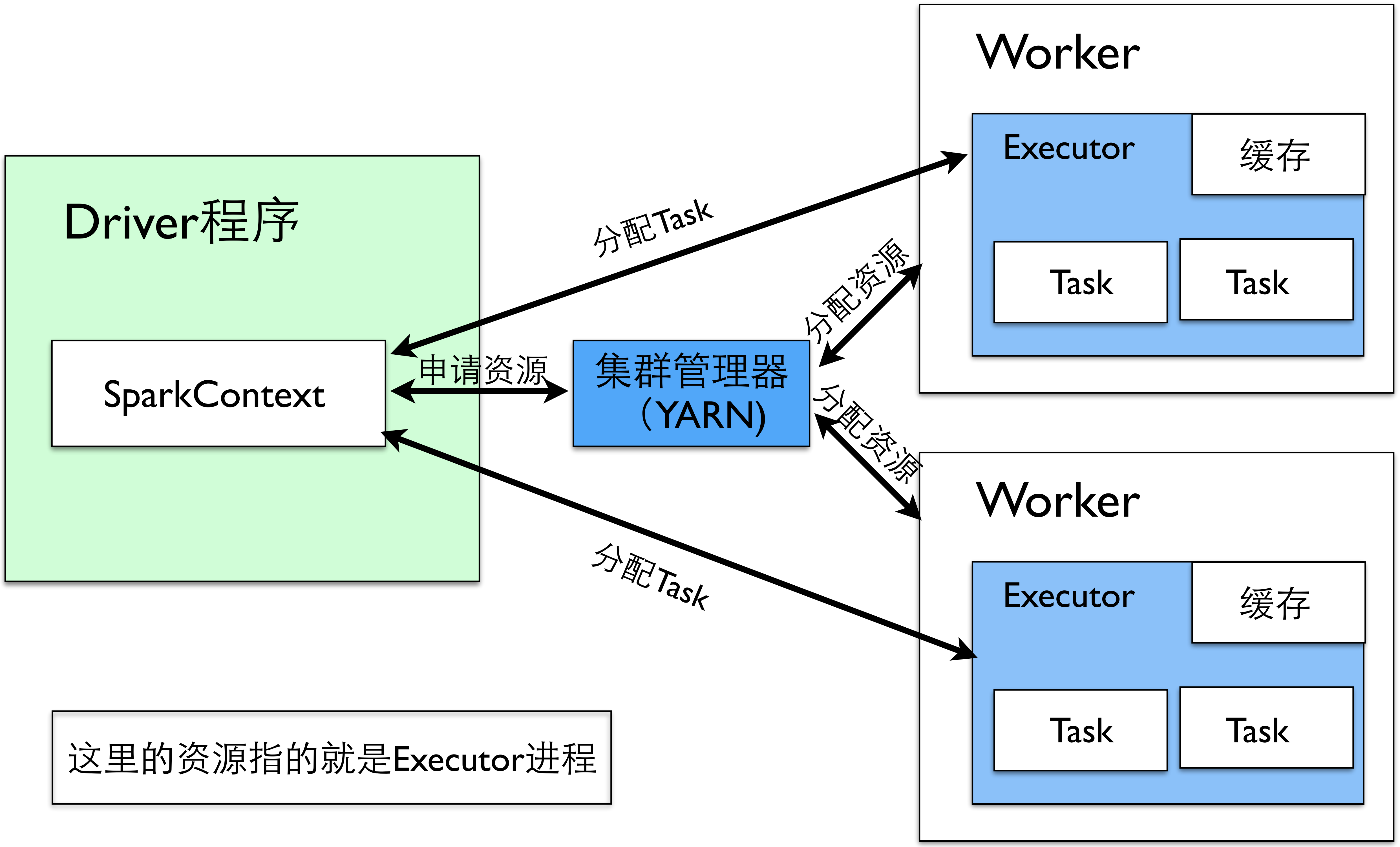
# Spark Core深度分析 & Spark SQL在产品中的运用

张逸

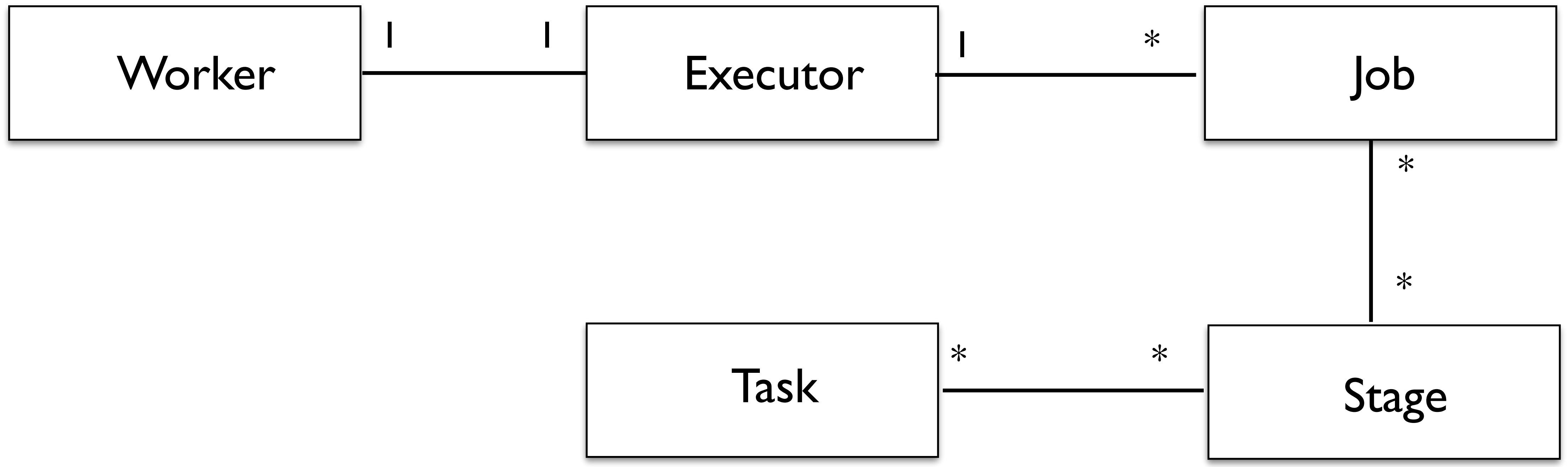
Big Eye

# 集群模式下运行的Spark程序



这里的资源指的就是Executor进程

# Spark执行过程中的相关术语



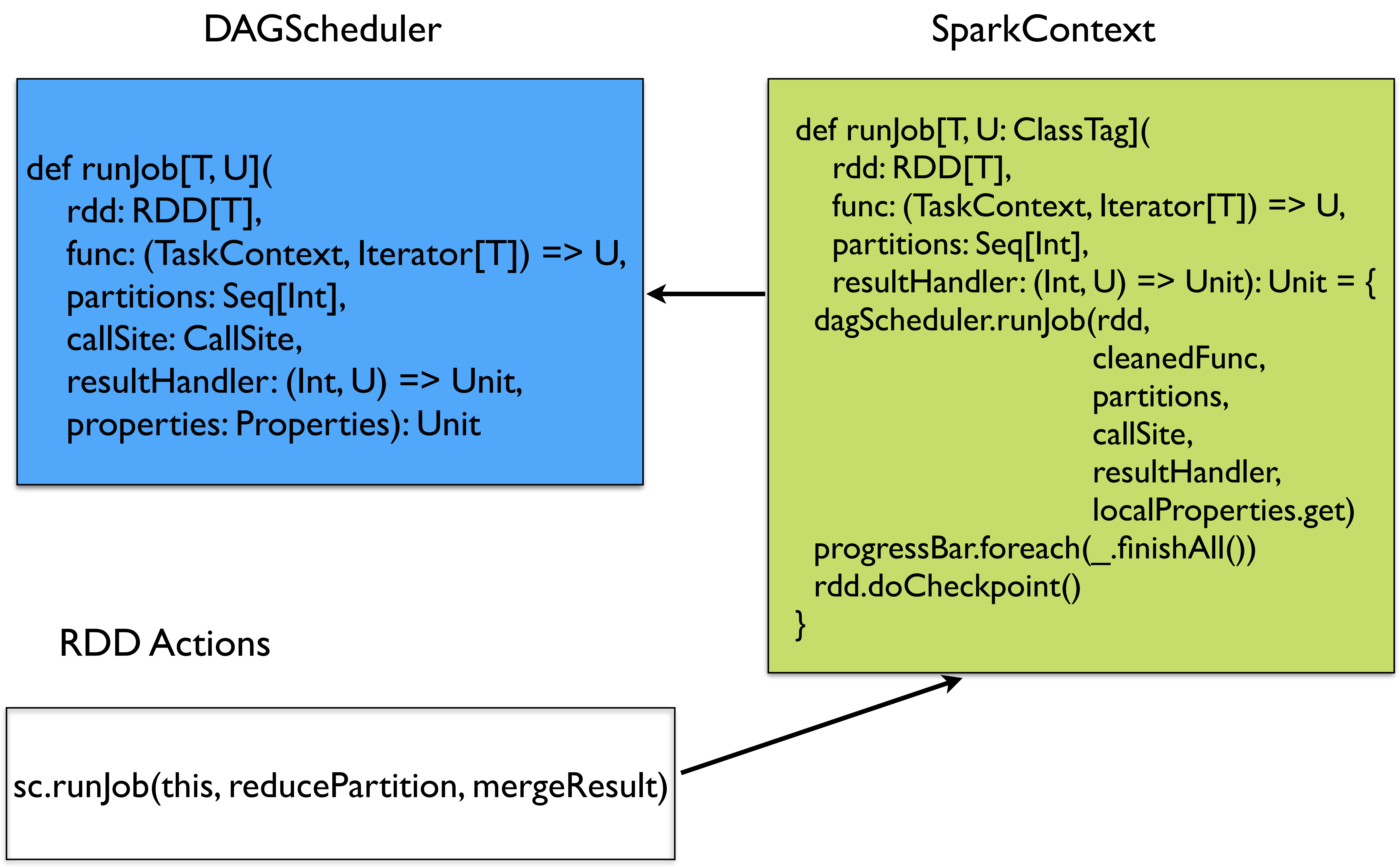
每个节点会启动一个进程，名为Worker。而在节点上每运行一个Spark程序，就是一个Executor。
在Executor中，每次执行RDD Action，就会提交一个Job。Job在执行过程中被分为多个Stage，而一次Stage又包含了多个Task。
task是最小的计算单元，负责执行一模一样的计算逻辑，只是每个task处理的数据不同而已。

