Spark 源码分析

王联辉 lianhuiwang09@gmail.com

参考资料

http://www.scala-lang.org/
http://akka.io/
https://github.com/akka/akka
http://code.alibabatech.com/blog/dev_related_1279/simple-introduction-on-akka-actor.html
https://github.com/mesos/spark
http://rdc.taobao.com/team/jm/archives/2043
http://yanbohappy.sinaapp.com/?p=16
http://jerryshao.me/Architecture/2013/04/21/Spark%E6%BA%90%E7%A0%81%E5%88%86%E6%9E%90%E4%B9%8B-scheduler%E6%A8%A1%E5%9D%97/
http://jerryshao.me/Architecture/2013/04/30/Spark%E6%BA%90%E7%A0%81%E5%88%86%E6%9E%90%E4%B9%8B-deploy%E6%A8%A1%E5%9D%97/

名词解释

RDD
全称为 Resilient Distributed Dataset，弹性分布式数据集。
就是分布在集群节点上的数据集，这些集合可以用来进行各种操作。最重要的一点是，某个操作计算后的数据集可以缓存在内存中，然后给其他计算使用，这种在迭代计算中很常见。比如：我们可以从 hdfs 文件里创建一个数据集，然后经过 filter 后，会生成一个新的数据集，还可以进行 group by,map 函数等操作，得到另一个数据集。

`Iterator`
迭代器，即是提供一级接口给其他人来访问 RDD 中的数据集。
`Job`
一道作业，即指应用完成某项需求所需要一系列工作，统一称为作业。
DAG
有向无环图，将作业分解成若干个阶段，每个阶段都是由若干个 `task` 组成，而这些阶段都是有先后顺序的，故将这些阶段组织成 DAG，来表示其先后顺序。
Stage
阶段，是指 `job` 中的一个结点。
Taskset

每个阶段将由若干个 task 组成，这些 task 统一称为 taskset。
Task
Task 是指最终在 slave 结点上运行的工作。

与 MR 的区别

MR 的缺点：
1. Shuffle 的性能。Map 到 reduce 之间数据多次需要 IO 操作。
2. 当有多个 MR 时，每轮的 MR 之间需要将结果写到 hdfs 上。
3. 只有 map,reduce 二种计算模型，无法建立一组 DAG 操作，来减少中间的一些操作开销。

以上的缺点都是 spark 的优点。

相关知识

Scala

Scala 是运行在 JVM 之上的编程语言。集成了面向对象和函数式语言的特点，代码量比 java 要少 2-3 倍。

Akka

是轻量级异步事件处理的消息系统。
http://code.alibabatech.com/blog/dev_related_1279/simple-introduction-on-akka-actor.html

运行实例

SparkLR

```
  val sc = new SparkContext(args(0), "SparkLR",
      System.getenv("SPARK_HOME"), Seq(System.getenv("SPARK_EXAMPLES_JAR")))
```

在 SparkContext 的构造函数，根据系统的配置信息文件，创建一个运行环境对象 SparkEnv 给后面的应用程序使用，主要的部件有以下这些

```
class SparkEnv (
```

```
    val executorId: String,
    val actorSystem: ActorSystem,
    val serializer: Serializer,
    val closureSerializer: Serializer,
    val cacheManager: CacheManager,
    val mapOutputTracker: MapOutputTracker,
    val shuffleFetcher: ShuffleFetcher,
    val broadcastManager: BroadcastManager,
    val blockManager: BlockManager,
    val connectionManager: ConnectionManager,
    val httpFileServer: HttpFileServer,
    val sparkFilesDir: String
  )
```

然后调用 sc.parallelize(generateData, numSlices).cache()，得到一个 scala 集合的 RDD 对象。

```
  for (i <- 1 to ITERATIONS) {
    println("On iteration " + i)
    val gradient = points.map { p =>
      (1 / (1 + exp(-p.y * (w dot p.x))) - 1) * p.y * p.x
    }.reduce(_ + _)
    w -= gradient
  }
```

再经过 map,reduce 函数后，最后得到一个结果，再与 w 进行-=计算。

其中 map()函数后得到 MappedRDD 对象，而 reduce 函数的逻辑如下：

```
def reduce(f: (T, T) => T): T = {
    val cleanF = sc.clean(f)
    val reducePartition: Iterator[T] => Option[T] = iter => {
      if (iter.hasNext) {
        Some(iter.reduceLeft(cleanF))
      } else {
        None
      }
    }
    var jobResult: Option[T] = None
    val mergeResult = (index: Int, taskResult: Option[T]) => {
      if (taskResult != None) {
        jobResult = jobResult match {
          case Some(value) => Some(f(value, taskResult.get))
          case None => taskResult
        }
      }
    }
    sc.runJob(this, reducePartition, mergeResult)
```

```
    // Get the final result out of our Option, or throw an exception if the RDD
was empty
    jobResult.getOrElse(throw new UnsupportedOperationException("empty
collection"))
  }
```

最后通过调用 SparkContext 的 runJob()方法，进行对 RDD 计算，得到最终结果。

GroupByTest

实例作用：
调用 parallelize()得到 ParallelCollectionRDD，调用 flatMap()得到 FlatMappedRDD，调用 cache()对前面的 RDD 进行 persist 工作，即调用了 SparkContext.persistentRdds()将此 RDD 保存到 TimeStampedHashMap 内存数据结构中。

```
val sc = new SparkContext(args(0), "GroupBy Test",
      System.getenv("SPARK_HOME"), Seq(System.getenv("SPARK_EXAMPLES_JAR")))

    val pairs1 = sc.parallelize(0 until numMappers, numMappers).flatMap { p =>
      val ranGen = new Random
      var arr1 = new Array[(Int, Array[Byte])](numKVPairs)
      for (i <- 0 until numKVPairs) {
        val byteArr = new Array[Byte](valSize)
        ranGen.nextBytes(byteArr)
        arr1(i) = (ranGen.nextInt(Int.MaxValue), byteArr)
      }
      arr1
    }.cache
    // Enforce that everything has been calculated and in cache
    pairs1.count

    println(pairs1.groupByKey(numReducers).count)
```

RRD.count 的工作原理：
val results = new Array[U](partitions.size)
首先会创建一个 Array，大小为 partitions 的大小。
然后运行 runJob()，将最终的各个 partitions 的结果放到这个 results 中。
再调用 sum，得到最终的纪录总数。

```
  def count(): Long = {
    sc.runJob(this, (iter: Iterator[T]) => {
      var result = 0L
      while (iter.hasNext) {
        result += 1L
```

```
        iter.next()
      }
      result
    }).sum
  }
```

那么接下来需要关注的是 runJob()，如何将每个 partition 的最终结果存放到 results 的相应位置，详细见 SparkContext 类中以下方法：

```
  def runJob[T, U: ClassManifest](
      rdd: RDD[T],
      func: (TaskContext, Iterator[T]) => U,
      partitions: Seq[Int],
      allowLocal: Boolean,
      resultHandler: (Int, U) => Unit) {
    val callSite = Utils.getSparkCallSite
    logInfo("Starting job: " + callSite)
    val start = System.nanoTime
    val result = dagScheduler.runJob(rdd, func, partitions, callSite,
allowLocal, resultHandler)
    logInfo("Job finished: " + callSite + ", took " + (System.nanoTime - start)
/ 1e9 + " s")
    rdd.doCheckpoint()
    result
  }
```

这里主要是调用了 DAGScheduler. runJob 方法，详细如下：

```
  def runJob[T, U: ClassManifest](
      finalRdd: RDD[T],
      func: (TaskContext, Iterator[T]) => U,
      partitions: Seq[Int],
      callSite: String,
      allowLocal: Boolean,
      resultHandler: (Int, U) => Unit)
  {
    if (partitions.size == 0) {
      return
    }
    val (toSubmit, waiter) = prepareJob(
        finalRdd, func, partitions, callSite, allowLocal, resultHandler)
    eventQueue.put(toSubmit)
    waiter.awaitResult() match {
      case JobSucceeded => {}
      case JobFailed(exception: Exception) =>
        logInfo("Failed to run " + callSite)
        throw exception
    }
```

```
  }
```

那么这里主要是 prepareJob 方法以及 DAG 调度里的事件处理。

先看 prepareJob 方法，主要是创建一个 JobSubmitted 和 JobWaiter 对象，然后将 JobSubmitted 放到 DAG 的事务队列，利用 JobWaiter 等待整个任务的执行完成。

```
  private[scheduler] def prepareJob[T, U: ClassManifest](
      finalRdd: RDD[T],
      func: (TaskContext, Iterator[T]) => U,
      partitions: Seq[Int],
      callSite: String,
      allowLocal: Boolean,
      resultHandler: (Int, U) => Unit)
    : (JobSubmitted, JobWaiter[U]) =
  {
    assert(partitions.size > 0)
    val waiter = new JobWaiter(partitions.size, resultHandler)
    val func2 = func.asInstanceOf[(TaskContext, Iterator[_]) => _]
    val toSubmit = JobSubmitted(finalRdd, func2, partitions.toArray, allowLocal,
callSite, waiter)
    return (toSubmit, waiter)
  }
```

DAGScheduler 中有个常驻方法，即 run()方法，里面主要是 processEvent 方法，用来对事务队列中的事件进行处理，其中就有 JobSubmitted 事件，具体 job 是如何在集群中运行的，请参见 DAGScheduler 部分。

那么当 job 执行完后，会向 DAGScheduler 中的 eventQueue 插入 CompletionEvent，然后事务处理方法，会调用 handleTaskCompletion 方法。

