Spark 实践学习 - 准备

1. Spark 开发环境配置

- Intellij IDEA+Scala Plugin
- 下载 IDEA 开发 IDE 工具，在 File->settings->Plugins->Browser Repositories 中搜索 Scala 安装

2. 知识预备

- RDD（数据项集合）
- Dependency（连接 RDD ）
- Transformation（ RDD 变换）
- Action（生成并执行一个 Job ）

3. 实践程序

- 编写 Spark 程序

4. Spark 程序分析

- RDD 与 Dependence 生成
- Job 生成
- Stage 划分与执行

***Task 层面与 BlockManager（管理具体数据在集群内传输变换）暂时不做讨论

Spark 实践学习 - Sbt, Scala, Java

Sbt 是项目管理工具， Maven ， Gradle 都是协助程序员管理项目开发的工具。

*** 项目管理概念：

此处指对于开发项目源代码的管理，类、函数库依赖的合理组织，项目打包生成可执行 Jar 包或者 War 包，再到发布到服务器端真正实现产品功能的整个过程，称之为项目管理。

Scala 与 Sbt 的关系

Sbt 是由 Scala 编写的项目管理工具，专门为管理 Scala 语言编写的项目开发。

Scala 与 Java 联系

都是基于 JVM 开发的编程语言， Scala 能够与 Java 极好地兼容，互相调用函数接口，只需要做简单的适配即可。所以 Scala 其实是在 Java 语言特性的基础上增加更多的函数式编程特性，同时具有函数式编程与面向对象编程特性的编程语言。

一些概念：

JVM 执行字节码，转成机器可执行的机器码， Java 与 Scala 以及 Groovy 等基于 JVM 规范建立的编程语言，都可以生成 .class 文件最终在 JVM 基础上运行。

《深入理解 Java 虚拟机》、《 Thinking in Java 》第 14 章节类型信息，对于理解 Java 语言设计较有帮助

《 Scala Cookbook 》阅读

Spark 实践学习 - RDD

建立如下实验代码， Debug 观察 RDD

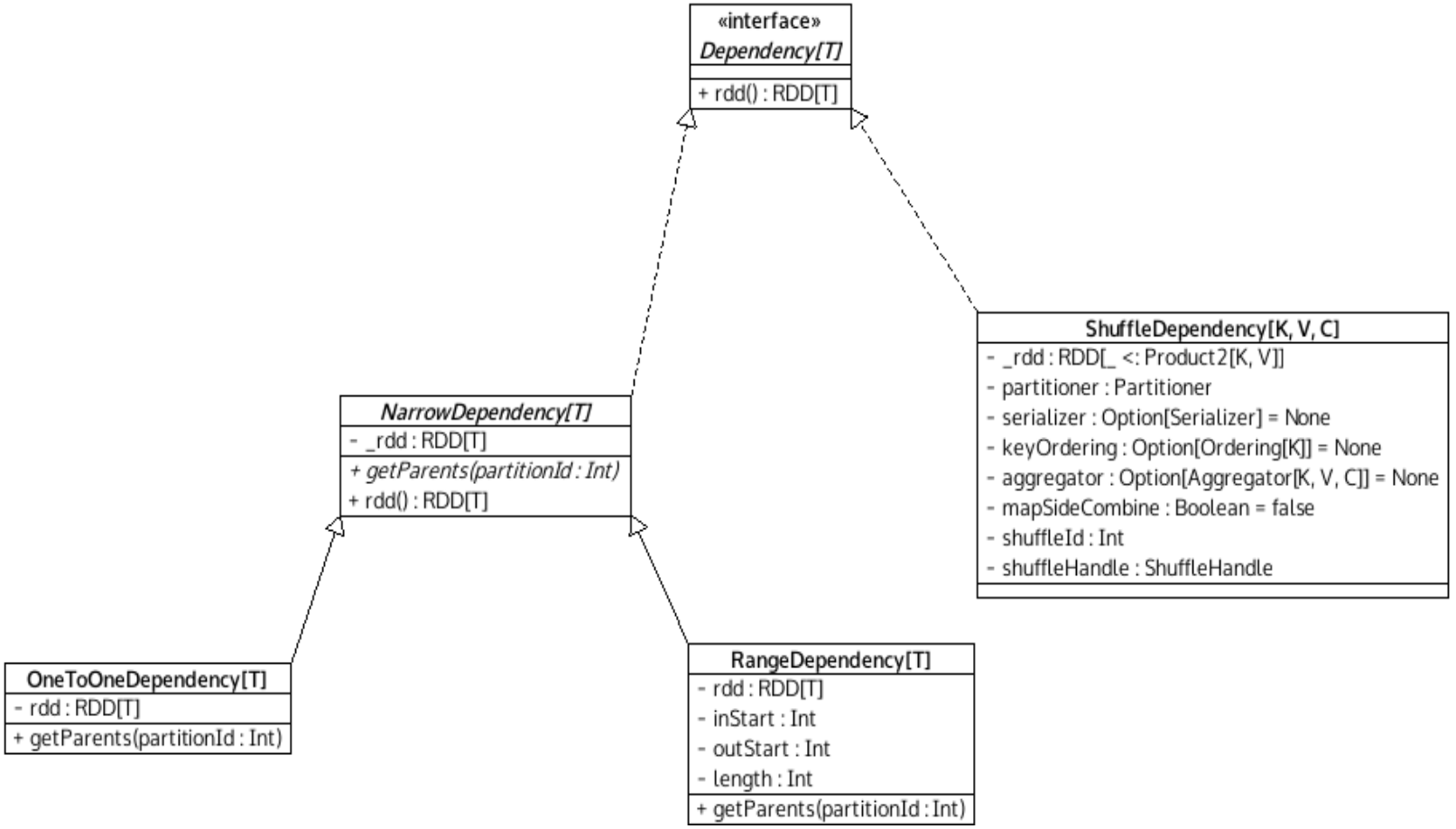
```
object ShowRDD {
  def printRDD(sc: SparkContext, data: Seq[Int], numPartition: Int = 4) = {
    val rdd = sc.parallelize(data, numPartition)
    val rdd1 = rdd.filter(_ % 3 != 0)
    rdd.foreachPartition(_.foreach(println))
    val result = rdd.collect()
    result.foreach(println)
  }

  def main (args: Array[String]) {
    val conf = new SparkConf().setAppName("Complex Job").setMaster("local[*]")
    val sc = new SparkContext(conf)
    printRDD(sc, Range(0, 10))
    sc.stop()
  }
}
```

