(conf.getenv("SPARK_LOCAL_IP")).getOrElse("0.0.0.0")
8     serverInfo = Some(startJettyServer(host, port, handlers, conf,
9       ← name))
10    logInfo("Bound %s to %s, and started at
11      ← http://%s:%d".format(className, host,
12        publicHostName, boundPort))
13  } catch {
14    case e: Exception =>
15      logError("Failed to bind %s".format(className), e)
16      System.exit(1)
17  }
18}

```

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 </>  
  
1 case "yarn-standalone" | "yarn-cluster" =>  
2   if (master == "yarn-standalone") {  
3     logWarning(  
4       "\"yarn-standalone\" is deprecated as of Spark 1.0. Use  
5       \"yarn-cluster\" instead."  
6     )  
7     val scheduler = try {  
8       val clazz =  
9         Utils.classForName("org.apache.spark.scheduler.cluster.  
10      YarnClusterScheduler")  
11      val cons = clazz.getConstructor(classOf[SparkContext])  
12      cons.newInstance(sc).asInstanceOf[TaskSchedulerImpl]  
13    } catch {  
14      case e: Exception =>  
15        throw new SparkException("YARN mode not available?", e)  
16    }  
17  val backend = try {  
18    val clazz = Utils.classForName("org.apache.spark.scheduler.cluster.  
19      YarnClusterSchedulerBackend")  
20    val cons = clazz.getConstructor(classOf[TaskSchedulerImpl],  
21      classOf[SparkContext])  
22    cons.newInstance(scheduler,  
23      sc.asInstanceOf[CoarseGrainedSchedulerBackend])  
24  } catch {  
25    ....  
26  }  
27  scheduler.initialize(backend)  
28  (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

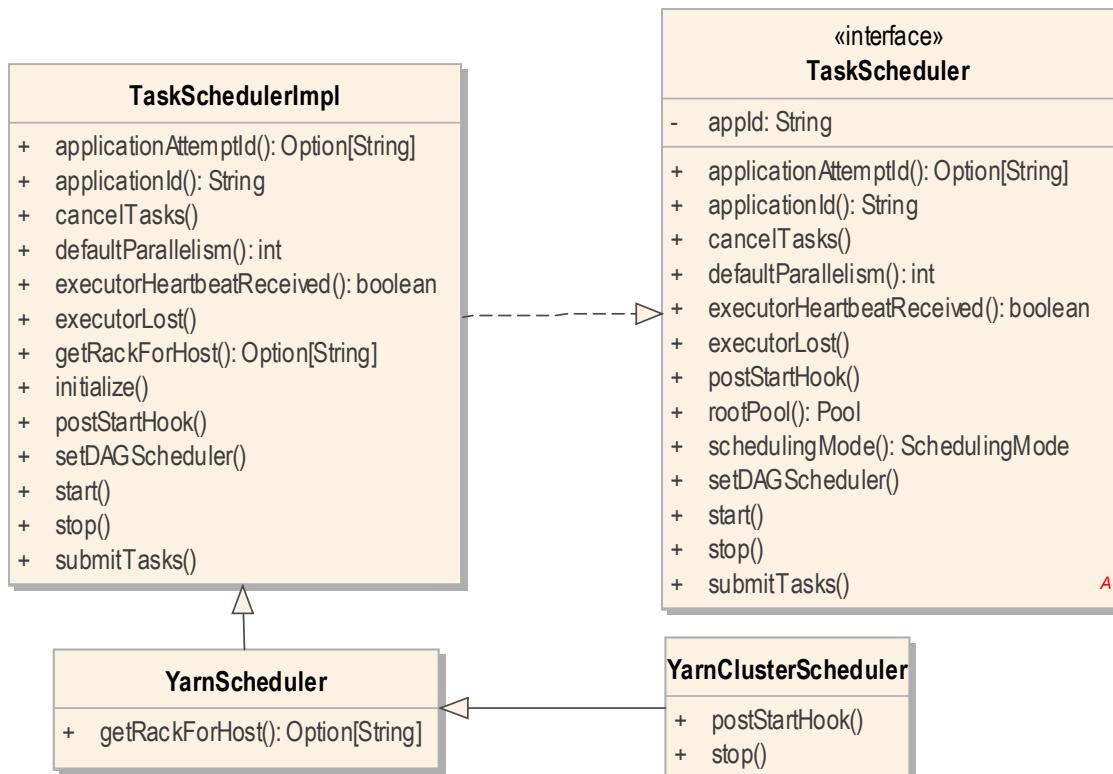


图 2.1 YarnScheduler 类图

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

</>

```

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

```

4   schedulableBuilder = {
5     schedulingMode match {
6       case SchedulingMode.FIFO =>
7         new FIFOschedulableBuilder(rootPool)
8       case SchedulingMode.FAIR =>
9         new FairSchedulableBuilder(rootPool, conf)
10      }
11    }
12   schedulableBuilder.buildPools()
13 }

```

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

```

</>                                         程序 2.9: 开启任务调度器                                         </>

1 override def start() {
2   //Yarn-cluster 模式下为 YarnClusterSchedulerBackend
3   backend.start()
4   if (!isLocal && conf.getBoolean("spark.speculation", false))
5     {
6       speculationScheduler.scheduleAtFixedRate(new Runnable {
7         override def run(): Unit =
8           Utils.tryOrStopSparkContext(sc) {
9             checkSpeculatableTasks()
10            }
11          }, SPECULATION_INTERVAL_MS, SPECULATION_INTERVAL_MS,
12            TimeUnit.MILLISECONDS)
13        }
14      }

```

2 YarnClusterSchedulerBackend 相关

与其相关的类及接口如图2.2所示

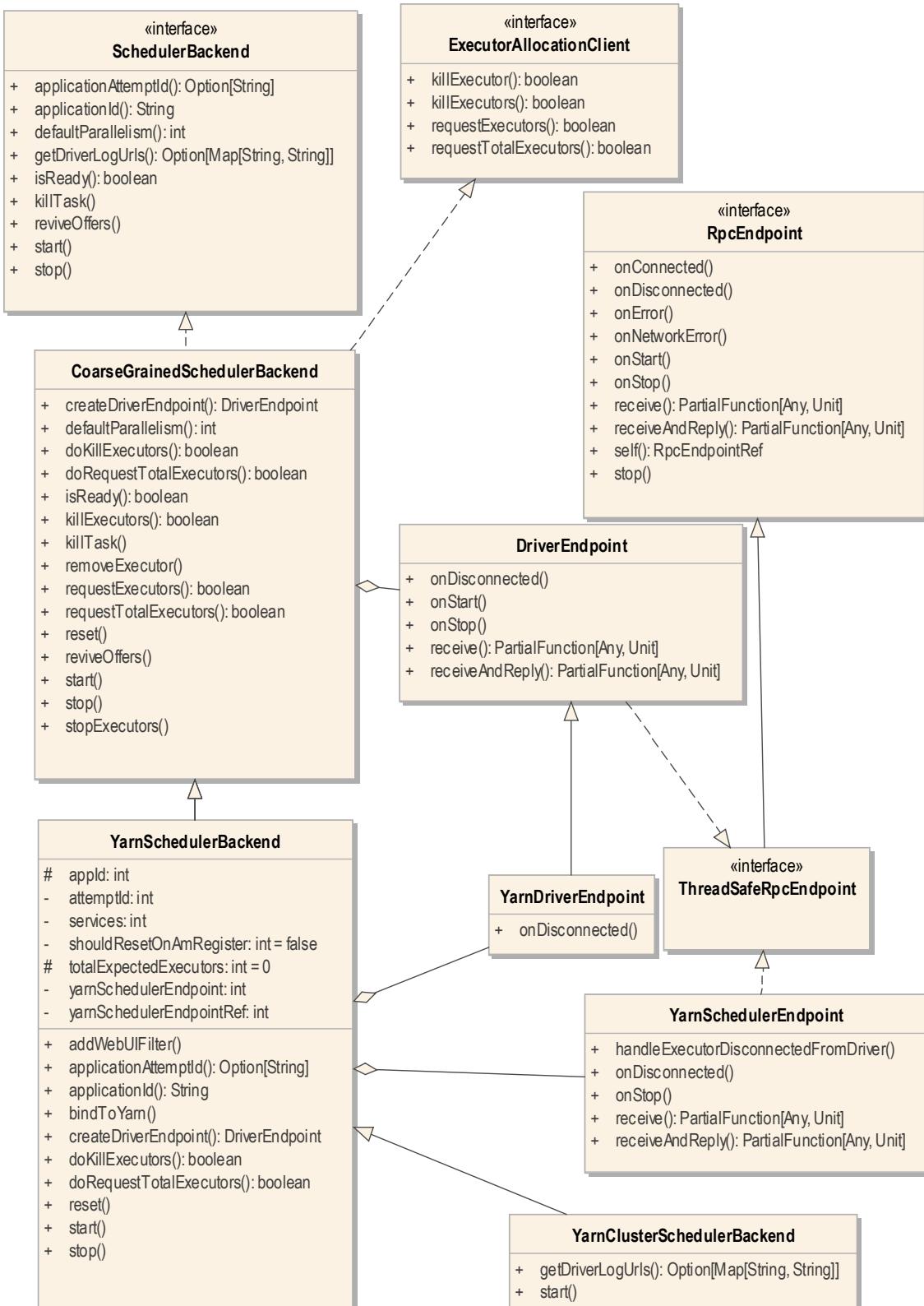


图 2.2 YarnSchedulerBackend 类图

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          </>
1  override def start() {
2    val properties = new ArrayBuffer[(String, String)]
3    for ((key, value) <- scheduler.sc.conf.getAll) {
4      if (key.startsWith("spark.")) {
5        properties += ((key, value))
6      }
7    }
8    // TODO (prashant) send conf instead of properties
9    driverEndpoint = rpcEnv.setupEndpoint(ENDPOINT_NAME,
10      → createDriverEndpoint(properties))
11 }
```

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。

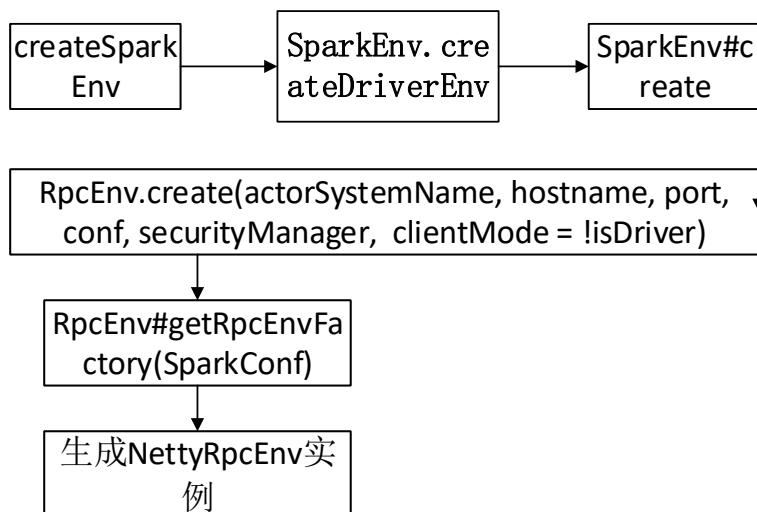


图 2.3 RPC 建立过程

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

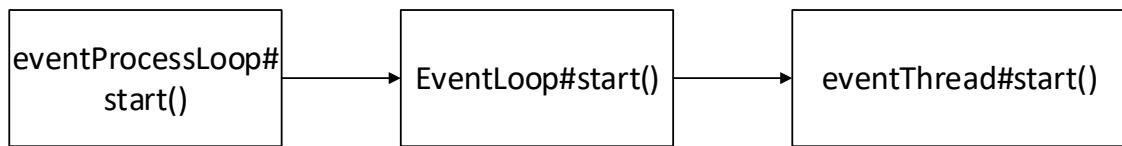


图 2.4 事件监听开启流程

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

```

</>
1 private val eventThread = new Thread(name) {
2   setDaemon(true)
3   override def run(): Unit = {
4     try {
5       while (!stopped.get) {
6         val event = eventQueue.take()
7         try {
8           onReceive(event)
9         } catch {
10           case NonFatal(e) => {
11             .....
12           }
13         }
14       }
15     } catch {
16       .....
17     }
18   }
19 }

```

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.onReceive </>

```
1 override def onReceive(event: DAGSchedulerEvent): Unit = {  
2   val timerContext = timer.time()  
3   try {  
4     doOnReceive(event)  
5   } finally {  
6     timerContext.stop()  
7   }  
8 }
```

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

$$\text{partitions} = \max\{\text{blockSize}, \text{defaultMinPartition}\} \quad (3-1)$$

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

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           </>
1 abstract class RDD[T: ClassTag]{
2   @transient private var _sc: SparkContext,
3   @transient private var deps: Seq[Dependency[_]]
4   ) extends Serializable with Logging {
5     .....
6 }
```

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

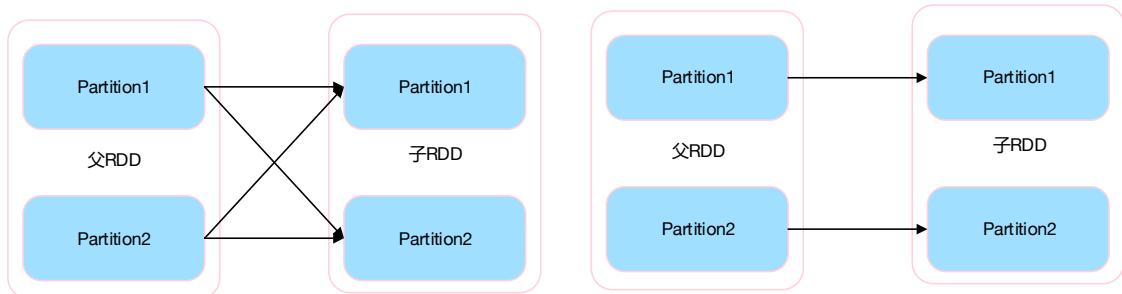


图 3.1 宽依赖（左）窄依赖（右）

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

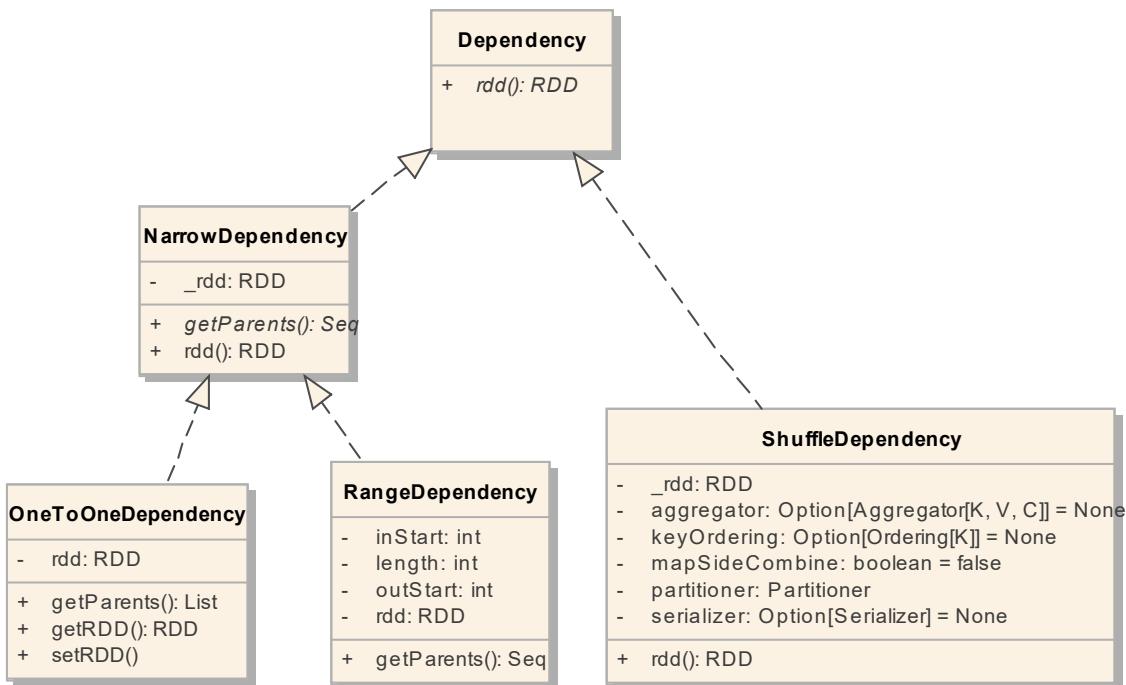


图 3.2 Spark RDD 依赖类层次图

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 </>
1 abstract class Dependency[T] extends Serializable {
2   def rdd: RDD[T]
3 }
  
```

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 </>
1 abstract class NarrowDependency[T]( _rdd: RDD[T]) extends Dependency[T]
2   {
3     def getParents(partitionId: Int): Seq[Int]
4     override def rdd: RDD[T] = _rdd
5   }
  