如果 job 执行成功的话，则会调用 job.listener.taskSucceeded(rt.outputId, event.result)。

而这个 listener 正是之前创建的 JobWaiter 对象，然后将这个 task 的 result 保存到之前创建好的 results Array 中。

需要返回客户端结果的 task 称为 ResultTask,这个结果首先由 ResultTask 在各个节点的 taskRunner 上运行，并且结果序列化后打包到 TaskResult，返回给 TaskSetManager 中的 statusUpdate 方法调用上，最终返回到 JobWaiter 上面。

共享变量

累计变量

跟 hadoop 里的 counter 含义是一样的，在 spark 里面叫做 Accumulator(累加器)

首先在 rdd 运行前定义一个初始化值为 0 的 Accumulator 变量 accum。如下所示：

```
val accum = sc.accumulator(0)
```

然后在你的 RDD 操作函数过程中就可以直接用+操作符进行累加。如下所示：

```
sc.parallelize(Array(1, 2, 3, 4)).foreach(x => accum += x)
```

在本次 app 运行完成后，你可以通过调用 Accumulator 变量 accum 的 value 元素来得到最后

的值，如下所示：
accum.value

广播变量

广播变量允许使用者指定某个变量，将其缓存到每一台运行其相应 task 的机器上，它的存储方式为 memory_and_disk，即先考虑内存，再考虑磁盘。目前 spark 支持的广播方式有 Http 协议和 Torrent 协议，默认是 Http 协议。
先通过 broadcast 函数定义一个广播变量 broadcastVar，如下所示：
val broadcastVar = sc.broadcast(Array(1, 2, 3))
然后再 RDD 的操作函数里可以直接通过调用广播变量 broadcastVar 的 value 元素来访问广播值。如下所示：
scala> broadcastVar.value
res0: Array[Int] = Array(1, 2, 3)

SparkContext

Spark 的上下文对象，用来与 spark cluster 进行连接，以及创建 RDD 以及保存与广播相关的变量信息。
主要有
SparkEnv：spark 应用需要的运行时环境
BlockManagerUI:　block manager 的 web http 服务。
addedFiles，addedJars：　应用需要用到的文件集合
persistentRdds: 所有的 RDD
MetadataCleaner: 周期删除旧 metadata 的常驻线程
TaskScheduler: task 的调度器
DAGScheduler: stage 的 DAG 调度器
hadoopConfiguration:　hadoop 配制信息

该类重要的方法：
def parallelize[T: ClassManifest](seq: Seq[T], numSlices: Int = defaultParallelism): RDD[T]
创建一个本地的 scala 对象集合的 RDD 对象。

textFile(path: String, minSplits: Int = defaultMinSplits): RDD[String]
得到一个 hadoop 上的 text 的 RDD 对象，相应的还有 seq file 等方法。

```
def runJob[T, U: ClassManifest](
        rdd: RDD[T],
        func: (TaskContext, Iterator[T]) => U,
        partitions: Seq[Int],
        allowLocal: Boolean,
        resultHandler: (Int, U) => Unit)
```

执行 RDD 计算。

SparkEnv

```
class SparkEnv (
    val executorId: String,
    val actorSystem: ActorSystem,
    val serializer: Serializer,
    val closureSerializer: Serializer,
    val cacheManager: CacheManager,
    val mapOutputTracker: MapOutputTracker,
    val shuffleFetcher: ShuffleFetcher,
    val broadcastManager: BroadcastManager,
    val blockManager: BlockManager,
    val connectionManager: ConnectionManager,
    val httpFileServer: HttpFileServer,
    val sparkFilesDir: String
  )
```

RDD

RDD 的起始

RDD 是一个有容错机制且可以被并行操作的元素集合。有二种类型：

一种是并行集合(parallelized collections):接收一个已经存在的 scala 集合，然后进行各种并行计算。如：Val distData=sc.parallelize(Array(1,2,3,4,5))

即创建了一个 Int 数组的 RDD 集合。一旦分布式数据集（distData）被创建好，它们将可以被并行操作。例如，我们可以调用 distData.reduce(_ +__)来将数组的元素相加。

另一种是 Hadoop 数据集：是从 hadoop 上的文件系统创建的 RDD。可以支持 textfile，sequence file 以及其他任何的 hadoop 输入格式。

如：val distFile=sc.textFile("hdfs://data.txt")

一旦创建完成，distFile 可以被进行数据集操作。例如，我们可以通过使用如下的 map 和 reduce 操作：distFile.map(_.size).reduce(_ + _)将所有数据行的长度相加。

一般情况下，我们用到后者比较多，即 Hadoop 数据集类型。如果有多个文件或目录，用‘,’逗号连接起来即可，如：sc.textFile("hdfs://data.txt,hdfs://data1.txt")。本地文件使用 file://开始，再加上目录名称，如：sc.textFile("file:///data/")。

RDD 的组成

一个 RDD 一般会有以下四个函数组成。
物理执行的计算过程
def compute(split: Partition, context: TaskContext): Iterator[T]

分片信息
protected def getPartitions: Array[Partition]

protected def getDependencies: Seq[Dependency[_]] = deps
Deps 是由 RDD 的构造函数传进来的。
```
  def this(@transient oneParent: RDD[_]) =
    this(oneParent.context , List(new OneToOneDependency(oneParent)))
```
因此默认就是与父 RDD 是 Narrow Dependency 依赖。

一些关于如何分块和数据所放位置的元信息
protected def getPreferredLocations(split: Partition): Seq[String] = Nil

RDD 的类型

RDD 支持两种操作：**转换（transformation）从现有的数据集创建一个新的数据集；而动作（actions）在数据集上运行计算后，返回一个值给驱动程序。** 例如，map 就是一种转换，它将数据集每一个元素都传递给函数，并返回一个新的分布数据集表示结果。另一方面，count()是一种动作，统计数据集所有的元素的个数，并将最终结果返回给 Driver 程序。

Spark 中的所有转换都是惰性的，也就是说，他们并不会直接计算结果。相反的，它们只是记住应用到基础数据集（例如一个文件）上的这些转换动作。只有当发生一个要求返回结果给 Driver 的动作时，这些转换才会真正运行。这个设计让 Spark 更加有效率的运行。例如，我们可以实现：通过 map 创建的一个新数据集，并在 reduce 中使用，最终只返回 reduce 的结果给 driver，而不是整个大的新数据集。

默认情况下，**每一个转换过的 RDD 都会在你在它之上执行一个动作时被重新计算。不过，你也可以使用 persist(或者 cache)方法，持久化一个 RDD 在内存中。**在这种情况下，Spark 将会在集群中，保存相关元素，下次你查询这个 RDD 时，它将能更快速访问。在磁盘上持久化数据集，或在集群间复制数据集也是支持的，这些选项将在本文档的下一节进行描述。

下面的表格列出了目前所支持的转换和动作（详情请参见 RDD API doc：

http://spark.apache.org/docs/0.9.1/api/core/index.html#org.apache.spark.rdd.RDD）：

转换（transformation）

转换	含义
map(func)	返回一个新分布式数据集，由每一个输入元素经过 func 函数转换后组成
filter(func)	返回一个新数据集，由经过 func 函数计算后返回值为 true 的输入元素组成
flatMap(func)	类似于 map，但是每一个输入元素可以被映射为0或多个输出元素（因此 func 应该返回一个序列，而不是单一元素）
mapPartitions(func)	类似于 map，但独立地在 RDD 的每一个分块上运行，因此在类型为 T 的 RDD 上运行时，func 的函数类型必须是 Iterator[T] => Iterator[U]
mapPartitionsWithSplit(func)	类似于 mapPartitions，但 func 带有一个整数参数表示分块的索引值。因此在类型为 T 的 RDD 上运行时，func 的函数类型必须是(Int, Iterator[T]) => Iterator[U]
coalesce(numPartitions, shuffle: = false)	合并父 RDD 的分区，使得其分区数为 numPartitions。Shuffle 是表示是否需要进行 shuffle 过程。常用于对从 hdfs 文件里读取大量小分区的 HadoopRDD 进行合并。
sample(withReplacement,fraction, seed)	根据 fraction 指定的比例，对数据进行采样，可以选择是否用随机数进行替换，seed 用于指定随机数生成器种子
union(otherDataset)	返回一个新的数据集，新数据集是由源数据集和参数数据集联合而成
distinct([numTasks]))	返回一个包含源数据集中所有不重复元素的新数据集
groupByKey([numTasks])	在一个（K,V）对的数据集上调用，返回一个（K，Seq[V])对的数据集 注意：默认情况下，只有8个并行任务来做操作，但是你可以传入一个可选的 numTasks 参数来改变它
reduceByKey(func, [numTasks])	在一个（K，V)对的数据集上调用时，返回一个（K，V）对的数据集，使用指定的 reduce 函数，将相同 key 的值聚合到一起。类似 groupByKey，reduce 任务个数是可以通过第二个可选参数来配置的
sortByKey([ascending], [numTasks])	在一个（K，V)对的数据集上调用，K 必须实现 Ordered 接口，返回一个按照 Key 进行排序的（K，V）对数据集。升序或降

	序由 ascending 布尔参数决定
join(otherDataset, [numTasks])	在类型为（K,V)和（K,W)类型的数据集上调用时，返回一个相同 key 对应的所有元素对在一起的(K, (V, W))数据集
cogroup(otherDataset, [numTasks])	在类型为（K,V)和（K,W)的数据集上调用，返回一个 (K, Seq[V], Seq[W])元组的数据集。这个操作也可以称之为 groupwith
cartesian(otherDataset)	笛卡尔积，在类型为 T 和 U 类型的数据集上调用时，返回一个 (T, U)对数据集(两两的元素对)