# SparkContext的执行过程

```
// 创建和启动TaskScheduler
val (sched, ts) = SparkContext.createTaskScheduler(this, master, deployMode)
_schedulerBackend = sched
_taskScheduler = ts
_dagScheduler = new DAGScheduler(this)
_heartbeatReceiver.ask[Boolean](TaskSchedulerIsSet)


_taskScheduler.start()
```

将SparkContext传入，创建DAGScheduler:
def this(sc: SparkContext) = this(sc, sc.taskScheduler)

# SparkContext的执行过程



DAGScheduler

```
def runJob[T, U](
    rdd: RDD[T],
    func: (TaskContext, Iterator[T]) => U,
    partitions: Seq[Int],
    callSite: CallSite,
    resultHandler: (Int, U) => Unit,
    properties: Properties): Unit
```

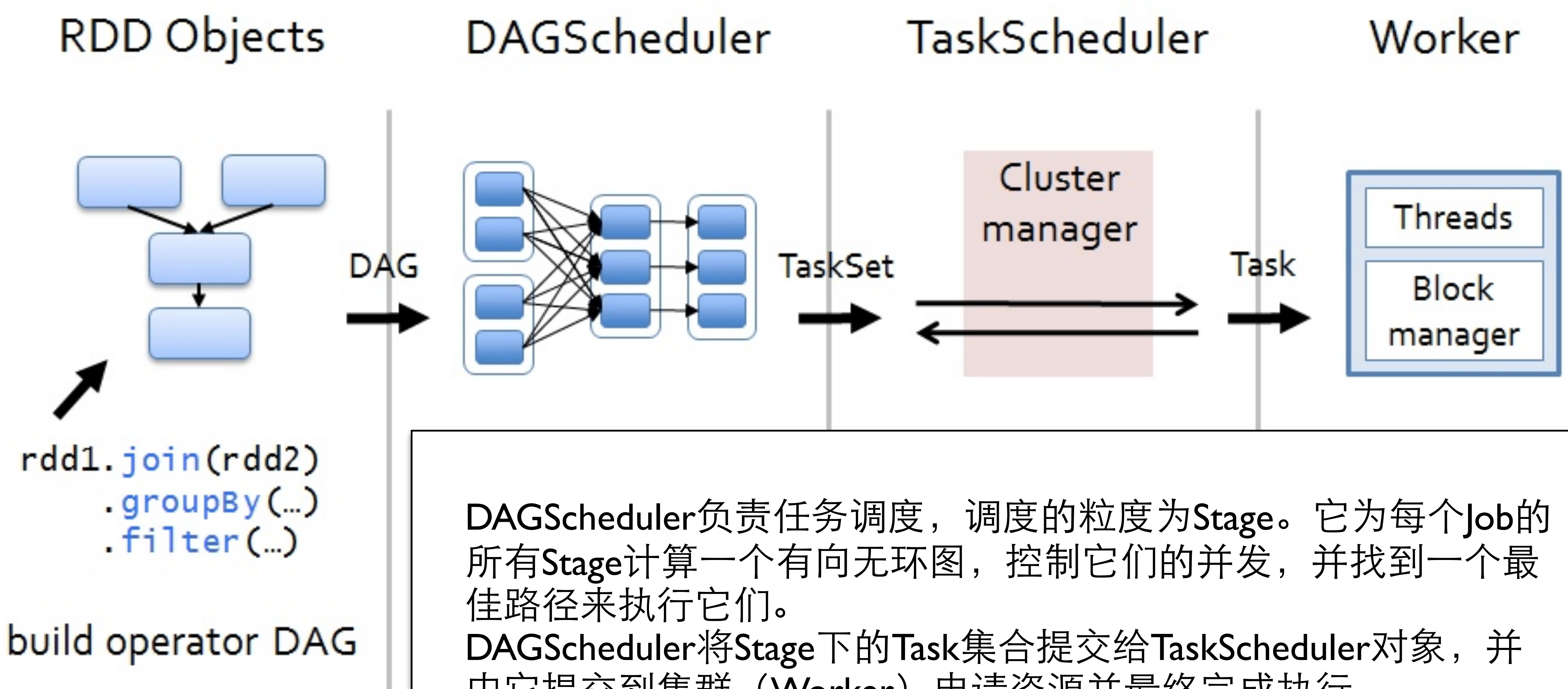
SparkContext

```
def runJob[T, U: ClassTag](
    rdd: RDD[T],
    func: (TaskContext, Iterator[T]) => U,
    partitions: Seq[Int],
    resultHandler: (Int, U) => Unit): Unit = {
  dagScheduler.runJob(rdd,
                      cleanedFunc,
                      partitions,
                      callSite,
                      resultHandler,
                      localProperties.get)
  progressBar.foreach(_.finishAll())
  rdd.doCheckpoint()
}
```