```

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

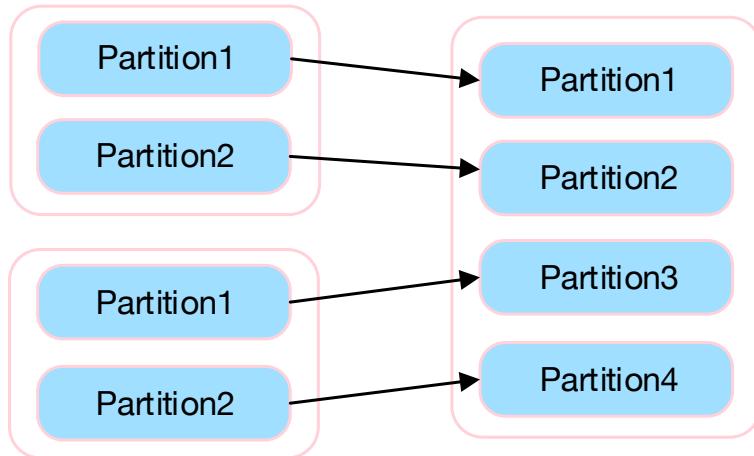


图 3.3 RangeDependency 示例

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

	程序 3.4: <i>ShuffleDependency</i>	
--	----------------------------------	--

```

1 class ShuffleDependency[K: ClassTag, V: ClassTag, C: ClassTag]{
2     @transient private val _rdd: RDD[_ <: Product2[K, V]],
3     val partitioner: Partitioner,
4     val serializer: Option[Serializer] = None,
5     val keyOrdering: Option[Ordering[K]] = None,
6     val aggregator: Option[Aggregator[K, V, C]] = None,
7     val mapSideCombine: Boolean = false)
8     extends Dependency[Product2[K, V]] {
9         override def rdd: RDD[Product2[K, V]] =
10             _rdd.asInstanceOf[RDD[Product2[K, V]]]
11         private[spark] val keyClassName: String =
12             reflect.classTag[K].runtimeClass.getName
13         private[spark] val valueClassName: String =
14             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

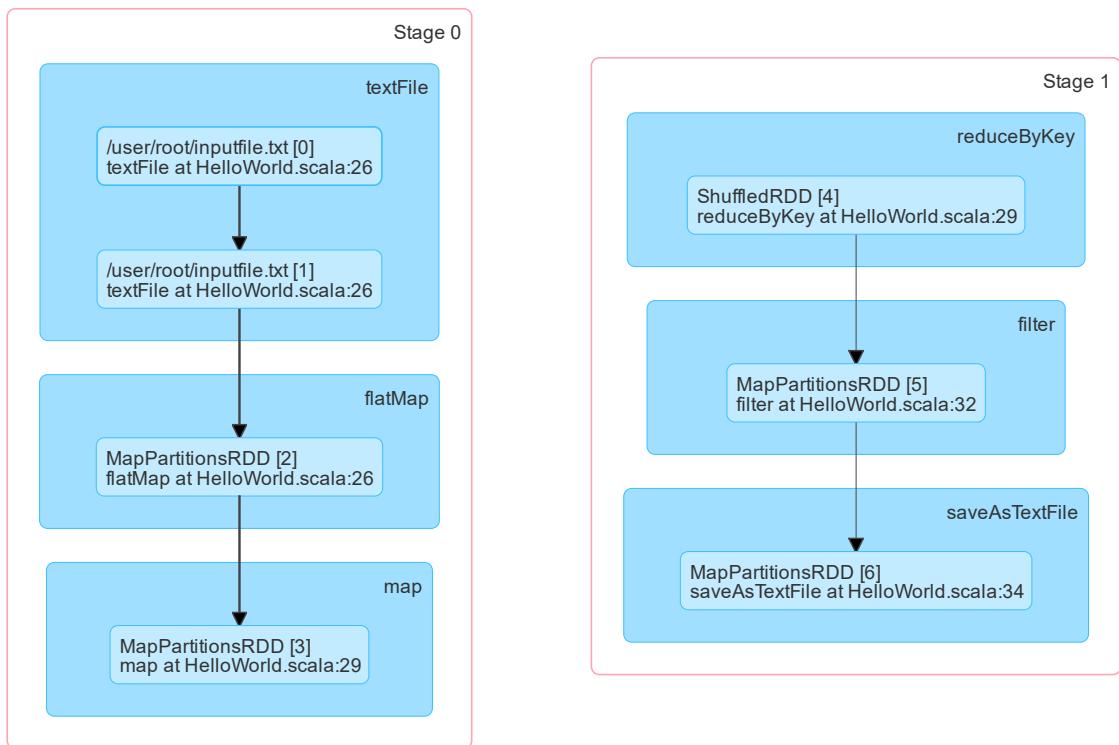


图 3.4 WordCount 的 RDD 转换

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 `shuffledRDD`，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。

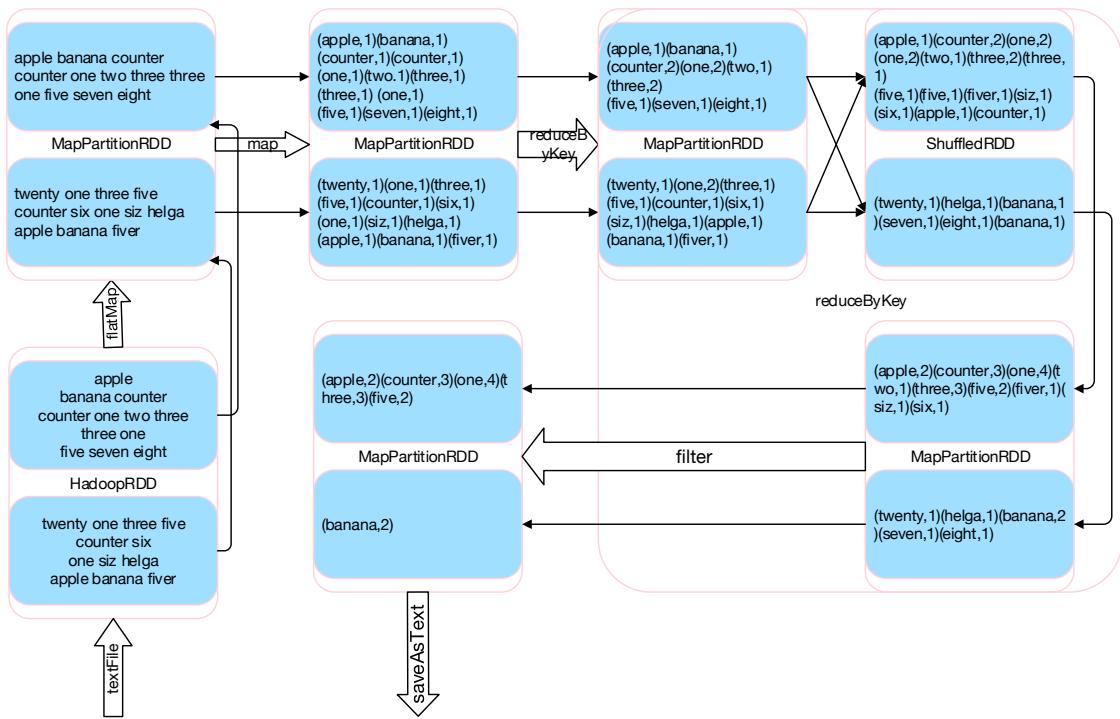


图 3.5 WordCount 的数据逻辑图

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                                         </>
1 def runJob[T, U](rdd: RDD[T], func: (TaskContext, Iterator[T]) =>
2   U, partitions: Seq[Int], callSite: CallSite, resultHandler: (Int, U)
3   => Unit, properties: Properties): Unit = {
4   .....
5   val waiter = submitJob(rdd, func, partitions, callSite,
6     resultHandler, properties)
7   waiter.awaitResult() match {
8     case JobSucceeded =>
9       (waiter.jobId, callSite.shortForm, (System.nanoTime - start) /
10      1e9))
11     case JobFailed(exception: Exception) =>
12       (waiter.jobId, callSite.shortForm, (System.nanoTime - start) /
13      1e9))
14       val callerStackTrace = Thread.currentThread().getStackTrace.tail
15       exception.setStackTrace(exception.getStackTrace ++
16         callerStackTrace)
17       throw exception
18   }
19 }
```

DagScheduler.runJob 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           </>
    1 def submitJob[T, U](
    2   rdd: RDD[T],
    3   func: (TaskContext, Iterator[T]) => U,
    4   partitions: Seq[Int],
    5   callSite: CallSite,
    6   resultHandler: (Int, U) => Unit,
    7   properties: Properties): JobWaiter[U] = {
    8     .....
    9     val jobId = nextJobId.getAndIncrement()
   10    .....
   11    val waiter = new JobWaiter(this, jobId, partitions.size,
   12      resultHandler)
   13    eventProcessLoop.post(JobSubmitted(
   14      jobId, rdd, func2, partitions.toArray, callSite, waiter,
   15      SerializationUtils.clone(properties)))
   16    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 Shuffle 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,
2   Iterator[_]) => _, partitions: Array[Int],
3   jobId: Int, callSite: CallSite): ResultStage = {
4   val (parentStages: List[Stage], id: Int) = getParentStagesAndId(rdd,
5     jobId)
6   val stage = new ResultStage(id, rdd, func, partitions, parentStages,
7     jobId, callSite)
8   stageIdToStage(id) = stage
9   updateJobIdStageIdMaps(jobId, stage)
10  stage
11 }

```

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[_], first jobId: Int): List[Stage]
2   = {
3     val parents = new HashSet[Stage]
4     val visited = new HashSet[RDD[_]]
5     val waitingForVisit = new Stack[RDD[_]]
6     def visit(r: RDD[_]) {
7       if (!visited(r)) {
8         visited += r
9         for (dep <- r.dependencies) {
10           if (!parents.contains(dep))
11             parents += dep
12           visit(dep)
13         }
14       }
15     }
16     visit(rdd)
17     parents.toList
18   }

```

```

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: `RDDdeps: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(onetoOneDependency(HadoopRDD[1]),List(1,2)))`

`map:`

`MapPartitionRDD[3](List(onetoOneDependency(MapPartitionRDD[2])),List(1,2))`

`reduceByKey:`

`ShuffledRDD[4](List(shuffleDependency(MapPartitionRDD[3])),List(3,4))`

`filter:`

`MapPartitionRDD[5](List(onetoOneDependency(ShuffledRDD[4])),List(3,4))`

`saveTextFile:`

`MapPartitionRDD[6](List(onetoOneDependency(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) &&
5         !failedStages(stage)) {
6       val missing = getMissingParentStages(stage).sortBy(_.id)
7       if (missing.isEmpty) {
8         submitMissingTasks(stage, jobId.get)
9       } else {
10         for (parent <- missing) {
11           submitStage(parent)
12         }
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 TaskSchedulerImpl.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           </>

```

```

1 //为了避免 Task 集中分配到某些机器上，随机打乱
2 val shuffledOffers = Random.shuffle(offers)
3 //存储分配好的 Task
4 val tasks = shuffledOffers.map(o => new
5   ← ArrayBuffer[TaskDescription](o.cores))
5 val availableCpus = shuffledOffers.map(o => o.cores).toArray
6 val sortedTaskSets = rootPool.getSortedTaskSetQueue
7 for (taskSet <- sortedTaskSets) {
8   taskSet.parent.name, taskSet.name, taskSet.runningTasks))
9   if (newExecAvail) {
10     taskSet.executorAdded()
11   }
12 }
13 var launchedTask = false
14 for (taskSet <- sortedTaskSets; maxLocality <-
15   ← taskSet.myLocalityLevels) {
16   do {
17     launchedTask = resourceOfferSingleTaskSet(taskSet, maxLocality,
18       ← shuffledOffers, availableCpus, tasks)
19   } while (launchedTask)
20 }
21 if (tasks.size > 0) {
22   hasLaunchedTask = true
23 }
24 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

<i></></i>	程序 4.7:	<i></></i>
<i>CoarseGrainedSchedulerBackend.DriverEndPoint.launchTasks</i>		

```

1 private def launchTasks(tasks: Seq[Seq[TaskDescription]]) {
2   for (task <- tasks.flatten) {
3     val serializedTask = ser.serialize(task)
4     if (serializedTask.limit >= akkaFrameSize -
5       → AkkaUtils.reservedSizeBytes) {
6       scheduler.taskIdToTaskSetManager.get(task.taskId).foreach {
7         → taskSetMgr =>
8         try {
9           taskSetMgr.abort(msg)
10        } catch {
11          case e: Exception => logError("Exception in error callback", e)
12        }
13      }
14    } else {
15      val executorData = executorDataMap(task.executorId)
16      executorData.freeCores -= scheduler.CPUS_PER_TASK
17      executorData.executorEndpoint.send(LaunchTask(new
18        → SerializableBuffer(serializedTask)))
19    }
}

```

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 任务的状态更新为 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           </>

```

```

1 taskIdToTaskSetManager.get(tid) match {
2   case Some(taskSet) =>
3     if (TaskState.isFinished(state)) {
4       taskIdToTaskSetManager.remove(tid)
5       taskIdToExecutorId.remove(tid).foreach { execId =>
6         if (executorIdToTaskCount.contains(execId)) {
7           executorIdToTaskCount(execId) -= 1
8         }
9       }
10    }
11    if (state == TaskState.FINISHED) {
12      taskSet.removeRunningTask(tid)
13      taskResultGetter.enqueueSuccessfulTask(taskSet, tid,
14        ← serializedData)
15    } else if (Set(TaskState.FAILED, TaskState.KILLED,
16      ← TaskState.LOST).contains(state)) {
17      taskSet.removeRunningTask(tid)
18      taskResultGetter.enqueueFailedTask(taskSet, tid, state,
19        ← serializedData)
20    }
21    case None =>
22      logError(
23        ("Ignoring update with state %s for TID %s because its task set is
24          ← gone (this is " +
25        "likely the result of receiving duplicate task finished status
26          ← updates)")
27        .format(state, tid))
28  }

```

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

```

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 =
23          mapOutputTracker.getStatistics(shuffleStage.shuffleDep)
24        for (job <- shuffleStage.mapStageJobs) {
25          markMapStageJobAsFinished(job, stats)
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 shuffle 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

<i></></i>	程序 5.1: Driver 端 am 注册	<i></></i>
<pre> 1 allocator = client.register(driverUrl, 2 driverRef, 3 yarnConf, 4 _sparkConf, 5 uiAddress, 6 historyAddress, 7 securityMgr) 8 allocator.allocateResources() 9 reporterThread = launchReporterThread()</pre>		

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 ResourceManager

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。

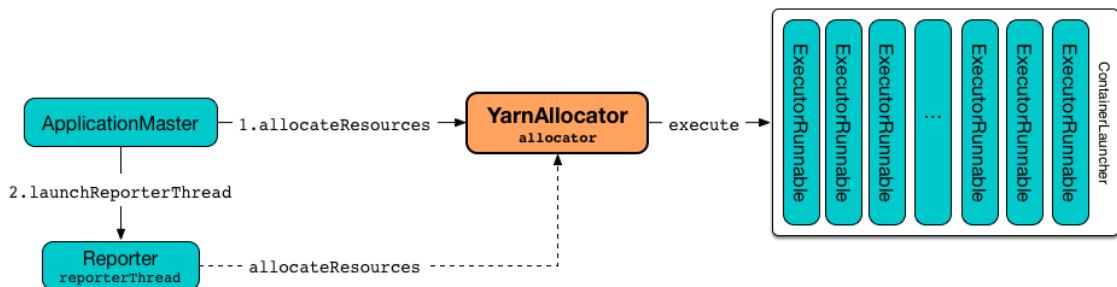


图 5.1 YarnAllocator 启动 Container 流程

程序 5.2: 启动 Executor