完整的转换列表可以在 RDD API doc 中获得。

动作（actions）

动作	含义
reduce(func)	通过函数 func（接受两个参数，返回一个参数）聚集数据集中的所有元素。这个功能必须可交换且可关联的，从而可以正确的被并行执行。
collect()	在驱动程序中，以数组的形式，返回数据集的所有元素。这通常会在使用 filter 或者其它操作并返回一个足够小的数据子集后再使用会比较有用。
count()	返回数据集的元素的个数。
first()	返回数据集的第一个元素（类似于 take（1））
take(n)	返回一个由数据集的前 n 个元素组成的数组。注意，这个操作目前并非并行执行，而是由驱动程序计算所有的元素
takeSample(withReplacement,num, seed)	返回一个数组，在数据集中随机采样 num 个元素组成，可以选择是否用随机数替换不足的部分，Seed 用于指定的随机数生成器种子
saveAsTextFile(path)	将数据集的元素，以 textfile 的形式，保存到本地文件系统，HDFS 或者任何其它 hadoop 支持的文件系统。对于每个元素，Spark 将会调用 toString 方法，将它转换为文件中的文本行
saveAsSequenceFile(path)	将数据集的元素，以 Hadoop sequencefile 的格式，保存到指定的目录下，本地系统，HDFS 或者任何其它 hadoop 支持的文件系统。这个只限于由 key-value 对组成，并实现了 Hadoop

	的 Writable 接口，或者隐式的可以转换为 Writable 的 RDD。（Spark 包括了基本类型的转换，例如 Int，Double，String，等等）
countByKey()	对(K,V)类型的 RDD 有效，返回一个(K，Int)对的 Map，表示每一个 key 对应的元素个数
foreach(func)	在数据集的每一个元素上，运行函数 func 进行更新。这通常用于边缘效果，例如更新一个累加器，或者和外部存储系统进行交互，例如 HBase

完整的转换列表可以在 RDD API doc 中获得。

RDD 的持久化

Spark 最重要的一个功能，**就是在不同操作间，持久化（或缓存）一个数据集在内存中。**当你持久化一个 RDD，每一个结点都将把它的计算分块结果保存在内存中，并在对此数据集（或者衍生出的数据集）进行的其它动作中重用。这将使得后续的动作(Actions)变得更加迅速（通常快10倍）。缓存是用 Spark 构建迭代算法的关键。

你可以用 persist()或 cache()方法来标记一个要被持久化的 RDD，然后一旦首次被一个动作（Action）触发计算，它将会被保留在计算结点的内存中并重用。Cache 有容错机制，如果 RDD 的任一分区丢失了，通过使用原先创建它的转换操作，它将会被自动重算（不需要全部重算，只计算丢失的部分）。当需要删除被持久化的 RDD，可以用 unpersistRDD()来完成该工作。

此外，每一个 RDD 都可以用不同的保存级别进行保存，从而允许你持久化数据集在硬盘，或者在内存作为序列化的 Java 对象（节省空间），甚至于跨结点复制。这些等级选择，是通过将一个 org.apache.spark.storage.StorageLevel 对象传递给 persist()方法进行确定。cache()方法是使用默认存储级别的快捷方法，也就是 StorageLevel.MEMORY_ONLY(将反序列化的对象存入内存）。

StorageLevel 有五个属性，分别是：useDisk_ 是否使用磁盘，useMemory_ 是否使用内存，useOffHeap_是否使用堆外内存如：Tachyon，deserialized_是否进行反序列化，replication_ 备份数目。

完整的可选存储级别如下：

存储级别	意义
MEMORY_ONLY	将 RDD 作为反序列化的的对象存储 JVM 中。如果 RDD 不能被内存装下，一些分区将不会被缓存，并且在需要

	的时候被重新计算。这是是默认的级别
MEMORY_AND_DISK	将 RDD 作为反序列化的的对象存储在 JVM 中。如果 RDD 不能被与内存装下，超出的分区将被保存在硬盘上，并且在需要时被读取
MEMORY_ONLY_SER	将 RDD 作为序列化的的对象进行存储（每一分区占用一个字节数组）。通常来说，这比将对象反序列化的空间利用率更高，尤其当使用 fast serializer,但在读取时会比较占用 CPU
MEMORY_AND_DISK_SER	与 MEMORY_ONLY_SER 相似，但是把超出内存的分区将存储在硬盘上而不是在每次需要的时候重新计算
DISK_ONLY	只将 RDD 分区存储在硬盘上
MEMORY_ONLY_2, MEMORY_AND_DISK_2, etc.	与上述的存储级别一样，但是将每一个分区都复制到两个集群结点上

存储级别的选择

Spark 的不同存储级别，旨在满足内存使用和 CPU 效率权衡上的不同需求。我们建议通过以下的步骤来进行选择：

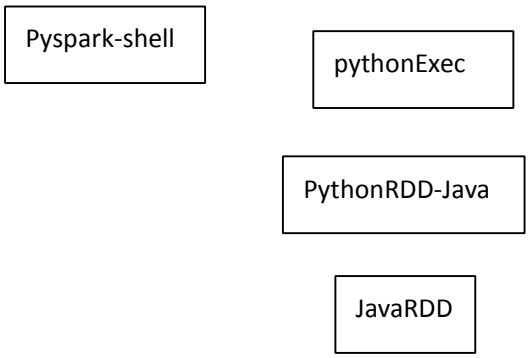
- 如果你的 RDDs 可以很好的与默认的存储级别(MEMORY_ONLY)契合，就不需要做任何修改了。这已经是 CPU 使用效率最高的选项，它使得 RDDs 的操作尽可能的快。
- 如果不行，试着使用 MEMORY_ONLY_SER 并且选择一个快速序列化的库使得对象在有比较高的空间使用率的情况下，依然可以较快被访问。
- 尽可能不要存储到硬盘上，除非计算数据集的函数，计算量特别大，或者它们过滤了大量的数据。否则，重新计算一个分区的速度，和与从硬盘中读取基本差不多快。

总结：调用 persist()或 cache()方法使用的是 MEMORY_ONLY 存储级别，对于广播变量，使用的是 MEMORY_AND_DISK 存储级别。如果想使用其他存储级别，可以调用 persist(StroageLevel)。MEMORY_AND_DISK 存储级别时当内存足够时直接保存到内存队列中，当内存不足时，将释放掉不属于同一个 RDD 的 block 的内存。

RDD 的 CheckPoint

当多次计算过程中有某些 RDD 被重复使用或者保持断点恢复功能，为了减少计算代价，可以调用 checkpoint()方法来标记 RDD 是需要被 checkpoint 的，当下次使用该 RDD 时，将直接使用该 RDD 的 CheckPointRdd，不再重新计算该 RDD 及其父 RDD。
一般最好将需要 checkpoint 的 RDD 事先 persist 或 cache，否则在进行 checkpoint 时需要重新计算这个 RDD 以及其父 RDD。因为 RDD 的 checkpoint 的真正执行是在 dagScheduler.runJob()之后，再调用 rdd.context.runJob()来完成真正的 checkpoint 的工作。

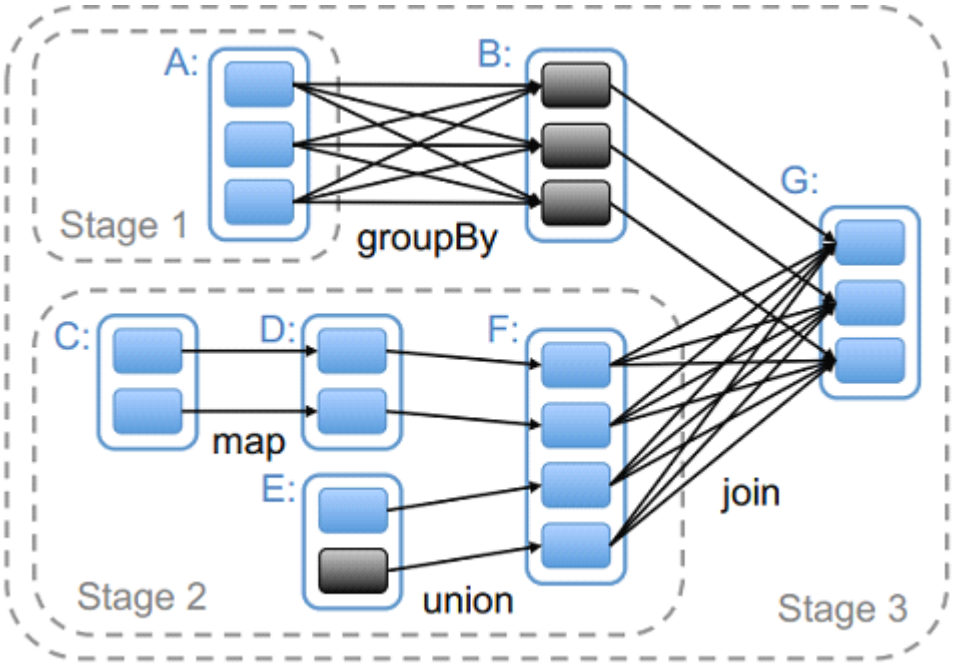
pySpark



JavaRDD

RDD 与 Stage 的划分

RDD 是指对于数据集的一次操作，比如从 hdfs 提取数据，filter, group by 等都算是一次操作。
所以 RDD 之间是有依赖关系。
根据依赖关系分为二种依赖类型：Narrow 和 Wide。
Narrow Dependency 指的是 child RDD 只依赖于 parent RDD(s)固定数量的 partition。
Wide Dependency 指的是 child RDD 的每一个 partition 都依赖于 parent RDD(s)所有 partition。
具体的理解为 Wide 依赖是需要结点之间进行 shuffle 操作的，其他的就是 Narrow。
所以 spark 将 shuffle 操作定义为 stage 的边界，有 shuffle 操作的 stage 叫做 ShuffleMapStage，对应的 task 叫做 ShuffleMapTask，其他的就是 ResultTask。
最终每一个 job 都为分解为多个 stage，并且这些 stage 形成 DAG 的依赖关系。如图所示：