RDD 个人理解：

起始的第一个 RDD 拥有外部数据接口，且无 Dependency ，保存初始需要处理的数据内容，而中间过程 RDD 则有几种不同特性，要么是最简单的 OneToOneDependency 或者 RangeDependency（均属于 NarrowDependency ），或者是复杂的 ShuffleDependency

学习参考资料： https://github.com/JerryLead/SparkInternals/tree/master/markdown

Spark 实践学习 - Dependency



Dependency 代表 RDD 之间的联系，每次均通过具体实现的 RDD 自行计算（ override def getDependencies: Seq[Dependency[_]] ）

在具体如何实现自己的 RDD 类型时，只需要实现 Partition 内数据的计算方式（ def compute(split: Partition, context: TaskContext): Iterator[T] ），如何划分 Partition （ def getPartitions: Array[Partition] ），以及如何计算 Dependency （ def getDependencies: Seq[Dependency[_]] ）

Spark 实践学习 - Transformation

所有的 Transformation 都是为了解决 RDD 与 parent RDD 之间的关系：

RDD 依赖一个或两个 parent RDD

RDD 的 partition 由 parent RDD 的 partition 计算而来，如何计算？根据之前提到的三个方法。

如果要自己设计一个 RDD ，那么需要注意的是 compute() 只负责定义 parent RDDs => output records 的计算逻辑，具体依赖哪些 parent RDDs 由 getDependency() 定义，具体依赖 parent RDD 中的哪些 partitions 由 dependency.getParents() 定义。

Spark 实践学习 - Action

一个 Action 生成一个 Job ，并将这个 Job 提交给 Spark 集群计算。

整个计算的过程划分为两步：第一步是每个 Partition 计算自己本身的结果，第二步是将所有 Partition 的结果汇总到 Driver 节点上计算最终结果。

一个 Job 在提交之后划分为 Stages ，此处根据 ShuffleDependency 节点划分

一个 Stage 内部必然是 NarrowDependency ，所以 Task 可以根据本节点上的 Partition 内数据自行完成计算， TaskSet 为当前 Stage 内划分的 Task（与 Partition 个数相同，流水线逻辑）集合，每个 TaskSet 由 TaskSetManager 进行包装管理，包括这组 TaskSet 内 Task 的开始计算，计算结束。 TaskSet 内的 Task 若全部完成计算，则将当前 Stage 设置 已经计算完成。