```

</>
1 for (container <- containersToUse) {
2   numExecutorsRunning += 1
3   val executorHostname = container.getNodeId.getHost
4   val containerId = container.getId
5   executorIdCounter += 1
6   val executorId = executorIdCounter.toString
7   executorIdToContainer(executorId) = container
8   containerIdToExecutorId(container.getId) = executorId
9   val containerSet =
10    → allocatedHostToContainersMap.getOrDefault(executorHostname,
11      new HashSet[ContainerId])
12   containerSet += containerId
13   allocatedContainerToHostMap.put(containerId, executorHostname)
14   val executorRunnable = new
15    → ExecutorRunnable(container, conf, sparkConf, driverUrl, executorId,
16    executorHostname, executorMemory, executorCores,
17    appAttemptId.getApplicationId.toString, securityMgr)
18   if (launchContainers) {
19     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，只会调用 run 方法，代码细节如程序5.3所示

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

1 SignalLogger.register(log)
2 SparkHadoopUtil.get.runAsSparkUser { () =>
3   val executorConf = new SparkConf
4   val port = executorConf.getInt("spark.executor.port", 0)
5   //这个 fetcher 通过 rpcenv 来获得 driver 的 ref，之后就关闭了
6   val fetcher = RpcEnv.create("driverPropsFetcher", hostname, port,
7     executorConf, new SecurityManager(executorConf), clientMode = true)
8   val driver = fetcher.setupEndpointRefByURI(driverUrl)
9   //获得 Driver 端的配置信息
10  val props = driver.askWithRetry[Seq[(String,
11    -> String)]](RetrieveSparkProps) ++
12    Seq[(String, String)](("spark.app.id", appId))
13  fetcher.shutdown()
14  // 使用从 Driver 端获取的配置信息创建 SparkEnv
15  val driverConf = new SparkConf()
16  for ((key, value) <- props) {
17    if (SparkConf.isExecutorStartupConf(key)) {
18      driverConf.setIfMissing(key, value)
19    } else {
20      driverConf.set(key, value)
21    }
22  }
23  //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 NodeManager

```

23  if (driverConf.contains("spark.yarn.credentials.file")) {
24
25      ← SparkHadoopUtil.get.startExecutorDelegationTokenRenewer(driverConf)
26
27  //创建 ExecutorEnv, 代码和 Driver 端的基本一摸一样
28  val env = SparkEnv.createExecutorEnv(driverConf, executorId,
29      ← hostname, port, cores, isLocal = false)
30  val sparkHostPort = env.conf.getOption("spark.executor.port").map {
31      ← port =>
32      hostname + ":" + port}.orNull
33  //这里创建了一个 Endpoint 用于与 driver 交互执行 task。是由此可以
34      ← 看出 Executor 的真正实现 CoarseGrainedExecutorBackend
35  env.rpcEnv.setupEndpoint("Executor", new
36      ← CoarseGrainedExecutorBackend(
37          env.rpcEnv, driverUrl, executorId, sparkHostPort, cores,
38          ← userClassPath, env))
39  //WorkerWatcher 就是用来监视 worker 的, 当与 worker 断开连接时, 它
40      ← 将会关闭这个 jvm 进程
41  workerUrl.foreach { url =>
42      env.rpcEnv.setupEndpoint("WorkerWatcher", new
43          ← WorkerWatcher(env.rpcEnv, url))
44  }
45  env.rpcEnv.awaitTermination()
46  SparkHadoopUtil.get.stopExecutorDelegationTokenRenewer()
47
48 }

```

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 </>
1 case LaunchTask(data) =>
2   if (executor == null) {
3     logError("Received LaunchTask command but executor was null")
4     System.exit(1)
5   } else {
6     val taskDesc = ser.deserialize[TaskDescription](data.value)
7     logInfo("Got assigned task " + taskDesc.taskId)
8     executor.launchTask(this, taskId = taskDesc.taskId, attemptNumber
9       ← = taskDesc.attemptNumber,
10      taskDesc.name, taskDesc.serializedTask)
11 }
```

Executor.launchTask 方法如程序块5.5所示

```
</> 程序 5.5: Executor.launcherTask </>
1 val tr = new TaskRunner(context, taskId = taskId, attemptNumber =
2   ← attemptNumber, taskName,
3   serializedTask)
4 runningTasks.put(taskId, tr)
5 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: Executor 执行 Task

```
</> //开始执行 Task
2 taskStart = System.currentTimeMillis()
3 var threwException = true
4 val (value, accumUpdates) = try {
5   val res = task.run(
6     taskAttemptId = taskId,
7     attemptNumber = attemptNumber,
8     metricsSystem = env.metricsSystem)
9   threwException = false
10  res
11 } finally {
12   .....
13 }
14 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

```

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

1 private[spark] def markTaskCompleted(): Unit = {
2   completed = true
3   val errorMessages = new ArrayBuffer[String](2)
4   onCompleteCallbacks.reverse.foreach { listener =>
5     try {
6       listener.onTaskCompletion(this)
7     } catch {
8       case e: Throwable =>
9         errorMessages += e.getMessage
10    }
11  }
12  if (errorMessages.nonEmpty) {
13    throw new TaskCompletionListenerException(errorMessages)
14  }
15}

```

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

```

</>           程序 5.8: Task run 方法核心部分           </>

1 //设置上下文信息，实际上会调用
2   ↳ org.apache.spark.TaskContext#setTaskContext
3 TaskContext.setTaskContext(context)
4 //更新 metrics 信息
5 context.taskMetrics.setHostname(Utils.localHostName())
6 context.taskMetrics.setAccumulatorsUpdater(context.collectInternalAccumulators)
7 //当前线程，在被打断的时候可以通过它来停止该线程
8 taskThread = Thread.currentThread()
9 if (_killed) { //如果当前 Task 被杀死，那么需要退出 Task 的执行
10   kill(interruptThread = false)
11 }

```

```

11 try {
12   //执行本次 Task, runTask 真正执行代码为子类中重写的 runTask, 对应
13   //→ ShuffleMapTask 和 ResultTask
14   (runTask(context), context.collectAccumulators())
15 } catch {
16 } finally {
17   // Call the task completion callbacks.
18   context.markTaskCompleted()
19 }
20 } finally {
21   TaskContext.unset()
22 }
23

```

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

```

</>                                         程序 5.9: 两种 Task 共同的处理方式 </>

1 // Deserialize the RDD using the broadcast variable.
2 val deserializeStartTime = System.currentTimeMillis()
3 //获取反序列化实例
4 val ser = SparkEnv.get.closureSerializer.newInstance()
5 //以下两端代码是其区别，上面的为 ShuffleMapTask，下面为 ResultTask
6 val (rdd, dep) = ser.deserialize[(RDD[_], ShuffleDependency[_], _,
7   ← _)](
8   //获取 rdd 和作用于 rdd 结果的函数
9   val (rdd, func) = ser.deserialize[(RDD[T], (TaskContext, Iterator[T]),
10   ← => U)](
11
12   ByteBuffer.wrap(taskBinary.value),
13   ← Thread.currentThread.getContextClassLoader)

```

```

11 _executorDeserializeTime = System.currentTimeMillis() -
    ↵ deserializeStartTime
12 //Task 的测量信息
13 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 </>

```

1 //从 SparkEnv 中获得 shuffleManager, 具体分析会在 shuffle 模块讲
   ↵ 述
2 val manager = SparkEnv.get.shuffleManager
3 //从 shuffleManager 获得写入器
4 writer = manager.getWriter[Any, Any](dep.shuffleHandle,
   ↵ partitionId, context)
5 //调用 rdd 开始计算, 并且最后通过写入器写入文件系统
6 writer.write(rdd.iterator(partition,
   ↵ context).asInstanceOf[Iterator[_ <: Product2[Any, Any]]])
7 //关闭写入器, 并将最后结果返回
8 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 </>

```

1 //调用 rdd 的迭代器执行传入函数的操作
2 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

<i></></i>	程序 5.12: Executor 对 Task 结果的处理	<i></></i>
<pre> 1 //任务运行结果的处理 2 val resultSer = env.serializer.newInstance() 3 val beforeSerialization = System.currentTimeMillis() 4 //value 中保存 Task 的计算结果 5 val valueBytes = resultSer.serialize(value) 6 val afterSerialization = System.currentTimeMillis() 7 //首先将结果直接放入 org.apache.spark.scheduler.DirectTaskResult 8 val directResult = new DirectTaskResult(valueBytes, accumUpdates, ↵ task.metrics.orNull) 9 val serializedDirectResult = ser.serialize(directResult) 10 val resultSize = serializedDirectResult.limit </pre>		

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。

<i></></i>	程序 5.13: Executor 回传结果策略	<i></></i>
<pre> 1 // directSend = sending directly back to the driver 2 val serializedResult: ByteBuffer = { 3 if (maxResultSize > 0 && resultSize > maxResultSize) { 4 //如果 resultSize 大于 spark.driver.maxResultSize 设置的大小 ↵ 值，则直接丢弃 5 ser.serialize(new IndirectTaskResult[Any](TaskResultBlockId(taskId), ↵ resultSize)) 6 } else if (resultSize >= akkaFrameSize - AkkaUtils.reservedSizeBytes) { ↵ 7 //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 8 //AKKA 的消息默认大小可以通过 spark.akka.frameSize 来设置 9 val blockId = TaskResultBlockId(taskId) </pre>		

```

10 env.blockManager.putBytes(
11   blockId, serializedDirectResult, StorageLevel.MEMORY_AND_DISK_SER)
12 ser.serialize(new IndirectTaskResult[Any](blockId, resultSize))
13 } else {
14   //结果回传 Driver
15   serializedDirectResult
16 }
17 }
18 //通过 AKKA 向 driver 汇报本次 Task 已经完成
19 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	
	<pre> </> 程序 5.14: Executor 执行回传消息 </> 1 val msg = StatusUpdate(executorId, taskId, state, data) 2 driver match { 3 case Some(driverRef) => driverRef.send(msg) 4 case None => logWarning(s"Drop \$msg because has not yet connected to ↵ driver") 5 } </pre>	

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

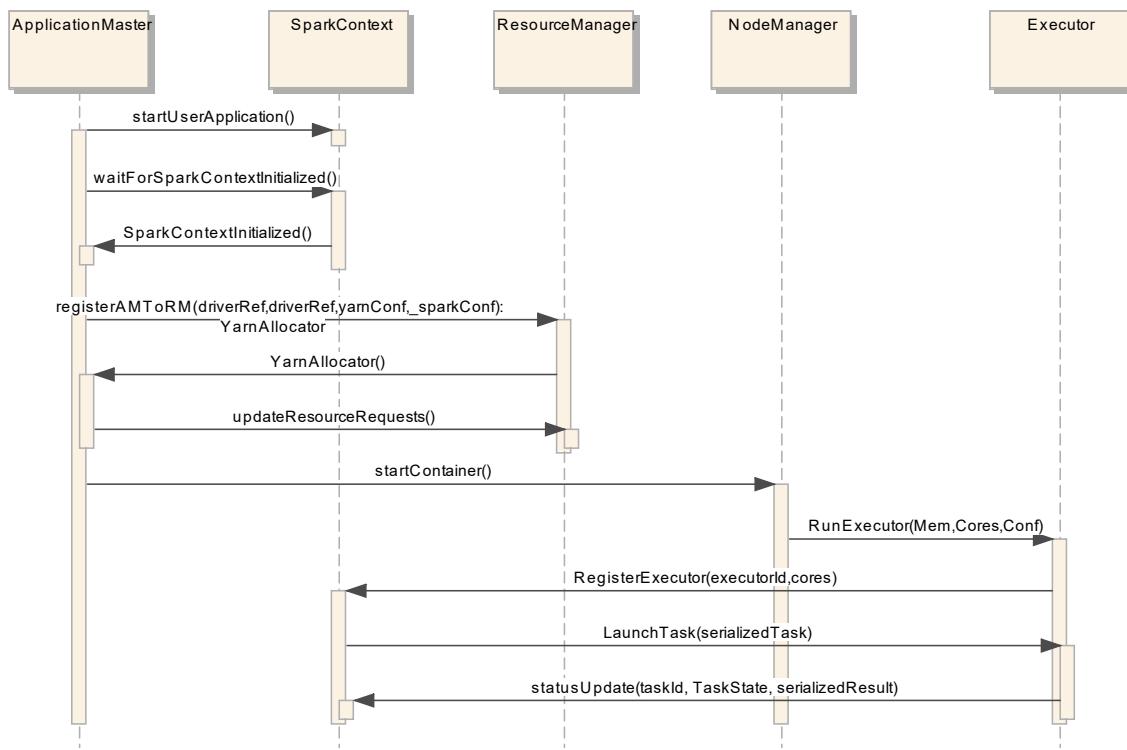


图 5.2 Executor 运行 Task 时序图

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 </>
1 // 指定 shuffle 的类型
2 val shortShuffleMgrNames = Map(
3   "hash" -> "org.apache.spark.shuffle.hash.HashShuffleManager",
4   "sort" -> "org.apache.spark.shuffle.sort.SortShuffleManager",
5   "tungsten-sort" -> "org.apache.spark.shuffle.sort.SortShuffleManager")
6 val shuffleMgrName = conf.get("spark.shuffle.manager", "sort")
7 val shuffleMgrClass =
8   ↳ shortShuffleMgrNames.getOrElse(shuffleMgrName.toLowerCase,
9   ↳ shuffleMgrName)
10 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           </>
    1 //从 SparkEnv 中获取 shuffleManager
    2 val manager = SparkEnv.get.shuffleManager
    3 //从 manager 中获取 Writer, 这里是 SortShuffleWriter
    4 writer = manager.getWriter[Any, Any](dep.shuffleHandle, partitionId,
      ↵ context)
    5 //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
    6 writer.write(rdd.iterator(partition, context).asInstanceOf[Iterator[_
      ↵ <: Product2[Any, Any]]])
    7 writer.stop(success = true).get

```

6.1 Shuffle Writer 历史版本

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。

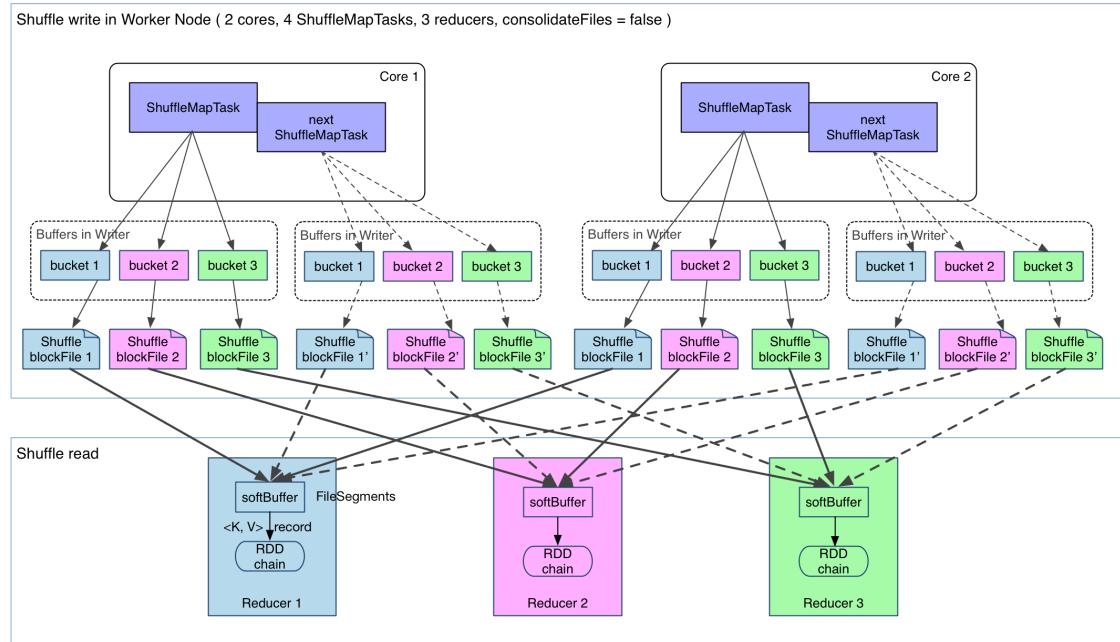


图 6.1 早期的 ShuffleWriter 工作原理

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 $M \times R$ ^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 $\text{core}^{\text{①}} * \text{R}$ 个文件。此种模式下运行原理如图6.2所示