RDD 的物理计算逻辑

RDD 的迭代计算是由每一个 Task.run()来触发的。

而在 RDD 的迭代方法 iterator()里时，会判断该 RDD 的存储形式是否为空，不为空，则将利用 CacheManager 进行缓存该 RDD 的结果。如下所示：

```
  final def iterator(split: Partition, context: TaskContext): Iterator[T] = {
    if (storageLevel != StorageLevel.NONE) {
      SparkEnv.get.cacheManager.getOrCompute(this, split, context,
storageLevel)
    } else {
      computeOrReadCheckpoint(split, context)
    }
  }
```

在 computeOrReadCheckpoint 方法，则会先判断是否 checkpoint 过，则调用 checkpointRdd 的 iterator 方法。如果不是 checkpoint 的 RDD，则调用 RDD 的 compute()方法来计算，因此每个 RDD 实现类都会有一个 compute()方法用于该 RDD 的物理计算逻辑。

比如 MappedRDD 的 compute()方法，如下所示：

```
override def compute(split: Partition, context: TaskContext) =
    firstParent[T].iterator(split, context).map(f)
```

调用父 RDD 的 iterator()方法获得 iter 对象，然后调用 iter 的 map()方法返回一个新的 iter 对象，但是在这个 iter 对象里的元素，都是从上一个 iter 对象取得的元素，经过函数 f 进行调用过后的值。

在 Scala 里 Iterator 类的 map()方法的实现如下：

```
 def map[B](f: A => B): Iterator[B] = new AbstractIterator[B] {
    def hasNext = self.hasNext
```

```
    def next() = f(self.next())
  }
```

CacheManager

而CacheManager里getOrCompute方法中最重要的就是与blockManger里的cache相关工作。最开始是判断 blockManger 是否已经有该 RDD 的该分片的 block，如果有，说明是被 persist 过的，直接从 blockManager 里提取值即可。
如果在 blockManger 没有当前计算的分片的 block 值，则会先调用 RDD 的 computeOrReadCheckpoint()方法进行计算得到值。

```
def getOrCompute[T](rdd: RDD[T], split: Partition, context: TaskContext,
storageLevel: StorageLevel)
      : Iterator[T] = {
    val key = "rdd_%d_%d".format(rdd.id, split.index)
blockManager.get(key) match {
      case Some(cachedValues) =>
        // Partition is in cache, so just return its values
        logInfo("Found partition in cache!")
        return cachedValues.asInstanceOf[Iterator[T]]
      case None =>
           // Mark the split as loading (unless someone else marks it first)

// If we got here, we have to load the split
          val elements = new ArrayBuffer[Any]
          logInfo("Computing partition " + split)
          elements ++= rdd.computeOrReadCheckpoint(split, context)
```

最后判断是否只是本地环境运行，如果是，则直接将结果返回，不做任务其他处理。
否则要根据 storageLevel 是 disk 还是 memory，将数据以 block 的形式 put 到 blockManager 上面，会统一通知给 blockManagerMaster 里。详见 CacheManager 里 getOrCompute()后半部分代码。

Serializer

数据对象的序列化

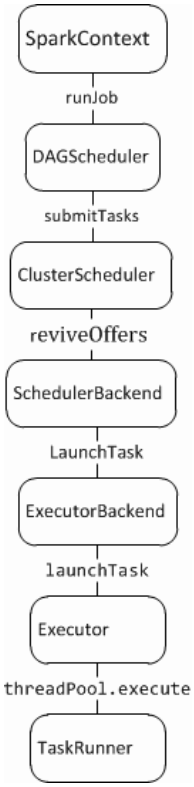
Java& kryo

RDD 的序列化

closureSerializer==JavaSerializer
ClosureCleaner
SparkILoop
spark.repl.Main

执行框架

sparkContext：指 spark 应用的上下文环境集合，包括与集群的连接，创建 RDD 的接口等。
DAGScheduler: 面向 stage 的调度，计算出 job 的 stage- DAG，向 TaskScheduler 提交 stage 的 taskset,同时跟踪 stage 的状态。
TaskScheduler：task 的调度，将 task 发送给集群，并执行它。
ExecutorBackend：slave 结点的后台执行接口。
Executor: 多线程 task 的执行器管理。
TaskRunner: task 的真正执行。



DAGScheduler

最重要的方法就是 runJob，用于接受一组 RDD 和最终结果的处理方法 func，partition 数目，是否只在本地执行，以及最终每个 task 的结果数组。
主要有二部分：一是 prepareJob 方法，既预处理该 job，将其提交到执行队列中，二是等待 job 的完成。

```
  def runJob[T, U: ClassManifest](
      finalRdd: RDD[T],
      func: (TaskContext, Iterator[T]) => U,
      partitions: Seq[Int],
      callSite: String,
      allowLocal: Boolean,
      resultHandler: (Int, U) => Unit)
  {
    if (partitions.size == 0) {
      return
    }
    val (toSubmit, waiter) = prepareJob(
        finalRdd, func, partitions, callSite, allowLocal, resultHandler)
    eventQueue.put(toSubmit)
    waiter.awaitResult() match {
      case JobSucceeded => {}
      case JobFailed(exception: Exception) =>
        logInfo("Failed to run " + callSite)
        throw exception
    }
  }
```

prepareJob

生成一个 JobWaiter 对象和 JobSubmitted 对象。

```
{
    assert(partitions.size > 0)
    val waiter = new JobWaiter(partitions.size, resultHandler)
    val func2 = func.asInstanceOf[(TaskContext, Iterator[_]) => _]
    val toSubmit = JobSubmitted(finalRdd, func2, partitions.toArray, allowLocal,
callSite, waiter)
    return (toSubmit, waiter)
  }
```

DAG-processEvent

DAGScheduler 事件处理函数。主要有 JobSubmitted，CompletionEvent，ExecutorLost，StopDAGScheduler，TaskSetFailed 这几类事件，重要的是前面二种。
当前的事件保存在 val eventQueue = new LinkedBlockingQueue[DAGSchedulerEvent]，通过 val event = eventQueue.poll(POLL_TIMEOUT, TimeUnit.MILLISECONDS)来获取当前需要处理的事件。

JobSubmitted

Job 提交事件，首先生成 stage，然后在本地或者提交给 TaskScheduler 去执行该 job。

```
val runId = nextRunId.getAndIncrement()
        val finalStage = newStage(finalRDD, None, runId)
        val job = new ActiveJob(runId, finalStage, func, partitions, callSite,
listener)
        clearCacheLocs()
        logInfo("Got job " + job.runId + " (" + callSite + ") with " +
partitions.length +
                " output partitions (allowLocal=" + allowLocal + ")")
        logInfo("Final stage: " + finalStage + " (" + finalStage.origin + ")")
        logInfo("Parents of final stage: " + finalStage.parents)
        logInfo("Missing parents: " + getMissingParentStages(finalStage))
        if (allowLocal && finalStage.parents.size == 0 && partitions.length ==
1) {
          // Compute very short actions like first() or take() with no parent
stages locally.
          runLocally(job)
        } else {
          activeJobs += job
          resultStageToJob(finalStage) = job
          submitStage(finalStage)
        }
```

newStage

创建新的 stage，如果有 shuffle 的话，还要向 mapOutputTracker 中心注册一下该 shuffle 工作。
还有一个重要的工作，就是得到所有它的父 stage。

```
  private def newStage(rdd: RDD[_], shuffleDep: Option[ShuffleDependency[_,_]],
priority: Int): Stage = {
```

```
    if (shuffleDep != None) {
      // Kind of ugly: need to register RDDs with the cache and map output tracker
here
      // since we can't do it in the RDD constructor because # of partitions is
unknown
      logInfo("Registering RDD " + rdd.id + " (" + rdd.origin + ")")
      mapOutputTracker.registerShuffle(shuffleDep.get.shuffleId,
rdd.partitions.size)
    }
    val id = nextStageId.getAndIncrement()
    val stage = new Stage(id, rdd, shuffleDep, getParentStages(rdd, priority),
priority)
    idToStage(id) = stage
    stage
  }
```

getParentStages

遍历所有的 RDD，如果 rdd 的依赖是 ShuffleDependency，则需创建一个 shuffle map stage，并将其保存到父 stage 列表中。

```
for (dep <- r.dependencies) {
          dep match {
            case shufDep: ShuffleDependency[_,_] =>
              parents += getShuffleMapStage(shufDep, priority)
            case _ =>
              visit(dep.rdd)
          }
        }
```

runLocally

开启一个线程，由于是本地执行，所以直接简单进行 func，即对 rdd 的迭代器进行计算。

```
val result = job.func(taskContext, rdd.iterator(split, taskContext))
```