RDD Actions

```
sc.runJob(this, reducePartition, mergeResult)
```

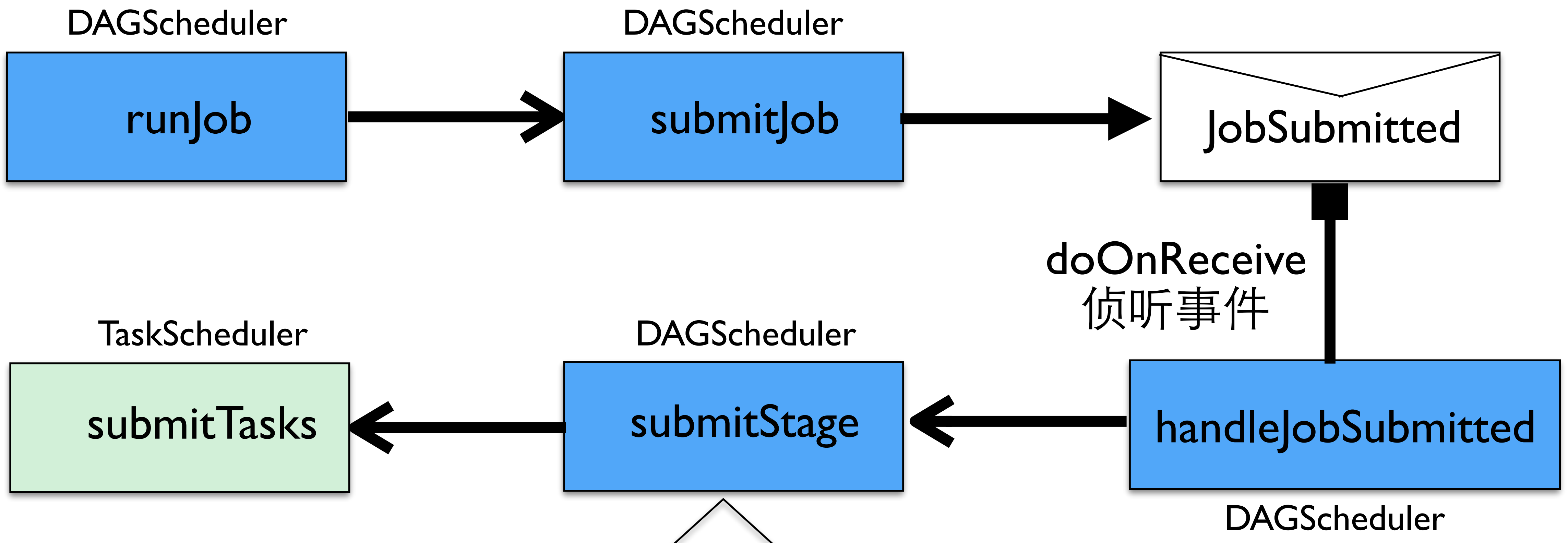
# 执行模型



```
rdd1.join(rdd2)
    .groupBy(…)
    .filter(…)
```