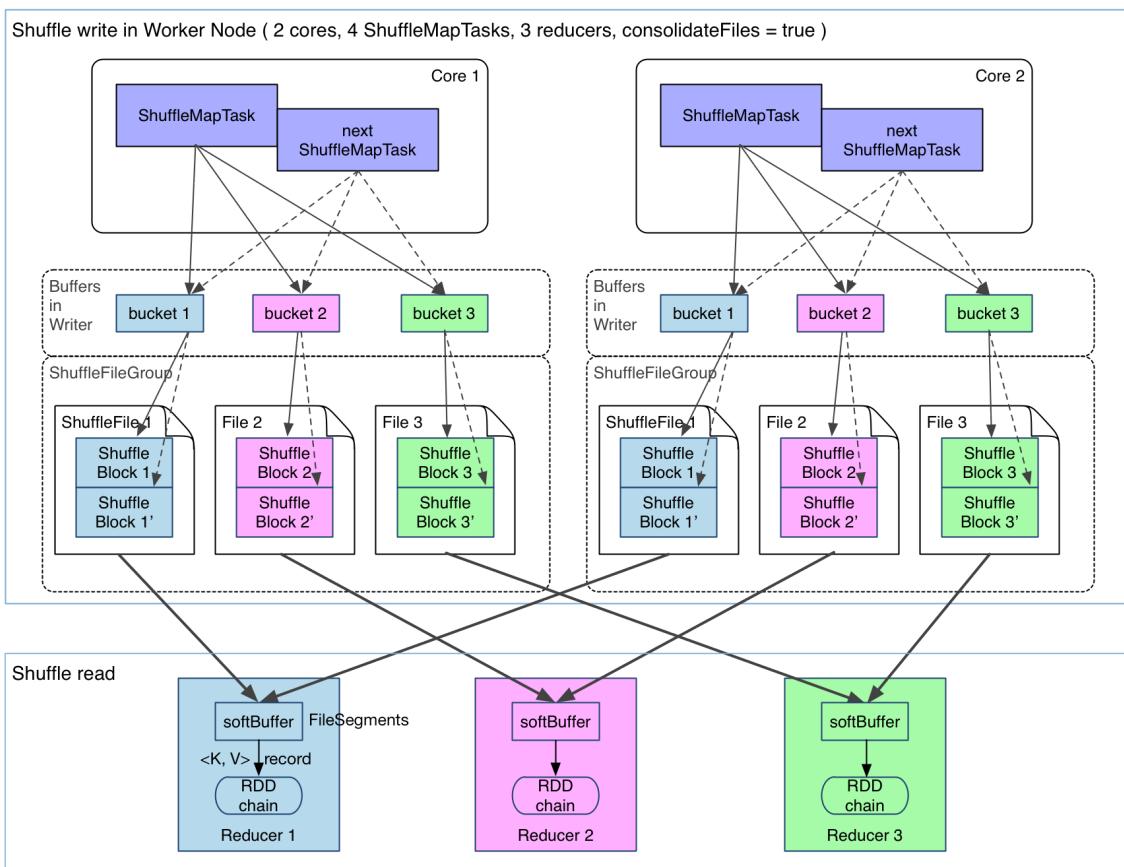


图 6.2 Shuffle Consolidate Files 工作原理

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

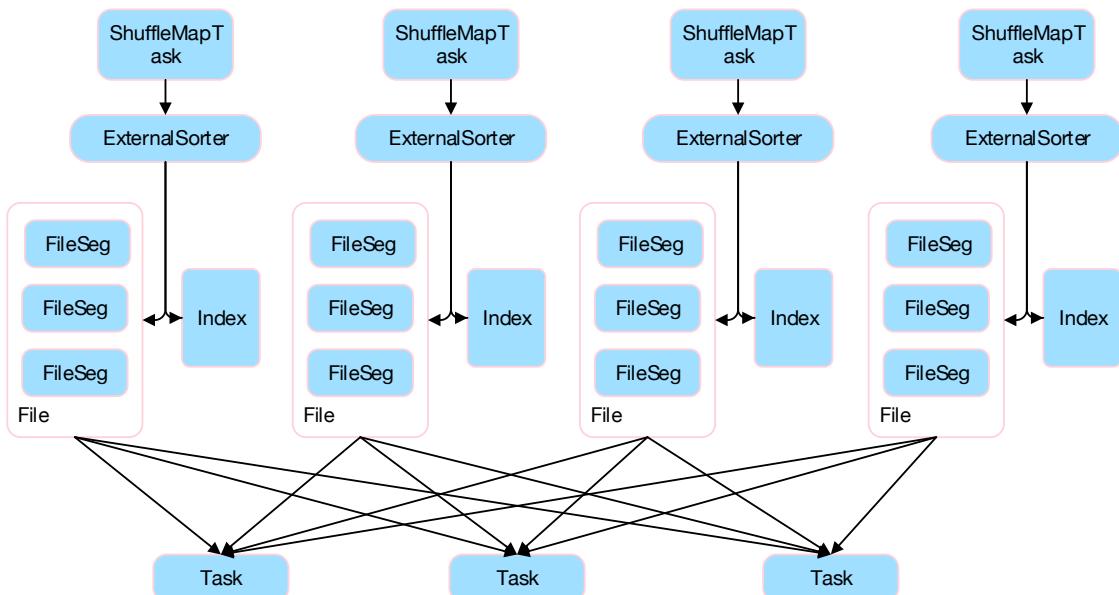


图 6.3 Sort Base Shuffle 工作原理

为了下游 Task 获得其所需要的分区，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 Spark1.6 版本 Shuffle Write

本节分析 Spakr1.6 中默认采用的 SortShuffleManager 对应的 writer。SortShuffleManager 中的 getWriter 会根据不同的 ShuffleHandle 产生相应的 ShuffleWriter

- 1 SerializedShuffleHandle 对应 UnsafeShuffleWriter
- 2 BypassMergeSortShuffleHandle 对应 BypassMergeSortShuffleWriter
- 3 BaseShuffleHandle 对应 SortShuffleWriter

6.2.1 BaseShuffleHandle 对应 SortShuffleWriter

其 writer 方法如程序6.3所示

</> 程序 6.3: SortShuffleWriter write 方法实现 </>

```

1 override def write(records: Iterator[Product2[K, V]]): Unit = {
2   //如果数据需要在 map 端 combine, 则需要传入 dep.aggregator, 下面的
3   //→ 这个 dep.keyOrdering 是空值,
4   //因为在 spark 的 sortedshuffle 中, 数据是不排序的。这里的这个
5   //→ partitioner 是用来为后续 RDD 构造 partitions 的
6   sorter = if (dep.mapSideCombine) {
7     require(dep.aggregator.isDefined, "Map-side combine without
8     → Aggregator specified!")
9     new ExternalSorter[K, V, C](
10       context, dep.aggregator, Some(dep.partitioner), dep.keyOrdering,
11       → dep.serializer)
12   } else {
13     //如果数据不需要在 map 端 combine, 则 aggregator 传 None 就行
14     new ExternalSorter[K, V, V](context, aggregator = None,
15       → Some(dep.partitioner), ordering = None, dep.serializer)
16   }
17   //将数据先放入缓存中, 如果缓存不够用 spill 到磁盘, 在这一步也会对
18   //→ 相同 key 值的数据进行 combine 操作
19   sorter.insertAll(records)
20   //打开一个文件
21   val output = shuffleBlockResolver.getDataFile(dep.shuffleId, mapId)
22   val tmp = Utils.tempFileWith(output)
23   //构造 blockId
24   val blockId = ShuffleBlockId(dep.shuffleId, mapId,
25     → IndexShuffleBlockResolver.NOOP_REDUCE_ID)
26   //将数据写入 data 文件
27   val partitionLengths = sorter.writePartitionedFile(blockId, tmp)
28   //将数据写入 index 文件
29   shuffleBlockResolver.writeIndexFileAndCommit(dep.shuffleId, mapId,
30     → partitionLengths, tmp)
31   //进行 shuffle read 时的一些参考信息

```

```

24 mapStatus = MapStatus(blockManager.shuffleServerId,
25   ↳ partitionLengths)
}

```

程序6.3中的 ExternalSorter 有下面三个作用

1. 实例化 ExternalSorter
2. 调用 insertAll() 方法对记录集进行操作
3. 返回已经排序或者聚合之后的记录的迭代器

6.2.2 实例化 ExternalSorter

首先就是实例化 ExternalSorter，这里有一个判断，如果要进行 map 端的 combine 操作的话就需要指定 Aggregator 和 Ordering，否则这两个参数为 None。本例采用的 reduceByKey 就进行了 Map 端的 combine 操作，由 ReduceByKey 算子构造 ShuffleRDD 的过程如图6.4所示

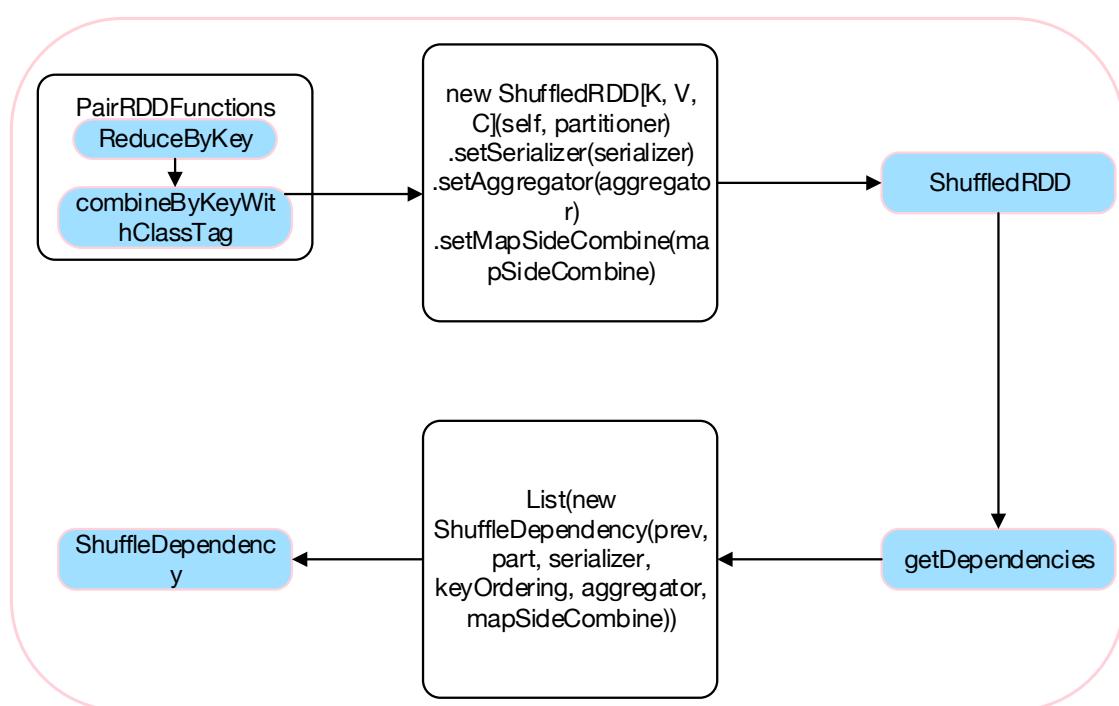


图 6.4 ReduceByKey 算子

6.2.3 shuffle 写操作

通过判断是否进行 Map 端 combine 操作而实例化出不同的 ExternalSorter 后，就会调用 insertAll 方法，将输入的记录写入到内存中，如果内存不足就 spill 到磁

盘中，具体的实现我们来看 insertAll 方法，如程序6.4所示

	程序 6.4: SortShuffleWriter insertAll 方法实现	
--	--	--

```

1  def insertAll(records: Iterator[Product2[K, V]]): Unit = {
2    val shouldCombine = aggregator.isDefined
3    //首先判断是否需要进行 Map 端的 combine 操作
4    if (shouldCombine) {
5      //如果需要进行 Map 端 combine 操作，使用 PartitionedAppendOnlyMap
6      //作为缓存
7      // mergeValue 其实就是我们用 reduceByKey 的时候参数值，即 _+_
8      val mergeValue = aggregator.get.mergeValue
9      //((C, V) => C =>func
10     //createCombiner 其实是和 mergeValue 一样的，都是 _+_
11     //，但是在使用
12     // createCombiner 中只传了一个参数，所以另一个参///数默认采用
13     // 默认值，这里为 0
14     val createCombiner = aggregator.get.createCombiner//V => C
15     var kv: Product2[K, V] = null
16     // 这个 update 也是一个函数，逻辑为如果某个 Key 已经存在记录
17     // record 就使用上面获取的聚合函数进行聚合操作，
18     // 如果不存在记录，就使用 createCombiner 方法进行初始化操作
19     val update = (hadValue: Boolean, oldValue: C) => {
20       if (hadValue) mergeValue(oldValue, kv._2) else
21         createCombiner(kv._2)
22     }
23     //遍历所有的 records
24     while (records.hasNext) {
25       //记录 spill 的频率，每当 read 一条 record 的时候就会记录一次
26       addElementsRead()
27       //使用 kv 存储当前读的 records
28       kv = records.next()
29       //对 key 相同的记录的 value 进行合并
30       //更新 map 中的数据，其中 private var map = new
31       // PartitionedAppendOnlyMap[K, C],

```

```

25 // 如果需要进行 combine 就将数据放在 map 中，然后在 map 中对数
   ↵ 据进行更新操作
26 map.changeValue((getPartition(kv._1), kv._1), update)
27 // 判断是否需要进行 spill 操作
28 maybeSpillCollection(usingMap = true)
29 }
30 } else {
31 // 如果不需要进行 Map 端的聚合操作，就直接将记录放到
   ↵ buffer(PartitionedPairBuffer) 中
32 while (records.hasNext) {
33 addElementsRead()
34 val kv = records.next()
35 buffer.insert(getPartition(kv._1), kv._1, kv._2.asInstanceOf[C])
36 maybeSpillCollection(usingMap = false)
37 }
38 }
39 }