然后再通知 job 的 listener 监视器。

```
new Thread("Local computation of job " + job.runId) {
      override def run() {
        try {
          SparkEnv.set(env)
          val rdd = job.finalStage.rdd
          val split = rdd.partitions(job.partitions(0))
          val taskContext = new TaskContext(job.finalStage.id,
```

```
job.partitions(0), 0)
          try {
            val result = job.func(taskContext, rdd.iterator(split,
taskContext))
            job.listener.taskSucceeded(0, result)
          } finally {
            taskContext.executeOnCompleteCallbacks()
          }
        } catch {
          case e: Exception =>
            job.listener.jobFailed(e)
        }
      }
    }.start()
  }
```

submitStage

首先计算当前 stage 的 missing parent stage，如果没有，则直接提交当前的 stage。
如果有，则对每个 missing parent stage，进行迭代进行 missing parent stage，直到没有为止，则将该 stage 提交。

```
  private def submitStage(stage: Stage) {
    logDebug("submitStage(" + stage + ")")
    if (!waiting(stage) && !running(stage) && !failed(stage)) {
      val missing = getMissingParentStages(stage).sortBy(_.id)
      logDebug("missing: " + missing)
      if (missing == Nil) {
        logInfo("Submitting " + stage + " (" + stage.rdd + "), which has no missing
parents")
        submitMissingTasks(stage)
        running += stage
      } else {
        for (parent <- missing) {
          submitStage(parent)
        }
        waiting += stage
      }
    }
  }
```

getMissingParentStages

如果rdd的依赖是ShuffleDependency,则新建一个shuffle map stage，且该stage是可用的话，则加入到missing parent stage队列中。

```
if (getCacheLocs(rdd).contains(Nil)) {
          for (dep <- rdd.dependencies) {
            dep match {
              case shufDep: ShuffleDependency[_,_] =>
                val mapStage = getShuffleMapStage(shufDep, stage.priority)
                if (!mapStage.isAvailable) {
                  missing += mapStage
                }
              case narrowDep: NarrowDependency[_] =>
                visit(narrowDep.rdd)
            }
          }
        }
```

submitMissingTasks

根据 stage 的性质，创建对应的 shuffleMapTask,ResultTask 的 task 集合。
并调用以下方法，来提交 task 集合的执行。

```
taskSched.submitTasks(
        new     TaskSet(tasks.toArray,     stage.id,     stage.newAttemptId(),
stage.priority))
```

waiter.awaitResult

TaskScheduler

Task 调度接口，具体实现有二个：ClusterScheduler，LocalScheduler。分别指集群执行调度和本地执行调度。

ClusterScheduler

submitTasks

提交 task 集合，为这个 taskSet 创建一个 TaskSetManager，然后将其放到 activeTaskSetsQueue 队列中。
注意：最后还会调用 SchedulerBackend 接口的 reviveOffers（）方法，该类有二种实现，详细见二种实现类的该方法。

```
override def submitTasks(taskSet: TaskSet) {
  val tasks = taskSet.tasks
  logInfo("Adding task set " + taskSet.id + " with " + tasks.length + " tasks")
  this.synchronized {
    val manager = new TaskSetManager(this, taskSet)
    activeTaskSets(taskSet.id) = manager
    activeTaskSetsQueue += manager
    taskSetTaskIds(taskSet.id) = new HashSet[Long]()

    if (hasReceivedTask == false) {
      starvationTimer.scheduleAtFixedRate(new TimerTask() {
        override def run() {
          if (!hasLaunchedTask) {
            logWarning("Initial job has not accepted any resources; " +
              "check your cluster UI to ensure that workers are registered")
          } else {
            this.cancel()
          }
        }
      }, STARVATION_TIMEOUT, STARVATION_TIMEOUT)
    }
    hasReceivedTask = true;
  }
  backend.reviveOffers()
}
```

resourceOffers

提供一个获取 task 的接口。先对 activeTaskSetsQueue 排序，然后遍历所有的资源，调用 manager.slaveOffer(execId, host, availableCpus(i))看是否有 task 可以运行。

```
def resourceOffers(offers: Seq[WorkerOffer]): Seq[Seq[TaskDescription]] = {
```

```
    synchronized {
      SparkEnv.set(sc.env)
      // Mark each slave as alive and remember its hostname
      for (o <- offers) {
        executorIdToHost(o.executorId) = o.hostname
        if (!executorsByHost.contains(o.hostname)) {
          executorsByHost(o.hostname) = new HashSet()
        }
      }
      // Build a list of tasks to assign to each slave
      val tasks = offers.map(o => new ArrayBuffer[TaskDescription](o.cores))
      val availableCpus = offers.map(o => o.cores).toArray
      var launchedTask = false
      for (manager <- activeTaskSetsQueue.sortBy(m => (m.taskSet.priority,
m.taskSet.stageId))) {
        do {
          launchedTask = false
          for (i <- 0 until offers.size) {
            val execId = offers(i).executorId
            val host = offers(i).hostname
            manager.slaveOffer(execId, host, availableCpus(i)) match {
              case Some(task) =>
                tasks(i) += task
                val tid = task.taskId
                taskIdToTaskSetId(tid) = manager.taskSet.id
                taskSetTaskIds(manager.taskSet.id) += tid
                taskIdToExecutorId(tid) = execId
                activeExecutorIds += execId
                executorsByHost(host) += execId
                availableCpus(i) -= 1
                launchedTask = true

              case None => {}
            }
          }
        } while (launchedTask)
      }
      if (tasks.size > 0) {
        hasLaunchedTask = true
      }
      return tasks
    }
  }
```

TaskSetManager

slaveOffer

```
  def slaveOffer(execId: String, host: String, availableCpus: Double):
Option[TaskDescription] = {
调用findTask，如果有task，则返回TaskDescription描述信息。
findTask(host, localOnly)
```

findTask

根据给定的结点地址，以及是否调度本地执行的参数，来找到需要执行的 task 在队列中的位置。

```
  private def findTask(host: String, localOnly: Boolean): Option[Int] = {
    val localTask = findTaskFromList(getPendingTasksForHost(host))
    if (localTask != None) {
      return localTask
    }
    val noPrefTask = findTaskFromList(pendingTasksWithNoPrefs)
    if (noPrefTask != None) {
      return noPrefTask
    }
    if (!localOnly) {
      val nonLocalTask = findTaskFromList(allPendingTasks)
      if (nonLocalTask != None) {
        return nonLocalTask
      }
    }
    // Finally, if all else has failed, find a speculative task
    return findSpeculativeTask(host, localOnly)
  }
```

SchedulerBackend

后台的集群调度接口，StandaloneSchedulerBackend，MesosSchedulerBackend。

StandaloneSchedulerBackend

reviveOffers

其实就是向 DriverActor 发送 ReviveOffers 事件。
该方法由 ClusterScheduler 的 submitTasks 方法最后会调用它。

DriverActor

ReviveOffers 事件处理过程：
首先调用了 ClusterScheduler 的 resourceOffers 方法得到相应的 task 描述信息集合。
然后向各个 slaves 的机器发送 LaunchTask(task)的事件。

```
def makeOffers() {
      launchTasks(scheduler.resourceOffers(
        executorHost.toArray.map {case (id, host) => new WorkerOffer(id, host,
freeCores(id))}))
    }
```

MesosSchedulerBackend

LocalScheduler

本地执行的 task 调度。相对很简单，直接遍历 DAGScheduler 产生的 stage 的 taskSet，在本地多线程的方式去执行每个 task。

```
    def submitTask(task: Task[_], idInJob: Int) {
      val myAttemptId = attemptId.getAndIncrement()
      threadPool.submit(new Runnable {
        def run() {
          runTask(task, idInJob, myAttemptId)
        }
      })
    }
```

runTask 就是真正的 task 的执行。

Executor

ExecutorBackend

Slave 的执行后台接口，具体实现有 StandaloneExecutorBackend，MesosExecutorBackend，分别是

StandaloneExecutorBackend

最开始在 preStart()会向 StandaloneSchedulerBackend 发送事件 RegisterExecutor，向其注册当前自己的 execute 相关信息。

Main()函数启动一个 Actor 后台服务。

```
val (actorSystem, boundPort) = AkkaUtils.createActorSystem("sparkExecutor",
hostname, 0)
    val actor = actorSystem.actorOf(
      Props(new StandaloneExecutorBackend(new Executor, driverUrl, executorId,
hostname, cores)),
      name = "Executor")
    actorSystem.awaitTermination()
```

Receive 多个事件消息。

LaunchTask 事件：

```
executor.launchTask(this, taskDesc.taskId, taskDesc.serializedTask)
```

MesosExecutorBackend

Executor

launchTask():

线程池去执行 task。

```
threadPool.execute(new TaskRunner(context, taskId, serializedTask))
```

TaskRunner

就是真正的 task 执行线程。
首先建立起 sparkEnv 的环境，反序列化得到 task 所需要文件，jar 包等信息。
然后更新 mapOutputTracker。
这时开始真正的 task执行，`task.run(taskId.toInt)`，即将会执行具体的`shuffleMapTask`或者`ResultTask`。
最后将 task 执行完的结果序列化，保存到 TaskResult,返回给 ClusterScheduler。

```
SparkEnv.set(env)
      Thread.currentThread.setContextClassLoader(urlClassLoader)
      val ser = SparkEnv.get.closureSerializer.newInstance()
      logInfo("Running task ID " + taskId)
      context.statusUpdate(taskId, TaskState.RUNNING, EMPTY_BYTE_BUFFER)
      try {
        SparkEnv.set(env)
        Accumulators.clear()
        val (taskFiles, taskJars, taskBytes) =
Task.deserializeWithDependencies(serializedTask)
        updateDependencies(taskFiles, taskJars)
        val task = ser.deserialize[Task[Any]](taskBytes,
Thread.currentThread.getContextClassLoader)
        logInfo("Its generation is " + task.generation)
        env.mapOutputTracker.updateGeneration(task.generation)
        val value = task.run(taskId.toInt)
        val accumUpdates = Accumulators.values
        val result = new TaskResult(value, accumUpdates)
        val serializedResult = ser.serialize(result)
```

ShuffleMapTask

是指需要进行 shuffle 的 MapTask.

```
  override def run(attemptId: Long): MapStatus = {
    val numOutputSplits = dep.partitioner.numPartitions

    val taskContext = new TaskContext(stageId, partition, attemptId)
    try {
      // Partition the map output.
      val buckets = Array.fill(numOutputSplits)(new ArrayBuffer[(Any, Any)])
      for (elem <- rdd.iterator(split, taskContext)) {
        val pair = elem.asInstanceOf[(Any, Any)]
```

```
        val bucketId = dep.partitioner.getPartition(pair._1)
        buckets(bucketId) += pair
      }//对RDD的结果进行分片，存放到buckets数组中。

      val compressedSizes = new Array[Byte](numOutputSplits)

      val blockManager = SparkEnv.get.blockManager
      for (i <- 0 until numOutputSplits) {
//针对每一个分片，将其结果放到blockManager里，最后将blockManagerId和结果大小作为MapStatus对象返回给DAGScheduler。
        val blockId = "shuffle_" + dep.shuffleId + "_" + partition + "_" + i
        // Get a Scala iterator from Java map
        val iter: Iterator[(Any, Any)] = buckets(i).iterator
        val size = blockManager.put(blockId, iter, StorageLevel.DISK_ONLY,
false)
        compressedSizes(i) = MapOutputTracker.compressSize(size)
      }

      return new MapStatus(blockManager.blockManagerId, compressedSizes)
    } finally {
      // Execute the callbacks on task completion.
      taskContext.executeOnCompleteCallbacks()
    }
  }
```