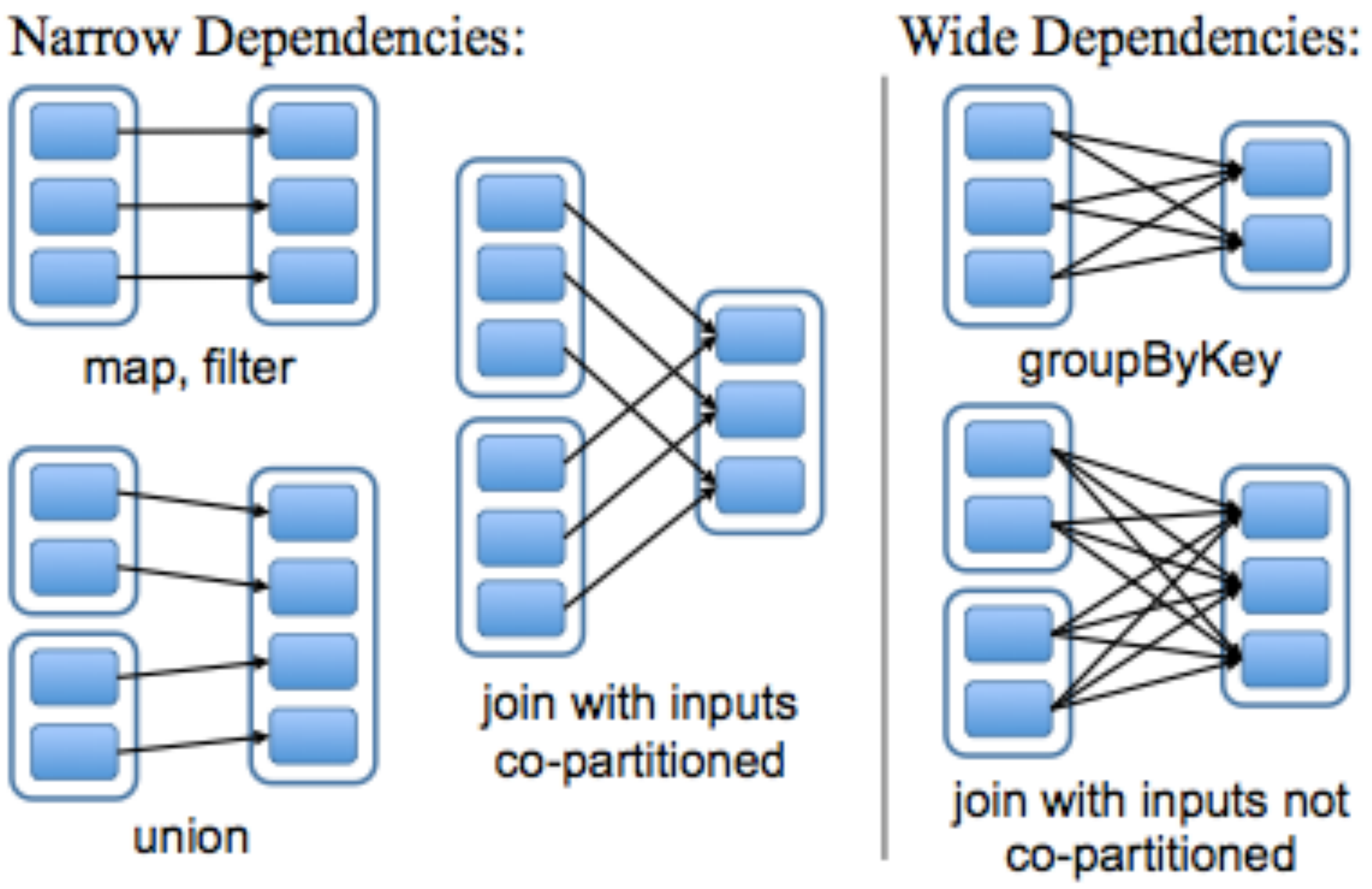
DAGScheduler负责任务调度，调度的粒度为Stage。它为每个Job的所有Stage计算一个有向无环图，控制它们的并发，并找到一个最佳路径来执行它们。
DAGScheduler将Stage下的Task集合提交给TaskScheduler对象，并由它提交到集群（Worker）申请资源并最终完成执行。
DAGScheduler定义的runJob方法会通过submitJob提交任务，之后由submitStage以递归方式提交stage。

# Scheduler的执行过程



Spark是根据shuffle类算子来进行stage的划分。如果我们的代码中执行了某个shuffle类算子（比如reduceByKey、join等），那么就会在该算子处，划分出一个stage界限来。shuffle算子执行以及之后的代码会被划分为下一个stage。因此一个stage刚开始执行的时候，它的每个task可能都会从上一个stage的task所在的节点，去通过网络传输拉取需要自己处理的所有key，然后对拉取到的所有相同的key使用我们自己编写的算子函数执行聚合操作。**这个过程就是shuffle**。

# RDD依赖


Narrow Dependencies:
Wide Dependencies:
map, filter
union
join with inputs co-partitioned
groupByKey
join with inputs not co-partitioned

每个Stage包含多个组成pipeline的transformations（尽可能以narrow dependency形式）

## Schedule tasks

RDD.collect()

```
def collect(): Array[T] = withScope {
  val results = sc.runJob(this, (iter: Iterator[T]) => iter.toArray)
  Array.concat(results: _*)
}
```

DAGScheduler.runJob()

```
def runJob[T, U](
    rdd: RDD[T],
    func: (TaskContext, Iterator[T]) => U,
    partitions: Seq[Int],
    callSite: CallSite,
    resultHandler: (Int, U) => Unit,
    properties: Properties): Unit = {
  val start = System.nanoTime
  val waiter = submitJob(rdd, func, partitions, callSite, resultHandler, properties)

  val awaitPermission = null.asInstanceOf[scala.concurrent.CanAwait]
  waiter.completionFuture.ready(Duration.Inf)(awaitPermission)
  waiter.completionFuture.value.get match {
   ……
  }
}
```