```

PartitionedAppendOnlyMap 和 PartitionedPairBuffer 其类图如图6.5所示

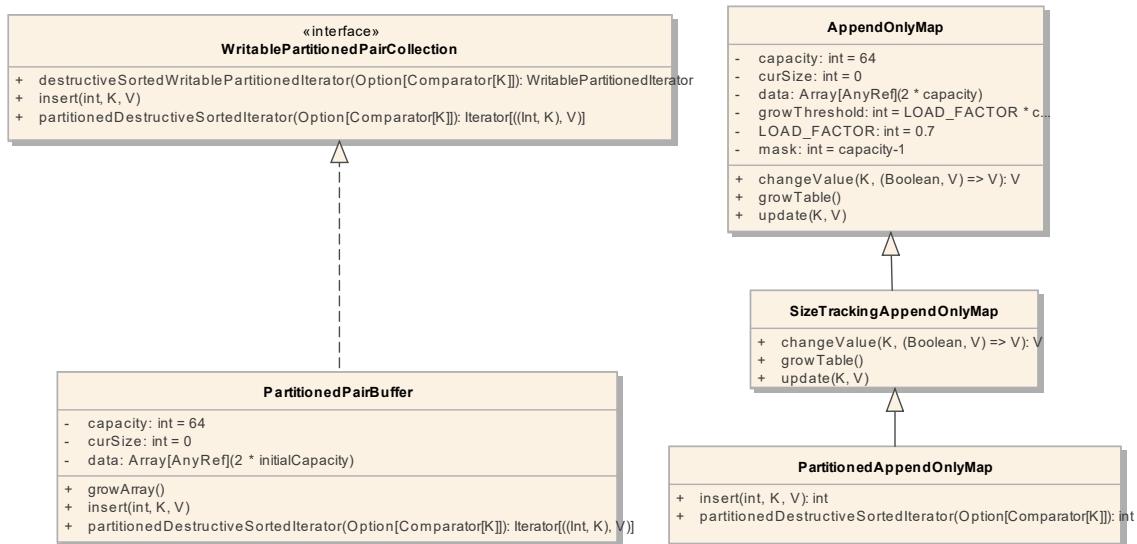


图 6.5 Map 和 Buffer 类图

由上图可以看出 reduceBy 算法需要 map 端聚合，`map.changeValue` 调用的是

AppendOnlyMap 中的 changeValue, 完成的工作是将记录写到 map 缓存中, 按照 reduceByKey 算子中定义的合并方法对相同 key 的 value 进行处理。如程序6.5所示

</> **程序 6.5: 缓存并更新 value 值** </>

```

1 def changeValue(key: K, updateFunc: (Boolean, V) => V): V = {
2   assert(!destroyed, destructionMessage)
3   val k = key.asInstanceOf[AnyRef]
4   //如果传入的 key 为空值, 队列自动增加长度,
5   // 这里的设计很巧妙: 因为队列自动增加, 就肯定会多出一个值, 如果你
6   // → 不给它赋值的话, 它就为 null,
7   // 但是这个值又不占队列中具体的位置, 只要在最后的时候保留一个没有
8   // → 赋值的位置即可。
9
10  if (k.eq(null)) {
11    if (!haveNullValue) {
12      incrementSize()
13    }
14    nullValue = updateFunc(haveNullValue, nullValue)
15    haveNullValue = true
16    return nullValue
17  }
18
19  var pos = rehash(k.hashCode) & mask
20  var i = 1
21  while (true) {
22    //data 是一个数组, 用来同时存储 key 和 value: key0, value0, key1,
23    // → value1, key2, value2, etc.
24    // 即 2 * pos 上存储的是 key 的值, 2 * pos + 1 上存储的是 value
25    // → 的值
26    val curKey = data(2 * pos)
27    if (k.eq(curKey) || k.equals(curKey)) {
28      //如果有相同的值的话, 则根据 updateFunc 更新这个 key 对应的
29      // → value
30      val newValue = updateFunc(true, data(2 * pos +
31      // → 1).asInstanceOf[V])
32    }
33  }
34}

```

```

24     data(2 * pos + 1) = newValue.asInstanceOf[AnyRef]
25     return newValue
26 } else if (curKey.eq(null)) {
27     //如果 key 不存在就将该 key 和对应的 value 添加到 data 这个数
28     //组中
29     val newValue = updateFunc(false, null.asInstanceOf[V])
30     data(2 * pos) = k
31     data(2 * pos + 1) = newValue.asInstanceOf[AnyRef]
32     incrementSize()
33     return newValue
34 } else {
35     //如果 pos 和其它的 key 重合，则继续计算其位置
36     val delta = i
37     pos = (pos + delta) & mask
38     i += 1
39 }
40 null.asInstanceOf[V] // Never reached but needed to keep compiler
41     ← happy
}

```

6.2.4 shuffle 结果本地缓存

这部分将 shuffle 结果缓存到本地，并为之生成索引文件，同时返回文件的迭代器。整体过程如程序6.6所示

</> 程序 6.6: 完成分区文件和索引文件的本地缓存 </>

```

1 //打开一个文件
2 val output = shuffleBlockResolver.getDataFile(dep.shuffleId, mapId)
3 val tmp = Utils.tempFileWith(output)
4 //构造 blockId
5 val blockId = ShuffleBlockId(dep.shuffleId, mapId,
    ← IndexShuffleBlockResolver.NOP_REDUCE_ID)

```

```

6 //将数据写入 data 文件
7 val partitionLengths = sorter.writePartitionedFile(blockId, tmp)
8 //将数据写入 index 文件
9 shuffleBlockResolver.writeIndexFileAndCommit(dep.shuffleId, mapId,
   ↳ partitionLengths, tmp)
10 //进行 shuffle read 时的一些参考信息
11 mapStatus = MapStatus(blockManager.shuffleServerId, partitionLengths)

```

6.2.4.1 写分区文件

此步骤负责将缓存在内存 map 或者 buffer 中的记录缓存到本地磁盘中，如果之前已经有部分数据溢出到磁盘，会将这些一起进行合并。写分区文件过程如程序6.7所示

</> **程序 6.7: 生成分区文件并写入数据** </>

```

1 def writePartitionedFile(
2   blockId: BlockId,
3   outputFile: File): Array[Long] = {
4   val lengths = new Array[Long](numPartitions)
5   //首先判断是否有数据已经 spill 到磁盘了
6   if (spills.isEmpty) {
7     //当只有内存中有数据时
8     val collection = if (aggregator.isDefined) map else buffer
9     // 获得迭代器
10    val it =
11      ↳ collection.destructiveSortedWritablePartitionedIterator(comparator)
12    //进行迭代并将数据写到磁盘
13    while (it.hasNext) {
14      val writer = blockManager.getDiskWriter(blockId, outputFile,
15        ↳ serInstance,
16        ↳ fileBufferSize, context.taskMetrics.shuffleWriteMetrics.get)
17      val partitionId = it.nextPartition()
18      while (it.hasNext && it.nextPartition() == partitionId) {
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65
66
67
68
69
70
71
72
73
74
75
76
77
78
79
80
81
82
83
84
85
86
87
88
89
90
91
92
93
94
95
96
97
98
99
100
101
102
103
104
105
106
107
108
109
110
111
112
113
114
115
116
117
118
119
120
121
122
123
124
125
126
127
128
129
130
131
132
133
134
135
136
137
138
139
140
141
142
143
144
145
146
147
148
149
150
151
152
153
154
155
156
157
158
159
159
160
161
162
163
164
165
166
167
168
169
169
170
171
172
173
174
175
176
177
178
179
179
180
181
182
183
184
185
186
187
187
188
189
189
190
191
192
193
194
195
196
197
197
198
199
199
200
201
202
203
204
205
205
206
207
207
208
209
209
210
211
211
212
212
213
213
214
214
215
215
216
216
217
217
218
218
219
219
220
220
221
221
222
222
223
223
224
224
225
225
226
226
227
227
228
228
229
229
230
230
231
231
232
232
233
233
234
234
235
235
236
236
237
237
238
238
239
239
240
240
241
241
242
242
243
243
244
244
245
245
246
246
247
247
248
248
249
249
250
250
251
251
252
252
253
253
254
254
255
255
256
256
257
257
258
258
259
259
260
260
261
261
262
262
263
263
264
264
265
265
266
266
267
267
268
268
269
269
270
270
271
271
272
272
273
273
274
274
275
275
276
276
277
277
278
278
279
279
280
280
281
281
282
282
283
283
284
284
285
285
286
286
287
287
288
288
289
289
290
290
291
291
292
292
293
293
294
294
295
295
296
296
297
297
298
298
299
299
300
300
301
301
302
302
303
303
304
304
305
305
306
306
307
307
308
308
309
309
310
310
311
311
312
312
313
313
314
314
315
315
316
316
317
317
318
318
319
319
320
320
321
321
322
322
323
323
324
324
325
325
326
326
327
327
328
328
329
329
330
330
331
331
332
332
333
333
334
334
335
335
336
336
337
337
338
338
339
339
340
340
341
341
342
342
343
343
344
344
345
345
346
346
347
347
348
348
349
349
350
350
351
351
352
352
353
353
354
354
355
355
356
356
357
357
358
358
359
359
360
360
361
361
362
362
363
363
364
364
365
365
366
366
367
367
368
368
369
369
370
370
371
371
372
372
373
373
374
374
375
375
376
376
377
377
378
378
379
379
380
380
381
381
382
382
383
383
384
384
385
385
386
386
387
387
388
388
389
389
390
390
391
391
392
392
393
393
394
394
395
395
396
396
397
397
398
398
399
399
400
400
401
401
402
402
403
403
404
404
405
405
406
406
407
407
408
408
409
409
410
410
411
411
412
412
413
413
414
414
415
415
416
416
417
417
418
418
419
419
420
420
421
421
422
422
423
423
424
424
425
425
426
426
427
427
428
428
429
429
430
430
431
431
432
432
433
433
434
434
435
435
436
436
437
437
438
438
439
439
440
440
441
441
442
442
443
443
444
444
445
445
446
446
447
447
448
448
449
449
450
450
451
451
452
452
453
453
454
454
455
455
456
456
457
457
458
458
459
459
460
460
461
461
462
462
463
463
464
464
465
465
466
466
467
467
468
468
469
469
470
470
471
471
472
472
473
473
474
474
475
475
476
476
477
477
478
478
479
479
480
480
481
481
482
482
483
483
484
484
485
485
486
486
487
487
488
488
489
489
490
490
491
491
492
492
493
493
494
494
495
495
496
496
497
497
498
498
499
499
500
500
501
501
502
502
503
503
504
504
505
505
506
506
507
507
508
508
509
509
510
510
511
511
512
512
513
513
514
514
515
515
516
516
517
517
518
518
519
519
520
520
521
521
522
522
523
523
524
524
525
525
526
526
527
527
528
528
529
529
530
530
531
531
532
532
533
533
534
534
535
535
536
536
537
537
538
538
539
539
540
540
541
541
542
542
543
543
544
544
545
545
546
546
547
547
548
548
549
549
550
550
551
551
552
552
553
553
554
554
555
555
556
556
557
557
558
558
559
559
560
560
561
561
562
562
563
563
564
564
565
565
566
566
567
567
568
568
569
569
570
570
571
571
572
572
573
573
574
574
575
575
576
576
577
577
578
578
579
579
580
580
581
581
582
582
583
583
584
584
585
585
586
586
587
587
588
588
589
589
590
590
591
591
592
592
593
593
594
594
595
595
596
596
597
597
598
598
599
599
600
600
601
601
602
602
603
603
604
604
605
605
606
606
607
607
608
608
609
609
610
610
611
611
612
612
613
613
614
614
615
615
616
616
617
617
618
618
619
619
620
620
621
621
622
622
623
623
624
624
625
625
626
626
627
627
628
628
629
629
630
630
631
631
632
632
633
633
634
634
635
635
636
636
637
637
638
638
639
639
640
640
641
641
642
642
643
643
644
644
645
645
646
646
647
647
648
648
649
649
650
650
651
651
652
652
653
653
654
654
655
655
656
656
657
657
658
658
659
659
660
660
661
661
662
662
663
663
664
664
665
665
666
666
667
667
668
668
669
669
670
670
671
671
672
672
673
673
674
674
675
675
676
676
677
677
678
678
679
679
680
680
681
681
682
682
683
683
684
684
685
685
686
686
687
687
688
688
689
689
690
690
691
691
692
692
693
693
694
694
695
695
696
696
697
697
698
698
699
699
700
700
701
701
702
702
703
703
704
704
705
705
706
706
707
707
708
708
709
709
710
710
711
711
712
712
713
713
714
714
715
715
716
716
717
717
718
718
719
719
720
720
721
721
722
722
723
723
724
724
725
725
726
726
727
727
728
728
729
729
730
730
731
731
732
732
733
733
734
734
735
735
736
736
737
737
738
738
739
739
740
740
741
741
742
742
743
743
744
744
745
745
746
746
747
747
748
748
749
749
750
750
751
751
752
752
753
753
754
754
755
755
756
756
757
757
758
758
759
759
760
760
761
761
762
762
763
763
764
764
765
765
766
766
767
767
768
768
769
769
770
770
771
771
772
772
773
773
774
774
775
775
776
776
777
777
778
778
779
779
780
780
781
781
782
782
783
783
784
784
785
785
786
786
787
787
788
788
789
789
790
790
791
791
792
792
793
793
794
794
795
795
796
796
797
797
798
798
799
799
800
800
801
801
802
802
803
803
804
804
805
805
806
806
807
807
808
808
809
809
810
810
811
811
812
812
813
813
814
814
815
815
816
816
817
817
818
818
819
819
820
820
821
821
822
822
823
823
824
824
825
825
826
826
827
827
828
828
829
829
830
830
831
831
832
832
833
833
834
834
835
835
836
836
837
837
838
838
839
839
840
840
841
841
842
842
843
843
844
844
845
845
846
846
847
847
848
848
849
849
850
850
851
851
852
852
853
853
854
854
855
855
856
856
857
857
858
858
859
859
860
860
861
861
862
862
863
863
864
864
865
865
866
866
867
867
868
868
869
869
870
870
871
871
872
872
873
873
874
874
875
875
876
876
877
877
878
878
879
879
880
880
881
881
882
882
883
883
884
884
885
885
886
886
887
887
888
888
889
889
890
890
891
891
892
892
893
893
894
894
895
895
896
896
897
897
898
898
899
899
900
900
901
901
902
902
903
903
904
904
905
905
906
906
907
907
908
908
909
909
910
910
911
911
912
912
913
913
914
914
915
915
916
916
917
917
918
918
919
919
920
920
921
921
922
922
923
923
924
924
925
925
926
926
927
927
928
928
929
929
930
930
931
931
932
932
933
933
934
934
935
935
936
936
937
937
938
938
939
939
940
940
941
941
942
942
943
943
944
944
945
945
946
946
947
947
948
948
949
949
950
950
951
951
952
952
953
953
954
954
955
955
956
956
957
957
958
958
959
959
960
960
961
961
962
962
963
963
964
964
965
965
966
966
967
967
968
968
969
969
970
970
971
971
972
972
973
973
974
974
975
975
976
976
977
977
978
978
979
979
980
980
981
981
982
982
983
983
984
984
985
985
986
986
987
987
988
988
989
989
990
990
991
991
992
992
993
993
994
994
995
995
996
996
997
997
998
998
999
999
1000
1000
1001
1001
1002
1002
1003
1003
1004
1004
1005
1005
1006
1006
1007
1007
1008
1008
1009
1009
1010
1010
1011
1011
1012
1012
1013
1013
1014
1014
1015
1015
1016
1016
1017
1017
1018
1018
1019
1019
1020
1020
1021
1021
1022
1022
1023
1023
1024
1024
1025
1025
1026
1026
1027
1027
1028
1028
1029
1029
1030
1030
1031
1031
1032
1032
1033
1033
1034
1034
1035
1035
1036
1036
1037
1037
1038
1038
1039
1039
1040
1040
1041
1041
1042
1042
1043
1043
1044
1044
1045
1045
1046
1046
1047
1047
1048
1048
1049
1049
1050
1050
1051
1051
1052
1052
1053
1053
1054
1054
1055
1055
1056
1056
1057
1057
1058
1058
1059
1059
1060
1060
1061
1061
1062
1062
1063
1063
1064
1064
1065
1065
1066
1066
1067
1067
1068
1068
1069
1069
1070
1070
1071
1071
1072
1072
1073
1073
1074
1074
1075
1075
1076
1076
1077
1077
1078
1078
1079
1079
1080
1080
1081
1081
1082
1082
1083
1083
1084
1084
1085
1085
1086
1086
1087
1087
1088
1088
1089
1089
1090
1090
1091
1091
1092
1092
1093
1093
1094
1094
1095
1095
1096
1096
1097
1097
1098
1098
1099
1099
1100
1100
1101
1101
1102
1102
1103
1103
1104
1104
1105
1105
1106
1106
1107
1107
1108
1108
1109
1109
1110
1110
1111
1111
1112
1112
1113
1113
1114
1114
1115
1115
1116
1116
1117
1117
1118
1118
1119
1119
1120
1120
1121
1121
1122
1122
1123
1123
1124
1124
1125
1125
1126
1126
1127
1127
1128
1128
1129
1129
1130
1130
1131
1131
1132
1132
1133
1133
1134
1134
1135
1135
1136
1136
1137
1137
1138
1138
1139
1139
1140
1140
1141
1141
1142
1142
1143
1143
1144
1144
1145
1145
1146
1146
1147
1147
1148
1148
1149
1149
1150
1150
1151
1151
1152
1152
1153
1153
1154
1154
1155
1155
1156
1156
1157
1157
1158
1158
1159
1159
1160
1160
1161
1161
1162
1162
1163
1163
1164
1164
1165
1165
1166
1166
1167
1167
1168
1168
1169
1169
1170
1170
1171
1171
1172
1172
1173
1173
1174
1174
1175
1175
1176
1176
1177
1177
1178
1178
1179
1179
1180
1180
1181
1181
1182
1182
1183
1183
1184
1184
1185
1185
1186
1186
1187
1187
1188
1188
1189
1189
1190
1190
1191
1191
1192
1192
1193
1193
1194
1194
1195
1195
1196
1196
1197
1197
1198
1198
1199
1199
1200
1200
1201
1201
1202
1202
1203
1203
1204
1204
1205
1205
1206
1206
1207
1207
1208
1208
1209
1209
1210
1210
1211
1211
1212
1212
1213
1213
1214
1214
1215
1215
1216
1216
1217
1217
1218
1218
1219
1219
1220
1220
1221
1221
1222
1222
1223
1223
1224
1224
1225
1225
1226
1226
1227
1227
1228
1228
1229
1229
1230
1230
1231
1231
1232
1232
1233
1233
1234
1234
1235
1235
1236
1236
1237
1237
1238
1238
1239
1239
1240
1240
1241
1241
1242
1242
1243
1243
1244
1244
1245
1245
1246
1246
1247
1247
1248
1248
1249
1249
1250
1250
1251
1251
1252
1252
1253
1253
1254
1254
1255
1255
1256
1256
1257
1257
1258
1258
1259
1259
1260
1260
1261
1261
1262
1262
1263
1263
1264
1264
1265
1265
1266
1266
1267
1267
1268
1268
1269
1269
1270
1270
1271
1271
1272
1272
1273
1273
1274
1274
1275
1275
1276
1276
1277
1277
1278
1278
1279
1279
1280
1280
1281
1281
1282
1282
1283
1283
1284
1284
1285
1285
1286
1286
1287
1287
1288
1288
1289
1289
1290
1290
1291
1291
1292
1292
1293
1293
1294
1294
1295
1295
1296
1296
1297
1297
1298
1298
1299
1299
1300
1300
1301
1301
1302
1302
1303
1303
1304
1304
1305
1305
1306
1306
1307
1307
1308
1308
1309
1309
1310
1310
1311
1311
1312
1312
1313
1313
1314
1314
1315
1315
1316
1316
1317
1317
1318
1318
1319
1319
1320
1320
1321
1321
1322
1322
1323
1323
1324
1324
1325
1325
1326
1326
1327
1327
1328
1328
1329
1329
1330
1330
1331
1331
1332
1332
1333
1333
1334
1334
1335
1335
1336
1336
1337
1337
1338
1338
1339
1339
1340
1340
1341
1341
1342
1342
1343
1343
1344
1344
1345
1345
1346
1346
1347
1347
1348
1348
1349
1349
1350
1350
135
```

```
16 //把同一个分区的记录写到一块 并返回该 blockId 中该
   ↳ partitionId 的长度
17     it.writeNext(writer)
18 }
19 writer.commitAndClose()
20 val segment = writer.fileSegment()
21 //最后返回的是每个 partition 写入数据的长度
22 lengths(partitionId) = segment.length
23 }
24 } else {
25 //需要持久化到硬盘中的情况，在调用 partitionedIterator 方法后，
   ↳ 会对应的 partition 中的已经 combine 过的数据
26 for ((id, elements) <- this.partitionedIterator) {
27     if (elements.hasNext) {
28         val writer = blockManager.getDiskWriter(blockId, outputFile,
           ↳ serInstance, fileBufferSize,
           context.taskMetrics.shuffleWriteMetrics.get)
29         //将 partition 中的数据写入 data 文件
30         for (elem <- elements) {
31             writer.write(elem._1, elem._2)
32         }
33         writer.commitAndClose()
34         val segment = writer.fileSegment()
35         //最后返回的是每个 partition 写入数据的长度
36         lengths(id) = segment.length
37     }
38 }
39 }
40 }
41 //更改 task 测量系统中的参数值
42 .....
43 lengths
44 }
```