ResultTask

主要实现了 run 方法。

```
  override def run(attemptId: Long): U = {
    val context = new TaskContext(stageId, partition, attemptId)
    try {
      func(context, rdd.iterator(split, context))
    } finally {
      context.executeOnCompleteCallbacks()
    }
  }
```

就是调用自定义的 func 函数来处理 rdd 的数据集合。

比如在 GroupByTest 的例子中,func 是

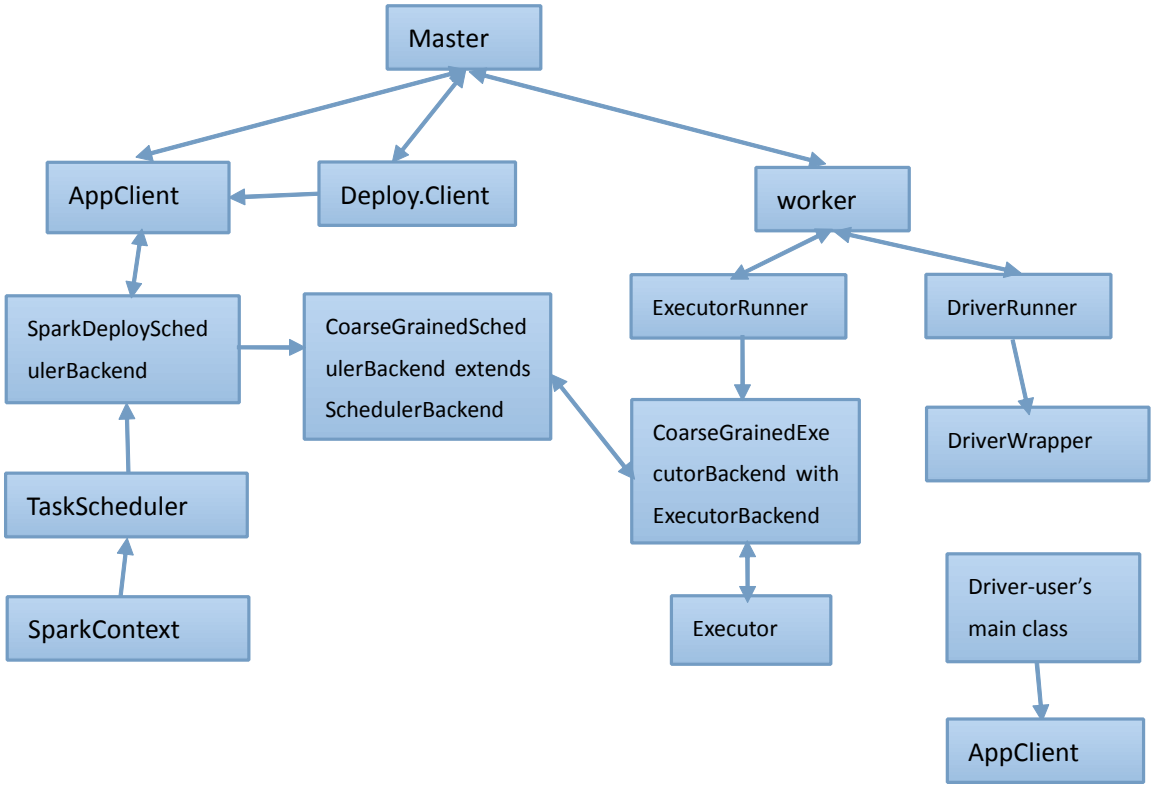
```
(iter: Iterator[T]) => {
      var result = 0L
      while (iter.hasNext) {
        result += 1L
        iter.next()
```

```
    }
    result
  })
```

这个 func 的作用就是 count。

布署模式

Standalone



Mesos

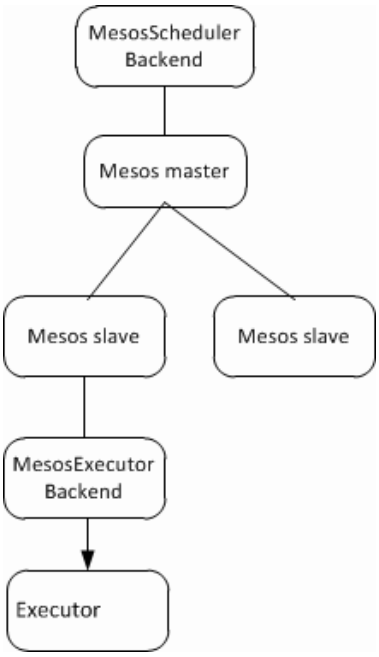
细粒度的资源管理框架。

https://github.com/apache/mesos/blob/trunk/docs/Mesos-Architecture.md

https://github.com/apache/mesos/blob/trunk/docs/App-Framework-development-guide.textile

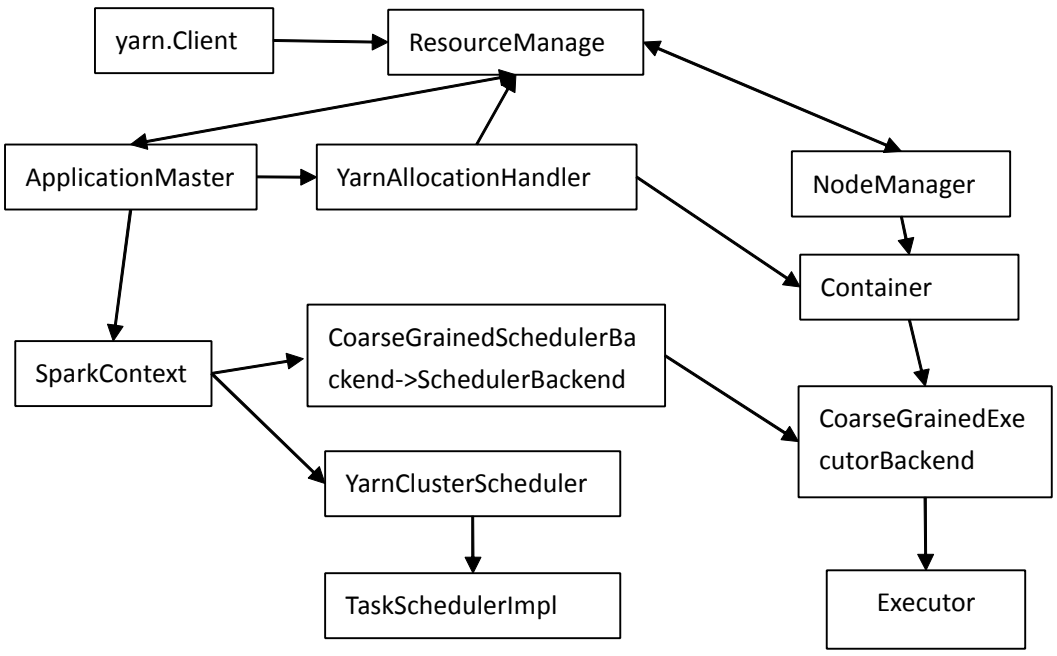
运行在 mesos 上的框架主要有二个部件：scheduler, executor。分别是各种计算模型用来消费