# Schedule tasks

```
def submitJob[T, U](
    rdd: RDD[T],
    func: (TaskContext, Iterator[T]) => U,
    partitions: Seq[Int],
    callSite: CallSite,
    allowLocal: Boolean,
    resultHandler: (Int, U) => Unit,
    properties: Properties = null): JobWaiter[U] =
{
  val maxPartitions = rdd.partitions.length
  partitions.find(p => p >= maxPartitions || p < 0).foreach { p =>
    throw new IllegalArgumentException(
      "Attempting to access a non-existent partition: " + p + ". " +
        "Total number of partitions: " + maxPartitions)
  }

  val jobId = nextJobId.getAndIncrement()
  if (partitions.size == 0) {
    return new JobWaiter[U](this, jobId, 0, resultHandler)
  }

  assert(partitions.size > 0)
  val func2 = func.asInstanceOf[(TaskContext, Iterator[_]) => _]
  val waiter = new JobWaiter(this, jobId, partitions.size, resultHandler)
  eventProcessActor ! JobSubmitted(
    jobId, rdd, func2, partitions.toArray, allowLocal, callSite, waiter, properties)
  waiter
}
```

DAGScheduler.submitJob()

Spark 1.6版本前使用了AKKA，这里的eventProcessActor负责提交Job

# Schedule tasks

DAGScheduler.submitJob()

```
def submitJob[T, U](
    rdd: RDD[T],
    func: (TaskContext, Iterator[T]) => U,
    partitions: Seq[Int],
    callSite: CallSite,
    resultHandler: (Int, U) => Unit,
    properties: Properties): JobWaiter[U] = {
  val maxPartitions = rdd.partitions.length
  partitions.find(p => p >= maxPartitions || p < 0).foreach { p =>
    throw new IllegalArgumentException()
  }

  val jobId = nextJobId.getAndIncrement()
  if (partitions.size == 0) {
    return new JobWaiter[U](this, jobId, 0, resultHandler)
  }

  assert(partitions.size > 0)
  val func2 = func.asInstanceOf[(TaskContext, Iterator[_]) => _]
  val waiter = new JobWaiter(this, jobId, partitions.size, resultHandler)
  eventProcessLoop.post(JobSubmitted(
    jobId, rdd, func2, partitions.toArray, callSite, waiter,
    SerializationUtils.clone(properties)))
  waiter
}
```

Spark 1.6版本后不再使用AKKA，这里的eventProcessLoop类型为DAGSchedulerEventProcessLoop，它继承自EventLoop。

# Spark为何弃用AKKA

Spark actually used to depend on Akka. Unfortunately this brought in all of Akka's dependencies (in addition to Spark's already quite complex dependency graph) and, as Todd mentioned, led to conflicts with projects using both Spark and Akka.

It would probably be possible to use Akka and shade it to avoid conflicts (some additional classloading tricks may be required). However, considering that only a small portion of Akka's features was used and scoped quite narrowly across Spark, it isn't worth the extra maintenance burden. Furthermore, akka-remote uses Netty internally, so reducing dependencies to core functionality is a good thing IMO.

## Schedule tasks

```
private def doOnReceive(event: DAGSchedulerEvent): Unit = event match {
  case JobSubmitted(jobId, rdd, func, partitions, callSite, listener, properties) =>
    dagScheduler.handleJobSubmitted(jobId, rdd, func, partitions, callSite, listener, properties)

  case MapStageSubmitted(jobId, dependency, callSite, listener, properties) =>
    dagScheduler.handleMapStageSubmitted(jobId, dependency, callSite, listener, properties)

  case StageCancelled(stageId) =>
    dagScheduler.handleStageCancellation(stageId)

  case JobCancelled(jobId) =>
    dagScheduler.handleJobCancellation(jobId)

  case JobGroupCancelled(groupId) =>
    dagScheduler.handleJobGroupCancelled(groupId)
}
```

DAGSchedulerEventProcessLoop

# Schedule tasks

```
private[scheduler] def handleJobSubmitted(jobId: Int,
      finalRDD: RDD[_],
      func: (TaskContext, Iterator[_]) => _,
      partitions: Array[Int],
      callSite: CallSite,
      listener: JobListener,
      properties: Properties) {
    var finalStage: ResultStage = null
    finalStage = newResultStage(finalRDD, func, partitions, jobId, callSite)
    val job = new ActiveJob(jobId, finalStage, callSite, listener, properties)
    clearCacheLocs()

    val jobSubmissionTime = clock.getTimeMillis()
    jobIdToActiveJob(jobId) = job
    activeJobs += job
    finalStage.setActiveJob(job)
    val stageIds = jobIdToStageIds(jobId).toArray
    val stageInfos = stageIds.flatMap(id => stageIdToStage.get(id).map(_.latestInfo))
    listenerBus.post(
      SparkListenerJobStart(job.jobId, jobSubmissionTime, stageInfos, properties))
    submitStage(finalStage)
  }
```

## Schedule tasks

```
private def submitStage(stage: Stage) {
  val jobId = activeJobForStage(stage)
  if (jobId.isDefined) {
    logDebug("submitStage(" + stage + ")")
    if (!waitingStages(stage) && !runningStages(stage) && !failedStages(stage)) {
      val missing = getMissingParentStages(stage).sortBy(_.id)
      logDebug("missing: " + missing)
      if (missing.isEmpty) {
        logInfo("Submitting " + stage + " (" + stage.rdd + "), which has no missing parents")
        submitMissingTasks(stage, jobId.get)
      } else {
        for (parent <- missing) {
          submitStage(parent)
        }
        waitingStages += stage
      }
    }
  } else {
    abortStage(stage, "No active job for stage " + stage.id, None)
  }
}
```