一个 Stage 开始计算的前提是依赖的所有 Stage 都已经计算完成，所以从最左边开始计算 Stages 划分图（ DAG Diagram ）

Spark 实践学习 - Spark 程序

```
object CompJob {

  def main (args: Array[String]) {
    val conf = new SparkConf().setAppName("Complex Job").setMaster("local[*]")
    val sc = new SparkContext(conf)        // 启动 Spark 容器环境

    val data0 = sc.parallelize((0 until 18).map((i: Int) => (i, (i+'a').toChar)), 3)          //  从 Collections 中读取数据至 RDD 内
    val data1 = sc.parallelize((0 until 18).map(i => (i, (i+'A').toChar)))
    val data2 = data0.filter(_._1 % 3 != 0).repartition(2)
    val data3 = data1.repartition(6)
    val data4 = data2.union(data3)
    val data5 = data4.filter(_._1 % 4 != 0)
    val result = data5.groupByKey().filter(_._1 % 5 != 0)

    val r1 = result.count()     //  输出 records 的数量
    val r2 = result.collect()    //  将 records 返回 Array
    println(r1)
    r2.foreach(println)

    sc.stop()
  }
}
```

Spark 实践学习 - SparkContext

SparkContext 在整个 Spark 的运行环境中具有重要作用

初始化定义 Spark 全局事件监听， listenerBus ，包含多个监听器，用 ArrayList 保存，一个 blockqueue 队列维护所有收到的 Spark 相关事件，包括 Job 添加、运行、结束， Executor 添加或丢失等等，所有监听器都会去处理接收到的事件。

SparkEnv ，初始化 Spark 运行的基础环境，包括通信安全机制，节点各自的 ActorSystem 和端口（便于提供给远程程序发送信息），序列化工具， MapOutputTrackerMaster 与 MapOutputTrackerWorker（管理控制各节点上的 shuffle 输出结果）， ShuffleManager（目前为 SortShuffleManager ，涉及到底层文件读写管理，与 BlockManager 配合）， blockTransferService（数据块传输，基本上等于一个 Partition 数据内容）， BlockManager（数据块传输读写管理）， BroadcastManager（全局共享变量，其实也可以拿来作为数据传输工具）， CacheManager（缓存管理）， HttpFileServer（文件传输）， MetricsSystem（信息采集模块，设置 Sources 和 Sinks ）

ui ， WebService 的提供者， HeartbeatReceiver ，心跳机制

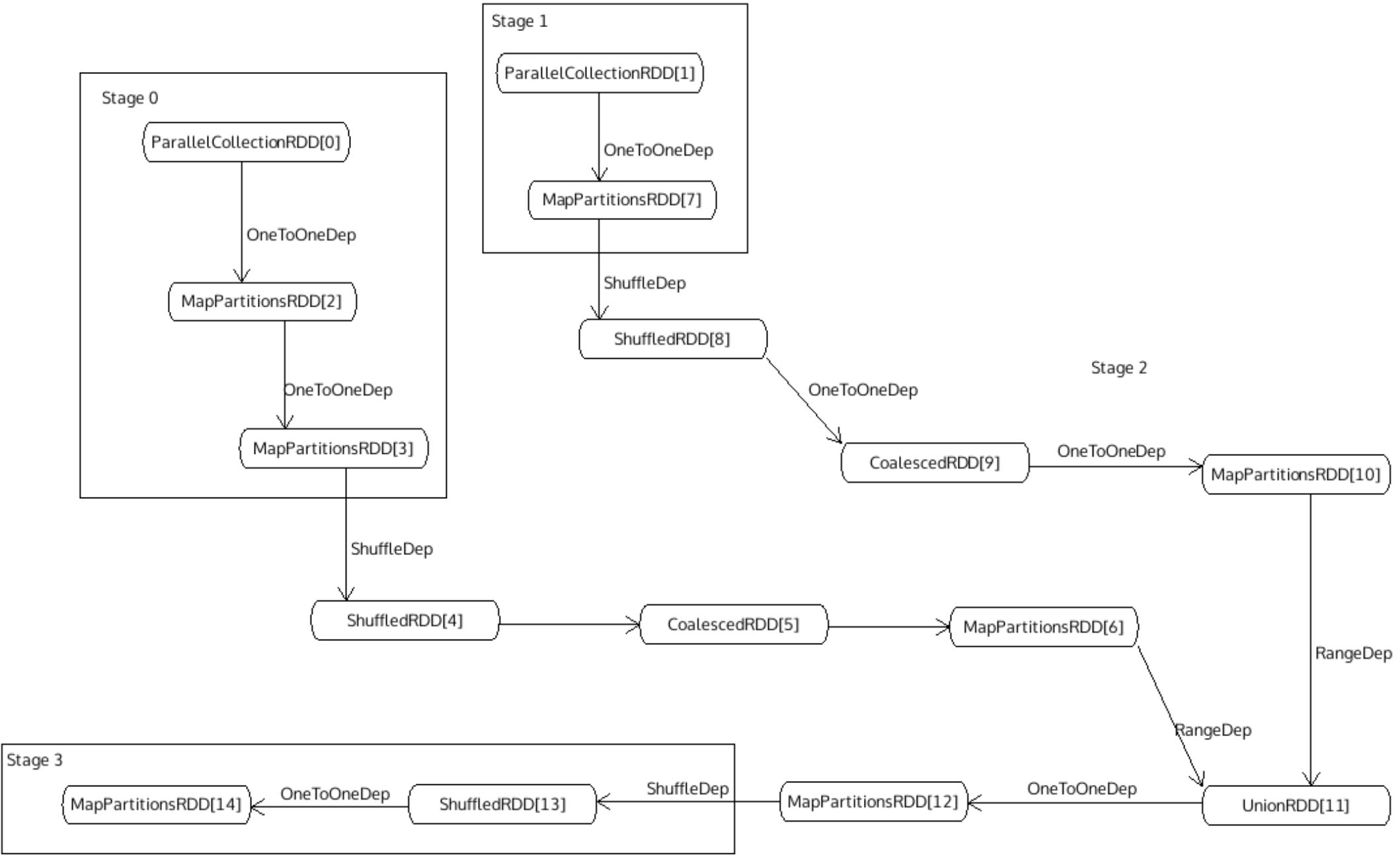
DAGScheduler ， TaskSchedulerImpl ， backend（ LocalBackend ， SparkDeploySchedulerBackend ）