资源的调度接口，在 slaves 结点上执行具体 task 的执行接口。



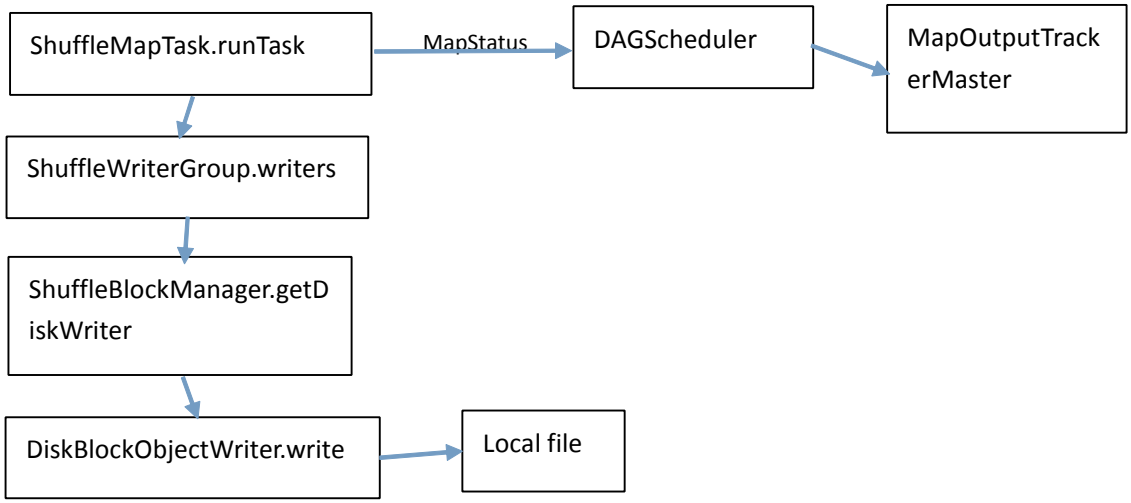
Yarn

Yarn-standalone 模式

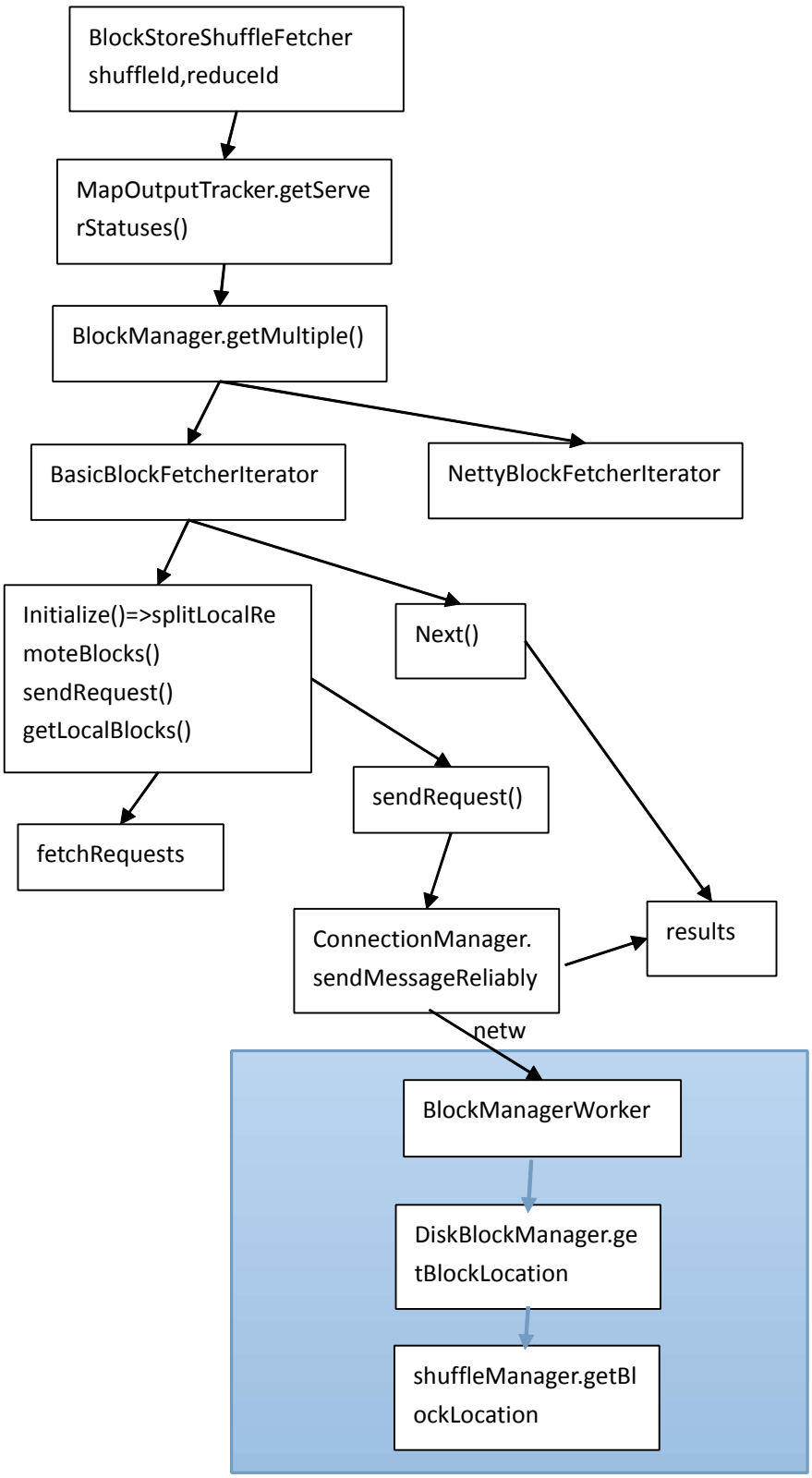


Shuffle

Spark 里 shuffle 的优点：
Map 和 reduce 减少了不必要的 sort 的逻辑。
Map 在 shuffle 环节没有 combine 过程，而是将 combine 交给上层操作来完成。故也减少了 combine 可能带来的多次合并 IO 操作。
Reduce 端在数据量小的情况下，不需要刷磁盘，直接给上层计算操作使用。
当然也有一些缺点：
Reduce 端直接向 map 内存拉数据，因此必须保证 map 端在一个 reduce 上的数据可以在内存里存放下。
Shuffle 的数据是拉模式，由 ShuffleRDD 来触动，所以比推模式在特定场景下要消耗更多的时间。
Reduce 必须在所有 Map 都结束了才开始启动数据拉动工作。
当数据量比较大，且必须要排序的情况下，reduce 端会有很多瓶颈。一方面是现在的外部合并还不支持对 key 进行排序，而只是对 key 的 hashcode 进行归并排序。另外一方面，所有的排序都是在 reduce 一端进行的，所以并行性不够理想。
Shuffle 主要有二部分组成：一部分是 map 端的 spill 模块，另外一部分是 reduce 的拉数据模块。
Shuffle writer 模块的架构如下所示：



Reduce 端数据拉取的模块架构如下：



ShuffleFetcher

获取 shuffle 的结果类
BlockStoreShuffleFetcher 为具体实现类，
调用 MapOutputTracker 获得 map 的相关输出的 block 的位置信息，如 BlockManagerId，
然后再调用 BlockManager 方法

```
getMultiple(blocksByAddress: Seq[(BlockManagerId, Seq[(String, Long)])])
      : Iterator[(String, Option[Iterator[Any]])]
```

得到结果的迭代器。

AppendOnlyMap

def destructiveSortedIterator(cmp: Comparator[(K, V)]): Iterator[(K, V)]
利用传过来的比较器对内存数据结构进行排序，并返回迭代器
主要是调用 JDK 提供的 Arrays.sort(data, 0, newIndex, rawOrdering) 即 mergesort 来进行排序。

SizeTrackingAppendOnlyMap

ExternalAppendOnlyMap

先定义一个 SizeTrackingAppendOnlyMap 的数据结构中，
如果当前内存足够，先将当前的 KV 放到之前定义好的 SizeTrackingAppendOnlyMap 内存中。
如果内存不够，则进行 spill 动作。
Spill 动作触发的条件(满足所有)：
记录数对大于 1000 并且记录数等于 0.7*当前容量 capacity

```
val mapSize = currentMap.estimateSize()
val availableMemory = maxMemoryThreshold -
            (shuffleMemoryMap.values.sum - previouslyOccupiedMemory.getOrElse(0L))
 shouldSpill = availableMemory < mapSize * 2
```

Spill:
从 diskblockManager 里创建一个新的临时 block 文件，并对该文件创建 writer 对象。文件句柄的 buffer 由参数 spark.shuffle.file.buffer.kb 决定。
使用 KV 的比较器对当前的 SizeTrackingAppendOnlyMap 的内存对象进行排序，返回一个迭代器。
然后遍历迭代器，将数据写到 block 本地文件中。每到 spark.shuffle.spill.batchSize 条记录里

进行文件 flush 一次，再重新写。
然后再创建一个新的 SizeTrackingAppendOnlyMap 的内存对象。
将该 block 本地文件创建一个 DiskMapIterator 迭代器，并且此失代器放到 spilledMaps 对象中。

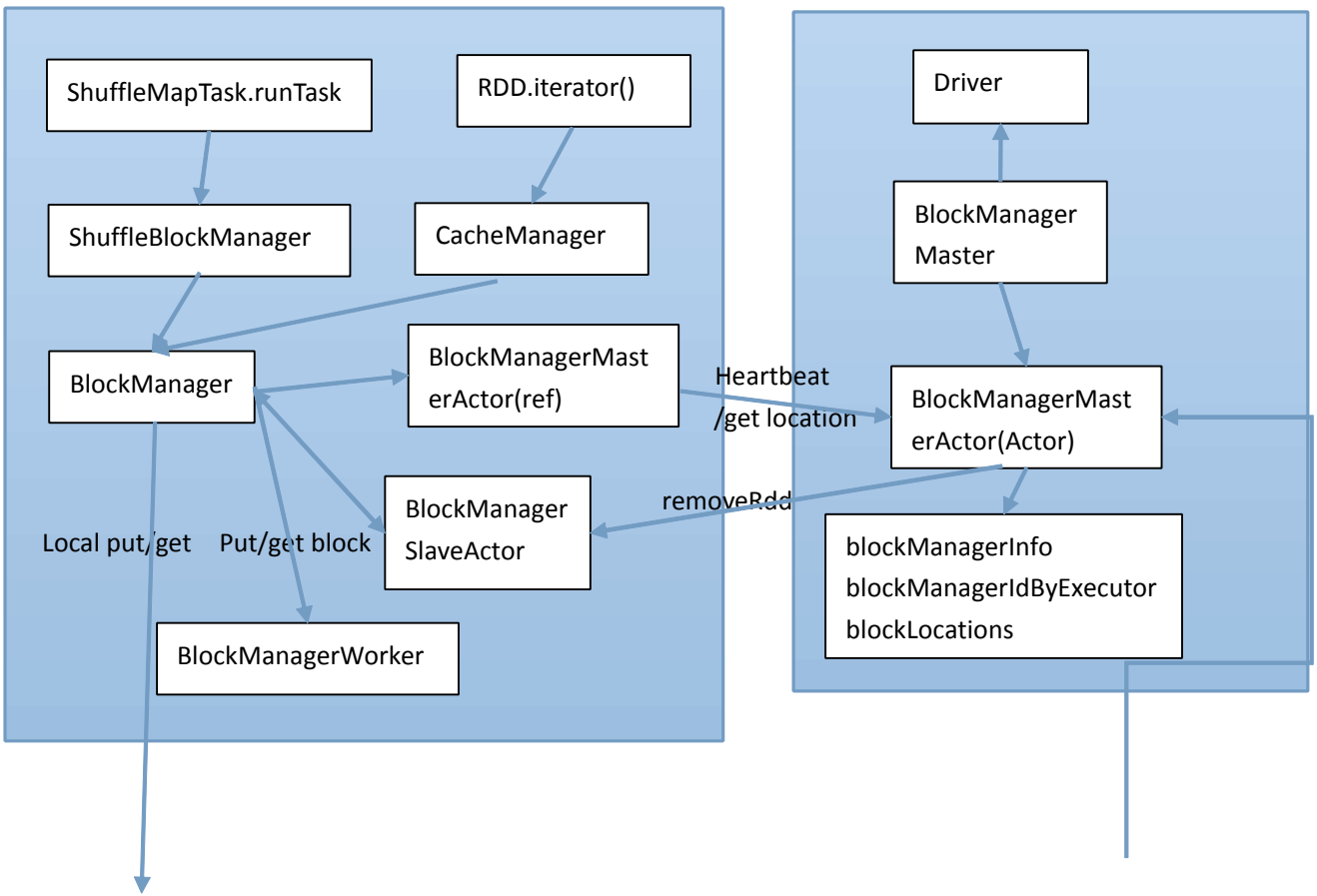
ExternalIterator 外部文件的 sort-merge 的迭代器
对于本地内存以及文件里的 map 数据块构建一个堆 mergeHeap
最开始对每个 map 数据块读取第一个 k 的 kv 对。
调用 next()时 首先从 mergeHeap 取得最小的 key 的元数，然后依次遍历堆，直到没有相同的 K 的 kv 对后结束，并对相同的 K 做合并操作。
当某个数据块里面的 buffer 为空，就从该数据块中取下一个 K，并将其放到堆中。