这是一个逆向递归过程，先查找所有缺失的上级stage并提交，带所有上级Stage都提交执行了，才轮到执行当前stage对应的task。

# Schedule tasks

```
private def submitMissingTasks(stage: Stage, jobId: Int) {
  val tasks: Seq[Task[_]] = …

  if (tasks.size > 0) {
    stage.pendingPartitions ++= tasks.map(_.partitionId)
    logDebug("New pending partitions: " + stage.pendingPartitions)
    taskScheduler.submitTasks(new TaskSet(
      tasks.toArray, stage.id, stage.latestInfo.attemptId, jobId, properties))
    stage.latestInfo.submissionTime = Some(clock.getTimeMillis())
  } else {
    markStageAsFinished(stage, None)
    submitWaitingChildStages(stage)
  }
}
```

# RDD Transformations

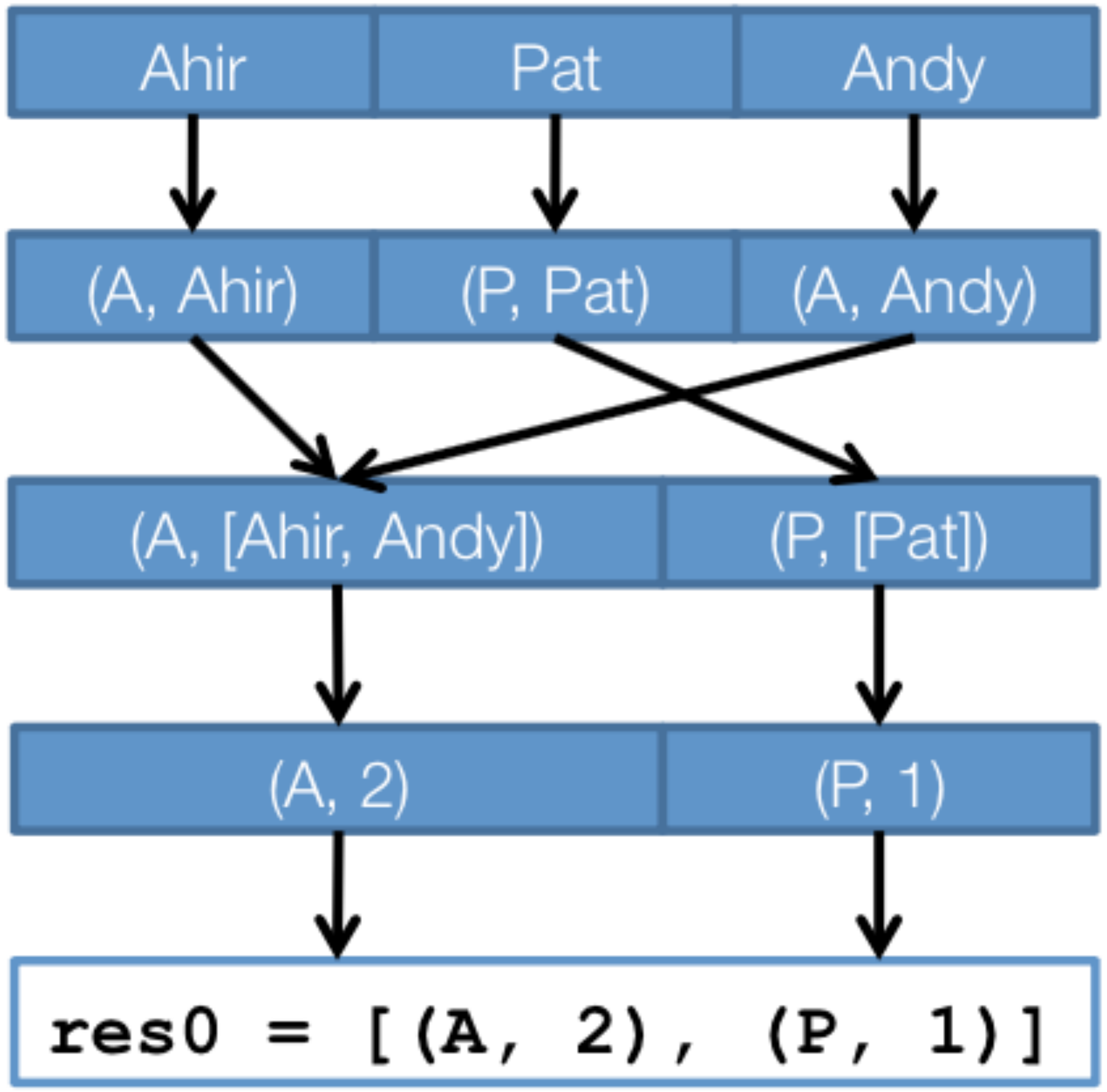
| Transformation | Meaning |
|---|---|
| **map**(*func*) | Return a new distributed dataset formed by passing each element of the source through a function *func*. |
| **filter**(*func*) | Return a new dataset formed by selecting those elements of the source on which *func* returns true. |
| **flatMap**(*func*) | Similar to map, but each input item can be mapped to 0 or more output items (so *func* should return a Seq rather than a single item). |
| **mapPartitions**(*func*) | Similar to map, but runs separately on each partition (block) of the RDD, so *func* must be of type Iterator<T> => Iterator<U> when running on an RDD of type T. |
| **mapPartitionsWithIndex**(*func*) | Similar to mapPartitions, but also provides *func* with an integer value representing the index of the partition, so*func* must be of type (Int, Iterator<T>) => Iterator<U> when running on an RDD of type T. |
| **sample**(*withReplacement*,*fraction*, *seed*) | Sample a fraction *fraction* of the data, with or without replacement, using a given random number generator seed. |
| **union**(*otherDataset*) | Return a new dataset that contains the union of the elements in the source dataset and the argument. |
| **intersection**(*otherDataset*) | Return a new RDD that contains the intersection of elements in the source dataset and the argument. |
| **distinct**([*numTasks*])) | Return a new dataset that contains the distinct elements of the source dataset. |
| **groupByKey**([*numTasks*]) | When called on a dataset of (K, V) pairs, returns a dataset of (K, Iterable<V>) pairs. |
| **reduceByKey**(*func*, [*numTasks*]) | When called on a dataset of (K, V) pairs, returns a dataset of (K, V) pairs where the values for each key are aggregated using the given reduce function *func*, which must be of type (V,V) => V. Like in `groupByKey`, the |
| **sortByKey**([*ascending*], [*numTasks*]) | When called on a dataset of (K, V) pairs where K implements Ordered, returns a dataset of (K, V) pairs sorted by keys in ascending or descending order, as specified in the boolean `ascending` argument. |
| **join**(*otherDataset*, [*numTasks*]) | When called on datasets of type (K, V) and (K, W), returns a dataset of (K, (V, W)) pairs with all pairs of elements for each key. Outer joins are also supported through `leftOuterJoin` and `rightOuterJoin`. |
| **cogroup**(*otherDataset*, [*numTasks*]) | When called on datasets of type (K, V) and (K, W), returns a dataset of (K, Iterable<V>, Iterable<W>) tuples. This operation is also called `groupWith`. |
| **cartesian**(*otherDataset*) | When called on datasets of types T and U, returns a dataset of (T, U) pairs (all pairs of elements). //相当于cross join |
| **pipe**(*command*, *[envVars]*) | Pipe each partition of the RDD through a shell command, e.g. a Perl or bash script. RDD elements are written to the process's stdin and lines output to its stdout are returned as an RDD of strings. |
| **coalesce**(*numPartitions*) | Decrease the number of partitions in the RDD to numPartitions. Useful for running operations more efficiently after filtering down a large dataset. |
| **repartition**(*numPartitions*) | Reshuffle the data in the RDD randomly to create either more or fewer partitions and balance it across them. This always shuffles all data over the network. |