上面函数通过判断 spills 是否为空进行了不同的处理，spills 为空时表示记录全部缓存在内存中只需要从内存 map 或者 buffer 中取出数据写到本地磁盘即可，spills 不为空时，得将已经写到磁盘的和内存中缓存的部分进行合并。接下来分析 spills 为空时的情况。

程序会获取 collection 的迭代器，这里会传入比较器，其中 destructiveSortedWritablePartitionedIterator 是个接口方法，其内部逻辑如程序6.8所示

程序 6.8: 排序分区迭代器

```

1 def destructiveSortedWritablePartitionedIterator(keyComparator:
2   → Option[Comparator[K]]) {
3   // 这里的 partitionedDestructiveSortedIterator 会根据是 map 或者
4   → buffer 有不同的实现
5   val it = partitionedDestructiveSortedIterator(keyComparator)
6   // 最后返回的是 WritablePartitionedIterator, 上面进行写操作的时候
7   → 就是调用该迭代器中的 writeNext 方法
8   new WritablePartitionedIterator {
9     private[this] var cur = if (it.hasNext) it.next() else null
10    def writeNext(writer: DiskBlockObjectWriter): Unit = {
11      writer.write(cur._1._2, cur._2)//((partitionId,K),V)
12      cur = if (it.hasNext) it.next() else null
13    }
14    def hasNext(): Boolean = cur != null
15    def nextPartition(): Int = cur._1._1
16  }
17 }
```

程序6.8中 partitionedDestructiveSortedIterator 会根据是 map 或者 buffer 有不同的实现，由图6.5可以看出 map 时其实现在 PartitionedAppendOnlyMap，从其实现上可以看出排序逻辑，如程序6.9所示

程序 6.9: PartitionedAppendOnlyMap 分区排序迭代器实现

```

1 def partitionedDestructiveSortedIterator(keyComparator:
2   → Option[Comparator[K]])
```

```

2 : Iterator[((Int, K), V)] = {
3   //先它这里会构造一个按 partition 和 key 进行比较的比较器，这个比较
   //器先按 partition 来排序，
4   //然后再用 key 的 hashCode 进行排序，记住这里 spark 系统是进行 key
   //排序了的，而不是没有排序，只是它这里是按照 key 的 hashCode 进
   //行的排序。
5   val comparator =
   //keyComparator.map(partitionKeyComparator).getOrElse(partitionComparator)
6   //后通过这个比较器再对这个 map 进行排序
7   destructiveSortedIterator(comparator)
8 }

```

6.2.4.2 写索引文件

具体的实现就不详细说明了，主要就是根据上一步的到的 partition 长度的数组将偏移量写入到 index 文件中。

所以最终每一个 task 的文件最终产生两个文件，而一个 spark 程序 shuffle 产生的数据就取决于并行度或者说 reducer 的数量，即 $2 * \text{reducer}$ 个。

最后就是实例化 MapStatus，shuffle read 的时候根据 MapStatus 获取数据。

6.2.5 BypassMergeSortShuffleHandle 对应 BypassMergeSortShuffleWriter

该 writer 在进行写记录之前会根据 reducer 的个数（例如 R 个）生成 R 个临时文件，然后将记录写入对应的临时文件中，最后将这些文件进行合并操作并写入到一个文件中，由于直接将记录写入了临时文件，并没有缓存在内存中，所以如果 reducer 的个数过多的话，就会为每个 reducer 打开一个临时文件，如果 reducer 的数量过多的话就会影响性能，所以使用该种方式需要满足一些条件

- 1 无排序器
- 2 无聚合器
- 3 reduce 端分区数量 <spark.shuffle.sort.bypassMergeThreshold. 默认值为 200

6.2.6 SerializedShuffleHandle 对应 UnsafeShuffleWriter

这部分也就是通常所说的 Tungsten，在此不做细讲。

6.2.7 Shuffle Write 流程图

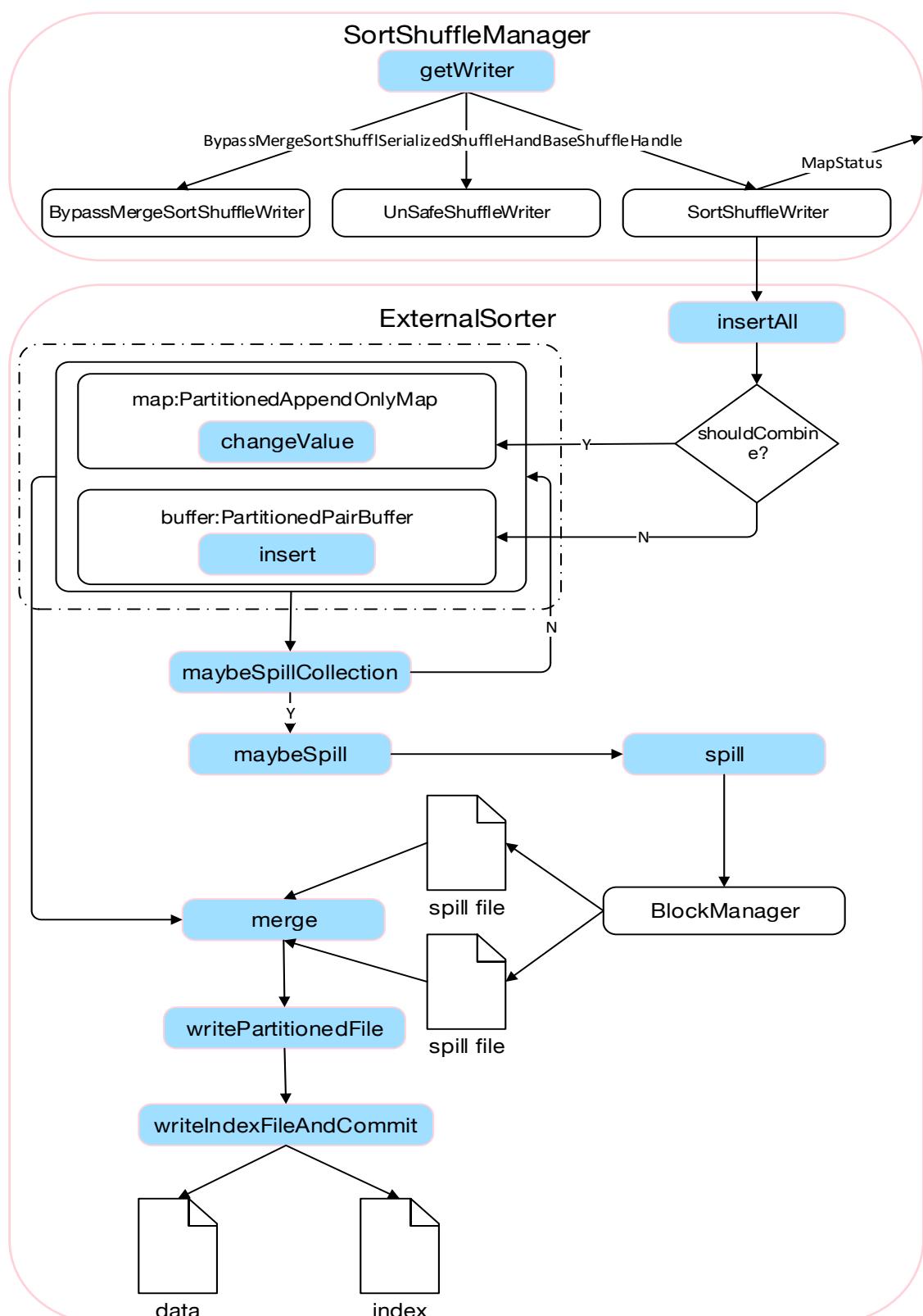


图 6.6 SortShuffleWriter 流程图

SortShuffleWriter 具体流程如图6.6所示

6.3 Spark1.6 版本 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，而不管是 HashShuffleManager 还是 SortShuffleManager，其 getReader 方法内部都是实例化了 BlockStoreShuffleReader，而 BlockStoreShuffleReader 正是实现了 ShuffleReader 接口。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 spark.reducer.maxMbInFlight，这里用 sortBuffer 表示，默认大小为 48MB。核心的 read 实现如程序6.10所示

</>
程序 6.10: *Shuffle Read* 的实现
</>

```

1 override def read(): Iterator[Product2[K, C]] = {
2   val blockFetcherItr = new
3     ShuffleBlockFetcherIterator(context, blockManager.shuffleClient,
4   blockManager, mapOutputTracker.getMapSizesByExecutorId(handle.shuffleId,
5     startPartition, endPartition),
6   SparkEnv.get.conf.getSizeAsMb("spark.reducer.maxSizeInFlight",
7     "48m") * 1024 * 1024)
8   // 将上面获取的信息进行压缩处理
9   val wrappedStreams = blockFetcherItr.map { case (blockId,
10     inputStream) =>
11     blockManager.wrapForCompression(blockId, inputStream)
12   }
13   // 获取序列化器
14   val ser = Serializer.getSerializer(dep.serializer)
15   .....
16   val interruptibleIter = new InterruptibleIterator[(Any,
17     Any)](context, metricIter)

```

```

13  val aggregatedIter: Iterator[Product2[K, C]] = if
14    ← (dep.aggregator.isDefined) { //需要聚合
15      if (dep.mapSideCombine) { //需要 map 端聚合
16        val combinedKeyValuesIterator =
17          interruptibleIter.asInstanceOf[Iterator[(K, C)]]
18        dep.aggregator.get.combineCombinersByKey(
19          combinedKeyValuesIterator, context)
20      } else { //只需在 reduce 端聚合
21        val keyValuesIterator =
22          interruptibleIter.asInstanceOf[Iterator[(K, Nothing)]]
23        dep.aggregator.get.combineValuesByKey(keyValuesIterator,
24          ← context)
25      }
26    } else { //不需要聚合
27      interruptibleIter.asInstanceOf[Iterator[Product2[K, C]]]
28    }
29  dep.keyOrdering match { //判断是否需要排序
30    case Some(keyOrd: Ordering[K]) =>
31      //对于需要排序的情况使用 ExternalSorter 进行排序, 注意如果
32      ← spark.shuffle.spill 是 false, 那么数据不会写到磁盘
33      val sorter = new ExternalSorter[K, C, C](context, ordering =
34        ← Some(keyOrd), serializer = Some(serializer))
35      sorter.insertAll(aggregatedIter)
36      .....
37      CompletionIterator[Product2[K, C], Iterator[Product2[K,
38        ← C]]](sorter.iterator, sorter.stop())
39    case None => aggregatedIter
40    //无需排序
41  }
42}

```

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。

上述函数会首先实例化 ShuffleBlockFetcherIterator，实例化时传入了几个参数，下面几个比较重要

- 1 blockManager.shuffleClient
- 2 mapOutputTracker.getMapSizesByExecutorId(handle.shuffleId, startPartition, endPartition)
- 3 SparkEnv.get.conf.getSizeAsMb("spark.reducer.maxSizeInFlight", "48m") * 1024 * 1024

6.3.1 shuffleClient

shuffleClient 就是用来读取其他 executors 上的 shuffle 文件的，其初始化如程序6.11所示

</> 程序 6.11: 创建 shuffleClient </>

```

1 private[spark] val shuffleClient = if (externalShuffleServiceEnabled)
2   {
3     val transConf = SparkTransportConf.fromSparkConf(conf, "shuffle",
4       numUsableCores)
5     new ExternalShuffleClient(transConf, securityManager,
6       securityManager.isAuthenticationEnabled(),
7       securityManager.isSaslEncryptionEnabled())
8   } else {
9     blockTransferService
10 }