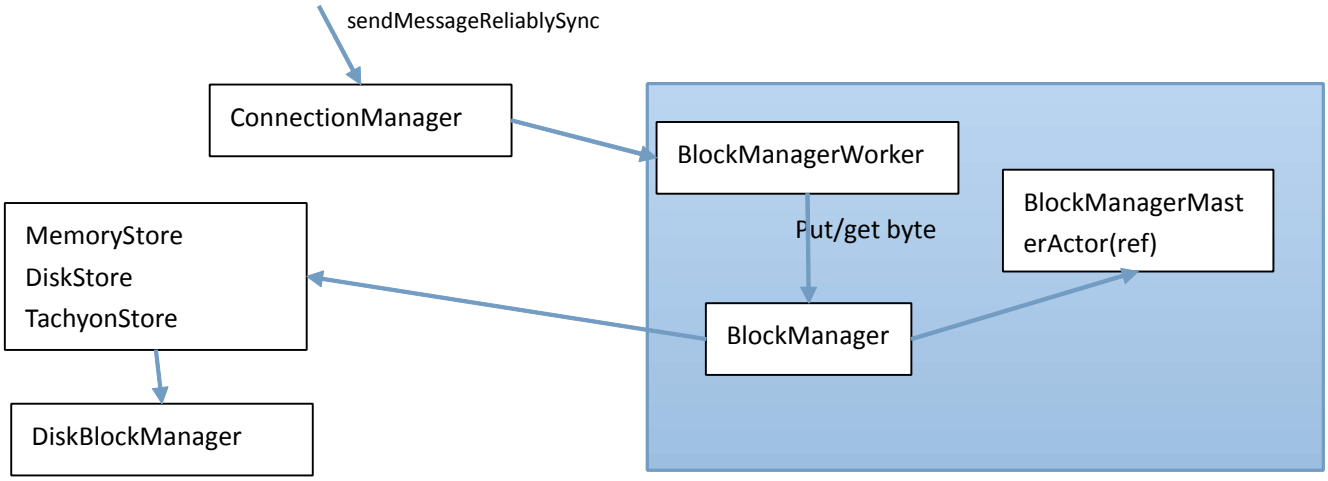
MapOutputTracker

Map 输出结果的汇总
BlockStoreShuffleFetcher

Storage 模块

Storage 是 spark 区别于 MR，Tez，Impala 等系统的重要特性。主要用于将计算完成或需要广播的数据以 block 为单位的保存在相应的 worker 结点上。
因此每个 Executor 都会有管理当前 Executor 的 BlockManagerWorker 和提供与 Master 通信的客户端接口 BlockManager。
在 master 结点，会有一个 BlockManagerMasterActor 来管理所有 slave 上的 BlockManagerWorker 的相关状态信息。





BlockManagerMaster

BlockManagerMasterActor
是 block manager 的 master 结点，主要用来接收所有 slave 上的 block manager 状态和消息，并做相应的处理。有二种模式：本地和远程模式。
其中主从结点之间的事件通知是用 ActorSystem 来做处理的。
其保存了 block manager 的相关相信息，如：
HashMap[BlockManagerId, BlockManagerMasterActor.BlockManagerInfo]
Block manager id 及 block 相关状态信息。

new JHashMap[String, Pair[Int, HashSet[BlockManagerId]]]
block id 及其保存的 block manager 的信息。

BlockManager

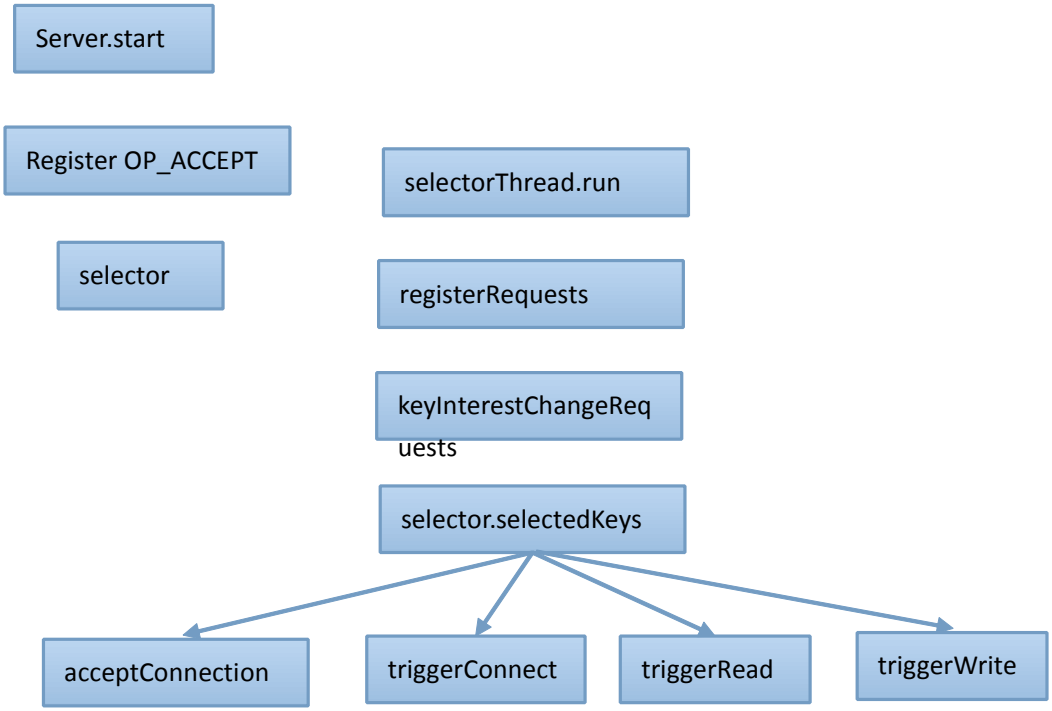
Block 管理的 slave 结点主类，与 BlockManagerMaster 进行交互通信的模块。
主要信息有：
BlockStore = new MemoryStore(this, maxMemory)
本地内存存储
```
BlockStore = new DiskStore(this, System.getProperty("spark.local.dir",
System.getProperty("java.io.tmpdir")))
```
本地磁盘存储

重要的接口有：
```
  def put(blockId: String, values: ArrayBuffer[Any], level: StorageLevel,
    tellMaster: Boolean = true) : Long = {
```
将一个数组 values 保存到 blockManager 中，一般该数组值都是通过迭代器得到的。
```
def getLocations(blockId: String): Seq[String]
```

从 blockManager 中取出该 id 相应的数据。
`def getLocal(blockId: String): Option[Iterator[Any]]`
从本地存储系统中取出相应 id 的原始数据。

ConnectionManager

基于 selector 管理



handleMessageExecutor 处理消息的线程池，默认为 20-60
handleReadWriteExecutor 处理网络读写的线程池，默认为 4-32
handleConnectExecutor 处理 Socket 连接的线程池，默认为 1-8

ContextCleaner

BroadcastManager

广播实现的的管理。
有二种实现广播的方式：HttpBroadcast 和 TorrentBroadcast。
在创建 HttpBroadcast 对象时 会将该值保存到当前的进程的 blockManager 里，如果运行的 master 不是本地模式，则还会将值写到 httpServer 的文件目录里。

当读取广播对象时，首先判断本地 blockManager 里是否已经有该值，如果没有，则从 httpServer 里拿到该值，然后再将值写到 blockManager 里，以供下一次继续使用。

优点：
当一个进程同时运行多个 task 时，只需要下载一次后，可以多次使用广播变量。因为它会保存到进程的 blockManager 里面，以供其他 task 使用。
缺点：
广播变量是要占用 blockManager 中的内存或磁盘，它的存储级别为 MEMORY_AND_DISK。保存在 blockManager 上后，无法手动删除，只能等 blockManager 下次 put 一个 block 时，判断当前内存使用量与设置的阀值比较，来进行是否删除内存数据，再刷到磁盘上。

TorrentBroadcast

Spark SQL

HttpFileServer

Bagel

MLlib

SGD

LBFGS

OWLQN

Graphx

监控系统

每一步 io 产生的数据记录数,大小及时间开稍
blockManger 的监控
Shuffle 过程中的 fetch 进度条

总结