所有的 Job 都是通过 DAGScheduler 进行处理，提交到 DAGScheduler 并等待执行结果。

DAGScheduler 本身有一个事件线程 eventProcessLoop 在创建 DAGScheduler 时启动， eventProcessLoop 接收 DAGSchedulerEvent ，根据接收到的事件处理相应功能，所以此处接收到 JobSubmitted 后，调用 handleJobSubmitted() 函数，进入 Stage 划分阶段。

backend 是通过 Actor 消息机制，辅助 TaskSchedulerImpl 实现任务调度管理的

Spark 实践学习 - DAG Diagram



Spark 实践学习 - Spark 程序

Job 运行总体流程：

- SparkContext 调用 DAGScheduler 的 runJob(rdd, cleanedFunc, partitions, allowLocal, resultHandler) 来提交 job
- DAGScheduler 的 runJob 继续调用 submitJob(rdd, func, partitions, allowLocal, resultHandler) 来提交 job
- submitJob() 首先得到一个 jobId ，向 DAGSchedulerEventProcessActor 发送 JobSubmitted 信息，该 actor 收到信息后进一步调用 dagScheduler.handleJobSubmitted() 来处理提交的 job。（事件驱动）
- handleJobSubmmitted() 首先调用 finalStage = newStage() 来划分 stage ，然后 submitStage(finalStage)。由于 finalStage 可能有 parent stages ，实际先提交 parent stages ，等到他们执行完， finalStage 需要再次提交执行。

Stage 划分算法（递归，转成循环实现）：

主要逻辑：

最后一个提交 Job 的 Stage 比较特殊，先划分出该 Job 的最后一个 Stage ，然后处理前面的 parent Stages ，不同 Job 可能使用共同的 Stage ，所以为了避免重复计算使用之前计算的 MapOutPut 结果覆盖当前 Stage。

从最后一个 RDD 开始 newStage ，每个 Stage 属于一个或多个 Job

获取 parent Stages

循环向前搜索到第一层 ShuffleDependency ， getShuffleMapStage ，获取当前 ShuffleDependency 前的 Stage DAG Diagram

判断接下来第一个 Stage 是否已经在 shuffleToMapStage 中注册

如果有直接返回注册的 Stage

否则 { 计算该 Stage 之前的 DAG Diagram ， registerShuffleDependencies()

首先获取当前 Stage 前所有的 ShuffleDependency ， getAncestorShuffleDependencies()

循环处理 { 根据每一个获得的 ShuffleDependency ， newOrUsedStage() ， Stage 都是会生成的，区别在于是否有已经计算过的数据， newOrUsedStage 便是用之前已经计算获得的结果覆盖当前新生成 Stage 的结果。在 shuffleToMapStage 注册 Stage }

newOrUsedStage() ，在 shuffleToMapStage 注册 Stage ，返回结果 Stage }

返回所有的 parent RDDs

新建最后一个调用 Job Action 的 Stage ，在 shuffleToMapStage 注册 Stage ，返回 finalStage

Over