```

由上可知，externalShuffleServiceEnabled 默认为 false，所以 shuffleClient 默认为 blockTransferService。而 blockTransferService 在 SparkEnv 初始化的传入，默认为 NettyBlockTransferService，具体细节下一节会讲述。

6.3.2 元数据获取

`mapOutputTracker.getMapSizesByExecutorId` 就是获得该 reduce task 的数据来源(数据的元数据信息), 传入的参数是 shuffle 的 Id 和 partition 的起始位置, 返回的是 `Seq[(BlockManagerId, Seq[(BlockId, Long)])]`, 也就是说数据是来自于哪个节点的哪些 block 的, 并且 block 的数据大小是多少。如程序6.12所示

</> **程序 6.12: 请求元数据** </>

```

1 def getMapSizesByExecutorId(shuffleId: Int, startPartition: Int,
2   ← endPartition: Int)
3 : Seq[(BlockManagerId, Seq[(BlockId, Long)])] = {
4   // 获得 Map 阶段输出的中间计算结果的元数据信息
5   val statuses = getStatuses(shuffleId)
6   // Synchronize on the returned array because, on the driver, it gets
7   ← mutated in place
8   // 将获得的元数据信息转化成形如 Seq[(BlockManagerId, Seq[(BlockId,
9     ← Long)])] 格式的位置信息, 用来读取指定的 Map 阶段产生的数据
10  statuses.synchronized {
11    return MapOutputTracker.convertMapStatuses(shuffleId,
12      ← startPartition, endPartition, statuses)
13  }
14}

```

最后的 `getSizeAsMb` 获取的是一项配置参数, 代表一次从 Map 端获取的最大数据量。上述程序中 `getStatuses` 用来获取元数据信息, 其执行细节如程序6.13所示

</> **程序 6.13: 获得元数据信息** </>

```

1 private def getStatuses(shuffleId: Int): Array[MapStatus] = {
2   // 根据 shuffleId 获得 MapStatus 组成的数组: Array[MapStatus]
3   val statuses = mapStatuses.get(shuffleId).orNull
4   if (statuses == null) {
5     //如果没有获取就进行 fetch 操作
6     val startTime = System.currentTimeMillis
7     // 用来保存 fetch 来的 MapStatus

```

```
8  var fetchedStatuses: Array[MapStatus] = null
9  fetching.synchronized {
10     //其他线程正在请求该 shuffleId 对应的信息，等待其他线程完成
11     while (fetching.contains(shuffleId)) {
12         try {
13             fetching.wait()
14         } catch {
15             case e: InterruptedException =>
16         }
17     }
18     fetchedStatuses = mapStatuses.get(shuffleId).orNull
19     if (fetchedStatuses == null) {
20         fetching += shuffleId
21     }
22 }
23 if (fetchedStatuses == null) {
24     // 如果得到了 fetch 的权利就进行抓取
25     try {
26         // 调用 askTracker 方法发送消息，消息的格式为
27         → GetMapOutputStatuses(shuffleId)
28         val fetchedBytes =
29             → askTracker[Array[Byte]](GetMapOutputStatuses(shuffleId))
30         //MapOutputTrackerMaster 返回元数据信息并逆序列化
31         fetchedStatuses =
32             → MapOutputTracker.deserializeMapStatuses(fetchedBytes)
33         // 保存到本地的 mapStatuses 中
34         mapStatuses.put(shuffleId, fetchedStatuses)
35     } finally {
36         fetching.synchronized {
37             fetching -= shuffleId
38             fetching.notifyAll()
39         }
40     }
41 }
```

```

38    }
39    if (fetchedStatuses != null) {
40        // 最后将抓取到的元数据信息返回
41        return fetchedStatuses
42    } else {
43        }
44    } else {
45        // 如果获取到了 Array[MapStatus] 就直接返回
46        return statuses
47    }
48 }

```

6.3.3 根据元数据信息抓取数据

上节 `ShuffleBlockFetcherIterator` 初始化的过程就完成了元数据信息的获取与转换，下面就是根据这些元数据信息抓取对应的数据。`ShuffleBlockFetcherIterator` 实例化的时候会执行一个 `initialize()` 方法，用来进行一系列的初始化操作，如程序6.14所示

</>	程序 6.14: <code>ShuffleBlockFetcherIterator</code> 初始化	</>
-----	--	-----

```

1 private[this] def initialize(): Unit = {
2     // 不管最后 task 是 success 还是 failure, 都要进行 cleanup 操作
3     context.addTaskCompletionListener(_ => cleanup())
4     // 这里会将本地的数据封装到本地请求块 ArrayBuffer[BlockId], 返回的
5     // 为需要远程请求的块, 数据结构为 ArrayBuffer[FetchRequest]
6     val remoteRequests = splitLocalRemoteBlocks()
7     // 这里的 fetchRequests 是一个队列, 我们将远程的请求以随机的顺序加
8     // 入到该队列, 然后使用下面的
9     // fetchUpToMaxBytes 方法取出队列中的远程请求, 同时对大小进行限制
10    fetchRequests ++= Utils.randomize(remoteRequests)
11    // 从 fetchRequests 取出远程请求, 并使用 sendRequest 方法发送请求
12    fetchUpToMaxBytes()
13    val numFetches = remoteRequests.size - fetchRequests.size

```

```

12 // 获取本地的 Blocks
13 fetchLocalBlocks()
14 }

```

6.3.4 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6.15所示

</> 程序 6.15: reduce 读取数据策略划分 </>

```

1 private[this] def splitLocalRemoteBlocks():ArrayBuffer[FetchRequest]={
2   // 为了将大小控制在 maxBytesInFlight 以下，可以增加并行度，即从 1
3   // → 个节点增加到 5 个
4   val targetRequestSize = math.max(maxBytesInFlight / 5, 1L)
5   val remoteRequests = new ArrayBuffer[FetchRequest]
6   var totalBlocks = 0
7   for ((address, blockInfos) <- blocksByAddress) {
8     totalBlocks += blockInfos.size
9     // 这里就是判断所要获取的是本地的 block 还是远程的 block
10    if (address.executorId == blockManager.blockManagerId.executorId)
11      //本地获取 local
12      //Block 在本地，需要过滤大小为 0 的 Block
13      localBlocks +== blockInfos.filter(_.size != 0).map(_.blockId)
14      numBlocksToFetch += localBlocks.size
15    } else { //远程获取 remote
16      val iterator = blockInfos.iterator
17      var curRequestSize = 0L
18      var curBlocks = new ArrayBuffer[(BlockId, Long)]
19      while (iterator.hasNext) { //blockId 是 ShuffleBlockId
20        val (blockId, size) = iterator.next()
21        if (size > 0) {
22          if (curRequestSize + size > targetRequestSize) {
23            remoteRequests += FetchRequest(blockId, curBlocks)
24            curRequestSize = 0
25            curBlocks = new ArrayBuffer[(BlockId, Long)]
26          }
27          curRequestSize += size
28          curBlocks += (blockId, size)
29        }
30      }
31      if (curRequestSize > 0) {
32        remoteRequests += FetchRequest(blockId, curBlocks)
33      }
34    }
35  }
36  return remoteRequests
37 }

```

```

20         curBlocks += ((blockId, size));remoteBlocks += blockId
21         numBlocksToFetch += 1;curRequestSize += size
22     } else if (size < 0) {
23         throw new BlockException(blockId, "block size " + size)
24     }
25     // 满足大小的限制就构建一个 FetchRequest 并加入到 remoteRequests
26     ↳ 中
27     if (curRequestSize >= targetRequestSize) {
28         remoteRequests += new FetchRequest(address, curBlocks)
29         curBlocks = new ArrayBuffer[(BlockId, Long)]
30         curRequestSize = 0
31     }
32     if (curBlocks.nonEmpty) { //剩余的请求组成一次 Request
33         remoteRequests += new FetchRequest(address, curBlocks)
34     }
35 }
36 }
37 remoteRequests
38 }

```

程序6.15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。

6.3.5 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。这里就相对简单了，进行迭代，如果获取到就将 SuccessFetchResult 保存到 results 中，如果没有就将 FailureFetchResult 保存到 results 中具体过程如程序6.16所示

程序 6.16: 获取本地块

```

</>                                         程序 6.16: 获取本地块                                         </>

1 val iter = localBlocks.iterator
2 while (iter.hasNext) {
3   val blockId = iter.next()
4   try {
5     val buf = blockManager.getBlockData(blockId)
6     shuffleMetrics.incLocalBlocksFetched(1)
7     shuffleMetrics.incLocalBytesRead(buf.size)
8     buf.retain()
9     results.put(new SuccessFetchResult(blockId,
10           blockManager.blockManagerId, 0, buf))
11   } catch {
12     }
13 }

```

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6.17所示

程序 6.17: SortShuffleManager 读取 shuffle 数据实现

```

</>                                         程序 6.17: SortShuffleManager 读取 shuffle 数据实现                                         </>

1 //根据 ShuffleId 和 MapId 获得索引文件
2 val indexFile = getIndexFile(blockId.shuffleId, blockId.mapId)
3 val in = new DataInputStream(new FileInputStream(indexFile))

```

```

4 try {
5   ByteStreams.skipFully(in,blockId.reduceId*8)//本次 Block 的数据区
6   val offset = in.readLong()//开始
7   val nextOffset = in.readLong()//结束
8   new FileSegmentManagedBuffer(transportConf,getDataFile(
9     blockId.shuffleId, blockId.mapId),offset,nextOffset - offset)
10 }

```

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。

6.3.6 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 6.18 所示

程序 6.18：向远程节点发送读取 Block 请求

```

</>                                         程序 6.18：向远程节点发送读取 Block 请求                                         </>
1 private[this] def sendRequest(req: FetchRequest) {
2   req.blocks.size, Utils.bytesToString(req.size),
3   ← req.address.hostPort))
4   bytesInFlight += req.size
5   //首先获得要 fetch 的 blocks 的信息
6   val sizeMap = req.blocks.map { case (blockId, size) =>
7     ← (blockId.toString, size) }.toMap
8   val blockIds = req.blocks.map(_.1.toString)
9   val address = req.address
10  // 然后通过 shuffleClient 的 fetchBlocks 方法来获取对应远程节点上
11  ← 的数据
12  // 默认是通过 NettyBlockTransferService 的 fetchBlocks 方法实现的
13  shuffleClient.fetchBlocks(address.host, address.port,
14    ← address.executorId, blockIds.toArray,
15    new BlockFetchingListener {
16      // 最后，不管成功还是失败，都将结果保存在 results 中
17    }

```

```

13   override def onBlockFetchSuccess(blockId: String, buf:
14     ManagedBuffer): Unit = {//请求成功
15     if (!isZombie) {
16       buf.retain()
17       results.put(new SuccessFetchResult(BlockId(blockId),
18         address, sizeMap(blockId), buf))
19       shuffleMetrics.incRemoteBytesRead(buf.size)
20       shuffleMetrics.incRemoteBlocksFetched(1)
21     }
22   }
23
24   override def onBlockFetchFailure(blockId: String, e: Throwable):
25     Unit = {
26     results.put(new FailureFetchResult(BlockId(blockId), address,
27       e))
28   }
29 }
30 }
```

请求的结果最终保存在了 results 中，成功的就是 SuccessFetchResult，失败的就是 FailureFetchResult

程序中的 shuffleClient 就是 blockTransferService，如程序6.19所示

程序 6.19: BlockManager 获取 shuffleClient

```

</>           程序 6.19: BlockManager 获取 shuffleClient           </>
1 val externalShuffleServiceEnabled =
2   conf.getBoolean("spark.shuffle.service.enabled", false)
3 //externalShuffleServiceEnabled 默认为 false
4 private[spark] val shuffleClient = if (externalShuffleServiceEnabled)
5   {
6   val transConf = SparkTransportConf.fromSparkConf(conf, "shuffle",
7     numUsableCores)
8   new ExternalShuffleClient(transConf, securityManager,
9     securityManager.isAuthenticationEnabled(),
10    securityManager.isSaslEncryptionEnabled())
11 }
```

```
7 } else {  
8     blockTransferService  
9 }  
10 //此部分在 SparkEnv 里定义，直接通过 netty 的方式  
11 val blockTransferService = new NettyBlockTransferService(conf,  
    → securityManager, numUsableCores)
```

6.3.7 BlockStoreShuffleReader.read 方法

上面的步骤获取了数据，接下来的是对这些数据进行处理。具体细节如程序6.10所示。程序6.10中 aggregator 和 keyOrdering 的部分在分析 Shuffle Write 的时候已经分析过了，这里就不再赘述。

6.3.8 Shuffle Read 流程图

整个 shuffle read 的执行过程如图6.7所示

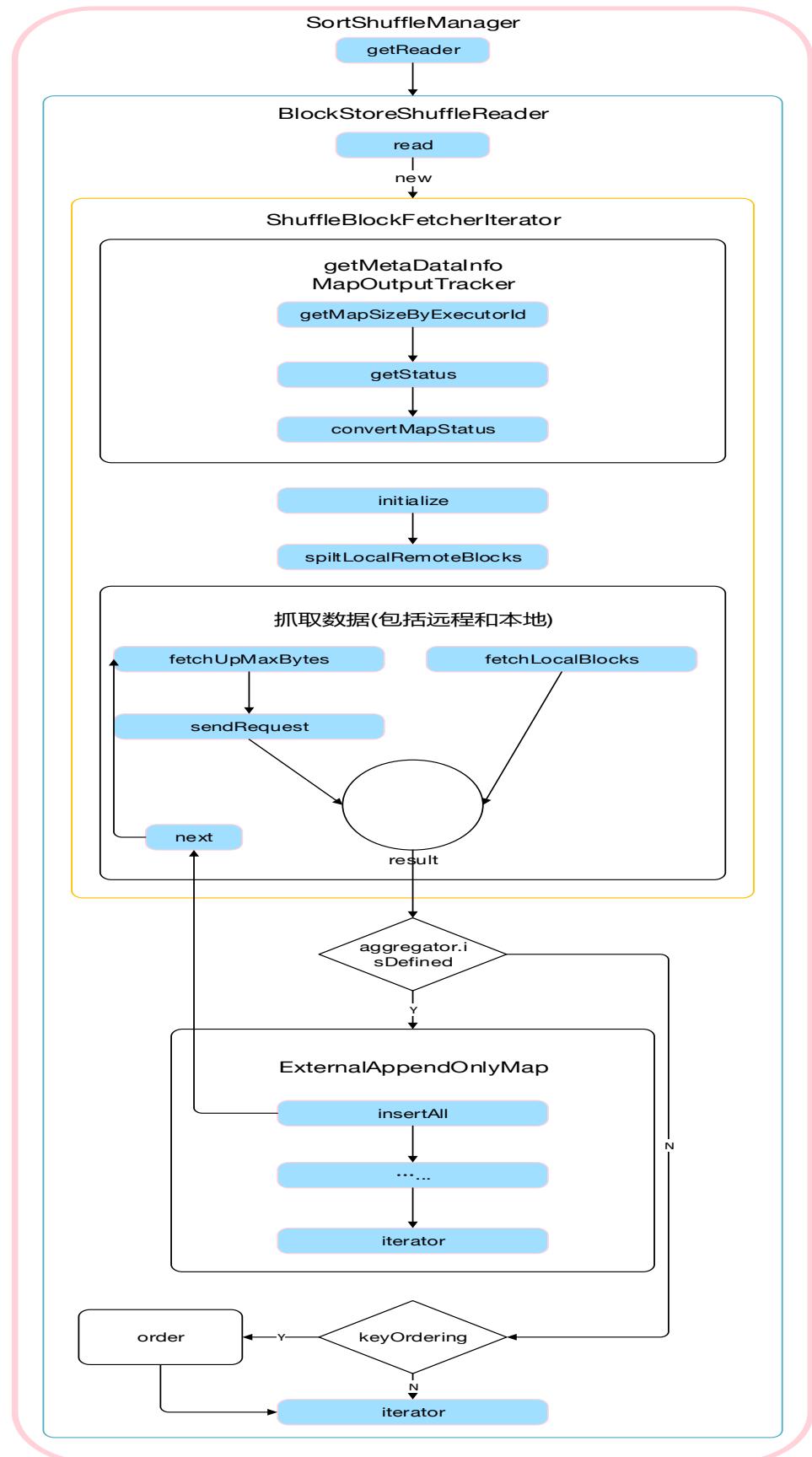


图 6.7 shuffleread 详细实现

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
</>

```

1 //管理所有数据，无论在哪的数据，创建 BlockManager
2 val blockManager = new BlockManager(executorId, rpcEnv,
3     ← blockManagerMaster, serializer, conf, memoryManager,
4     ← mapOutputTracker, shuffleManager, blockTransferService,
5     ← securityManager, numUsableCores)
6
7 //blockTransferService 的初始化和 shuffleClient 的初始化，
8     ← shuffleClient 默认是 NettyBlockTransferService
9
10 blockTransferService.init(this)
11 shuffleClient.init(appId)
12
13 blockManagerId = BlockManagerId(
14     executorId, blockTransferService.hostName, blockTransferService.port)
15
16 //当有外部的 ShuffleService 时，创建新的 BlockManagerId，否则就利用现
17     ← 有的 blockManagerId
18
19 shuffleServerId = if (externalShuffleServiceEnabled) {
20     BlockManagerId(executorId, blockTransferService.hostName,
21         ← externalShuffleServicePort)
22 } else {
23     blockManagerId
24 }
25
26 //向 BlockManagerMaster 注册 blockManagerId slaveEndpoint
27 master.registerBlockManager(blockManagerId, maxMemory,
28     ← slaveEndpoint)
29
30 if (externalShuffleServiceEnabled && !blockManagerId.isDriver) {
31 }
```

```

17    registerWithExternalShuffleServer()
18 }

```

程序中在向 BlockManagerMaster 注册的时候会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

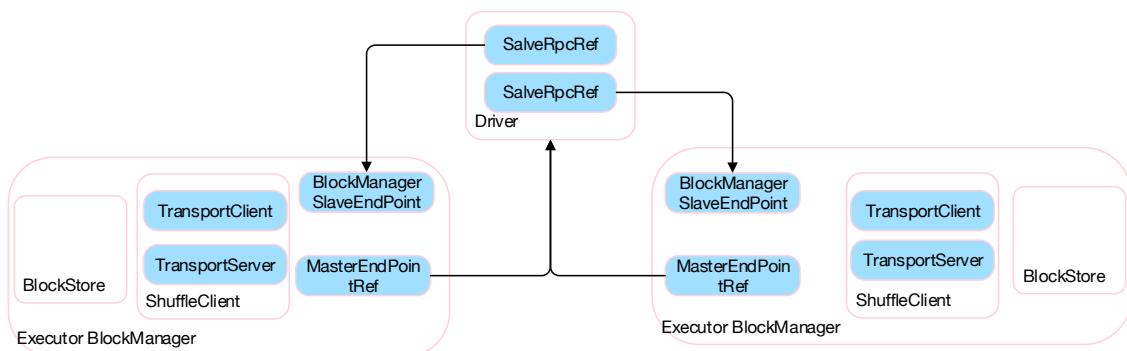


图 7.1 Spark 存储体系架构

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

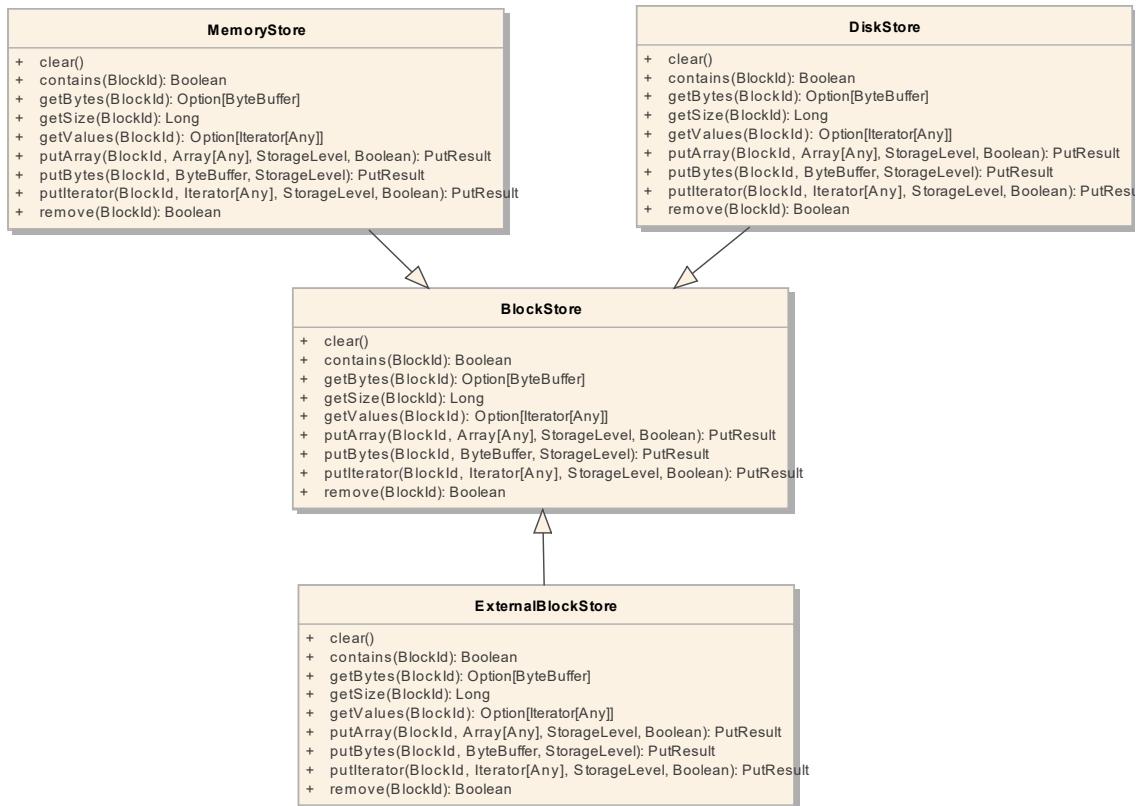


图 7.2 BlockStore 类图结构

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                                         </>


```

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

```


```

```

4  var serverBootstrap: Option[TransportServerBootstrap] = None
5  var clientBootstrap: Option[TransportClientBootstrap] = None
6  if (authEnabled) {
7    serverBootstrap = Some(new SaslServerBootstrap(transportConf,
8      ← securityManager))
9    clientBootstrap = Some(new SaslClientBootstrap(transportConf,
10     ← conf.getAppId, securityManager,
11     securityManager.isSaslEncryptionEnabled()))
12  }
13  //创建 TransportContext
14  transportContext = new TransportContext(transportConf, rpcHandler)
15  //创建 RPC 客户端工厂
16  clientFactory =
17    ← transportContext.createClientFactory(clientBootstrap.toSeq.asJava)
18  //创建 Netty 服务器 TransportServer, 端口由 spark.blockManager.port
19    ← 控制
20  server = createServer(serverBootstrap.toList)
21  appId = conf.getAppId
22}

```

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

```

</>          程序 7.3: NettyBlockRpcServer 核心实现          </>
1 message match {
2   case openBlocks: OpenBlocks =>
3     val blocks:
4       ← Seq[ManagedBuffer]=openBlocks.blockIds.map(BlockId.apply).map(blockManager.get

```

```

4   val streamId = streamManager.registerStream(appId,
5     blocks.iterator.asJava)
6   logTrace(s"Registered streamId $streamId with ${blocks.size}
7     buffers")
8   responseContext.onSuccess(new StreamHandle(streamId,
9     blocks.size).toByteBuffer)
10  case uploadBlock: UploadBlock =>
11    val level: StorageLevel
12      = serializer.newInstance().deserialize(ByteBuffer.wrap(uploadBlock.metadata))
13    val data = new
14      NioManagedBuffer(ByteBuffer.wrap(uploadBlock.blockData))
15    blockManager.putBlockData(BlockId(uploadBlock.blockId), data,
16      level)
17    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
</>

```

1 public TransportContext(
2   TransportConf conf,
3   RpcHandler rpcHandler,
4   boolean closeIdleConnections) {
5   //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
6   //程数量
7   this.conf = conf;
8   //负责 shuffle 的 I/O 服务端在接收到客户端 RPC 请求后，提供打开
9   // Block 或者上传 Block 的 RPC 处理，此处即为 NettyBlockRpcServer
10  this.rpcHandler = rpcHandler;
11  //在 shuffle 的 I/O 客户端对消息进行编码
12  this.encoder = new MessageEncoder();

```

```

11 //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
12 this.decoder = new MessageDecoder();
13 this.closeIdleConnections = closeIdleConnections;
14 }

```

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 </>

```

1 public TransportClientFactory(
2   TransportContext context,
3   List<TransportClientBootstrap> clientBootstraps) {
4     this.context = Preconditions.checkNotNull(context);
5     this.conf = context.getConf();
6     //缓存客户端列表
7     this.clientBootstraps =
8       ↳ Lists.newArrayList(Preconditions.checkNotNull(clientBootstraps));
9     //缓存客户端连接
10    this.connectionPool = new ConcurrentHashMap<SocketAddress,
11      ↳ ClientPool>();
12    //节点之间去数据的连接数，可以使用属性
13    ↳ spark.shuffle.io.numConnectionsPerPeer 来设置，默认为 1
14    this.numConnectionsPerPeer = conf.numConnectionsPerPeer();
15    this.rand = new Random();
16    IOMode ioMode = IOMode.valueOf(conf.ioMode());
17    //客户端 channel 被创建时使用的类，spark.shuffle.io.mode 来配置，默
18    ↳ 认为 NioSocketChannel
19    this.socketChannelClass = NettyUtils.getClientChannelClass(ioMode);
20    // 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 shuffleClient.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     {
9       override def createAndStart(blockIds: Array[String], listener:
  ↳ BlockFetchingListener) {
10         val client = clientFactory.createClient(host, port)

```

```

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

```

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: <i>uploadBlock</i> 实现	
	<pre> 1 override def uploadBlock(2 hostname: String, 3 port: Int, 4 execId: String, 5 blockId: BlockId, 6 blockData: ManagedBuffer, 7 level: StorageLevel): Future[Unit] = { 8 val result = Promise[Unit]() 9 //创建 Netty 服务客户端 10 val client = clientFactory.createClient(hostname, port) 11 //将 Block 的存储级别 StorageLevel 序列化 12 val levelBytes = serializer.newInstance().serialize(level).array() 13 //将 Block 中的 ByteBuffer 转化为数组，便于序列化 14 val nioBuffer = blockData.nioByteBuffer() </pre>	

```

15  val array = if (nioBuffer.hasArray) {
16      nioBuffer.array()
17  } else {
18      val data = new Array[Byte](nioBuffer.remaining())
19      nioBuffer.get(data)
20      data
21  }
22 //调用 Netty 客户端的 sendRpc 方法将字节数组上传，回调
23     ↳ RpcResponseCallback
24 client.sendRpc(new UploadBlock(appId, execId, blockId.toString,
25     ↳ levelBytes, array).toByteBuffer,
26     new RpcResponseCallback {
27         override def onSuccess(response: ByteBuffer): Unit = {
28             result.success(())
29         }
30     }
31     ↳
32     result.future
33 }

```

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*
</>

```

1 def registerBlockManager(
2   blockManagerId: BlockManagerId, maxMemSize: Long, slaveEndpoint:
3     ← RpcEndpointRef): Unit = {
4   logInfo("Trying to register BlockManager")
5   tell(RegisterBlockManager(blockManagerId, maxMemSize,
6     ← slaveEndpoint))
7   logInfo("Registered BlockManager")
8 }
```

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	
--	-----------------------------------	--

```

1 private def register(id: BlockManagerId, maxMemSize: Long,
2   ← slaveEndpoint: RpcEndpointRef) {
3   val time = System.currentTimeMillis()
4   //确保 blockManagerInfo 持有消息中的 BlockManagerId 及其对应信息
5   if (!blockManagerInfo.contains(id)) {
6     //确保一个 Executor 只有一个 BlockManagerId
7     blockManagerIdByExecutor.get(id.executorId) match {
8       //旧的 BlockManagerId 会被移除
9       case Some(oldId) =>
10         removeExecutor(id.executorId)
11       case None =>
12     }
13     blockManagerIdByExecutor(id.executorId) = id
14     blockManagerInfo(id) = new BlockManagerInfo(id,
15       ← System.currentTimeMillis(), maxMemSize, slaveEndpoint)
16     listenerBus.post(SparkListenerBlockManagerAdded(time, id,
17       ← maxMemSize))
18   }

```

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

</>

```

1 def getFile(filename: String): File = {
2   //根据文件名 blockId.name 计算哈希值
3   val hash = Utils.nonNegativeHash(filename)
4   //根据哈希值与本地文件一级目录的总数求余数，记为 dirId
5   val dirId = hash % localDirs.length
6   //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
7   //    ↳ 求余数，记为 subDirId
8   val subDirId = (hash / localDirs.length) % subDirsPerLocalDir
9   //如果 dirId/subDirId 目录存在，则获取 dirId/subDirId 目录下的文件
10  val subDir = subDirs(dirId).synchronized {
11    val old = subDirs(dirId)(subDirId)
12    if (old != null) {
13      old
14    } else {
15      //否则新建 dirId/subDirId 目录
16      val newDir = new File(localDirs(dirId), "%02x".format(subDirId))
17      if (!newDir.exists() && !newDir.mkdir()) {
18        throw new IOException(s"Failed to create local dir in
19          ↳ \$newDir.")
20      }
21      subDirs(dirId)(subDirId) = newDir
22      newDir
23    }
24  }
25  new File(subDir, filename)
26}

```

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

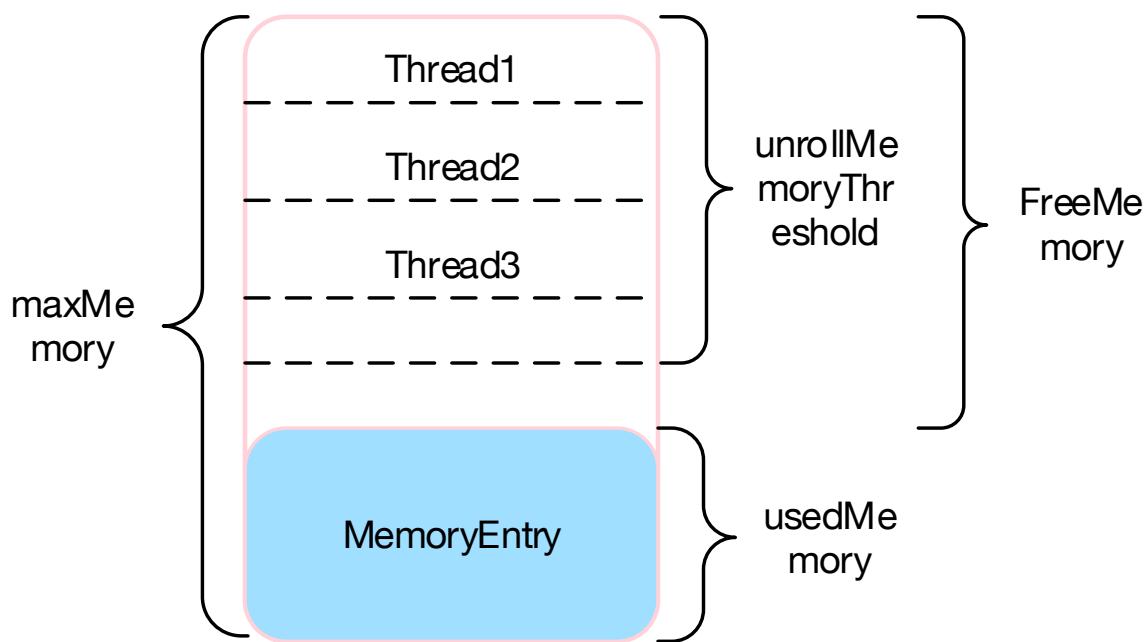


图 7.3 MemoryStore 的内存模型

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 没有足够的空间时，就会使用 DiskStore 将块存入磁盘。