# RDD Actions

| Action | Meaning |
|---|---|
| **reduce**(*func*) | Aggregate the elements of the dataset using a function*func* (which takes two arguments and returns one). The function should be commutative and associative so that it can be computed correctly in parallel. |
| **collect**() | Return all the elements of the dataset as an array at the driver program. This is usually useful after a filter or other operation that returns a sufficiently small subset of the data. |
| **count**() | Return the number of elements in the dataset. |
| **first**() | Return the first element of the dataset (similar to take(1)). |
| **take**(*n*) | Return an array with the first *n* elements of the dataset. Note that this is currently not executed in parallel. Instead, the driver program computes all the elements. |
| **takeSample**(*withReplacement*,*num*, *seed*) | Return an array with a random sample of *num* elements of the dataset, with or without replacement, using the given random number generator seed. |
| **takeOrdered**(*n*, *[ordering]*) | Return the first *n* elements of the RDD using either their natural order or a custom comparator. |
| **saveAsTextFile**(*path*) | Write the elements of the dataset as a text file (or set of text files) in a given directory in the local filesystem, HDFS or any other Hadoop-supported file system. Spark will call toString on each element to convert it to a line of text in the file. |
| **saveAsSequenceFile**(*path*)<br>(Java and Scala) | Write the elements of the dataset as a Hadoop SequenceFile in a given path in the local filesystem, HDFS or any other Hadoop-supported file system. This is available on RDDs of key-value pairs that either implement Hadoop's Writable interface. In Scala, it is also available on types that are implicitly convertible to Writable (Spark includes conversions for basic types like Int, Double, String, etc) |
| **saveAsObjectFile**(*path*)<br>(Java and Scala) | Write the elements of the dataset in a simple format using Java serialization, which can then be loaded using`SparkContext.objectFile()`. |
| **countByKey**() | Only available on RDDs of type (K, V). Returns a hashmap of (K, Int) pairs with the count of each key. |
| **foreach**(*func*) | Run a function *func* on each element of the dataset. This is usually done for side effects such as updating an accumulator variable (see below) or interacting with external storage systems. |

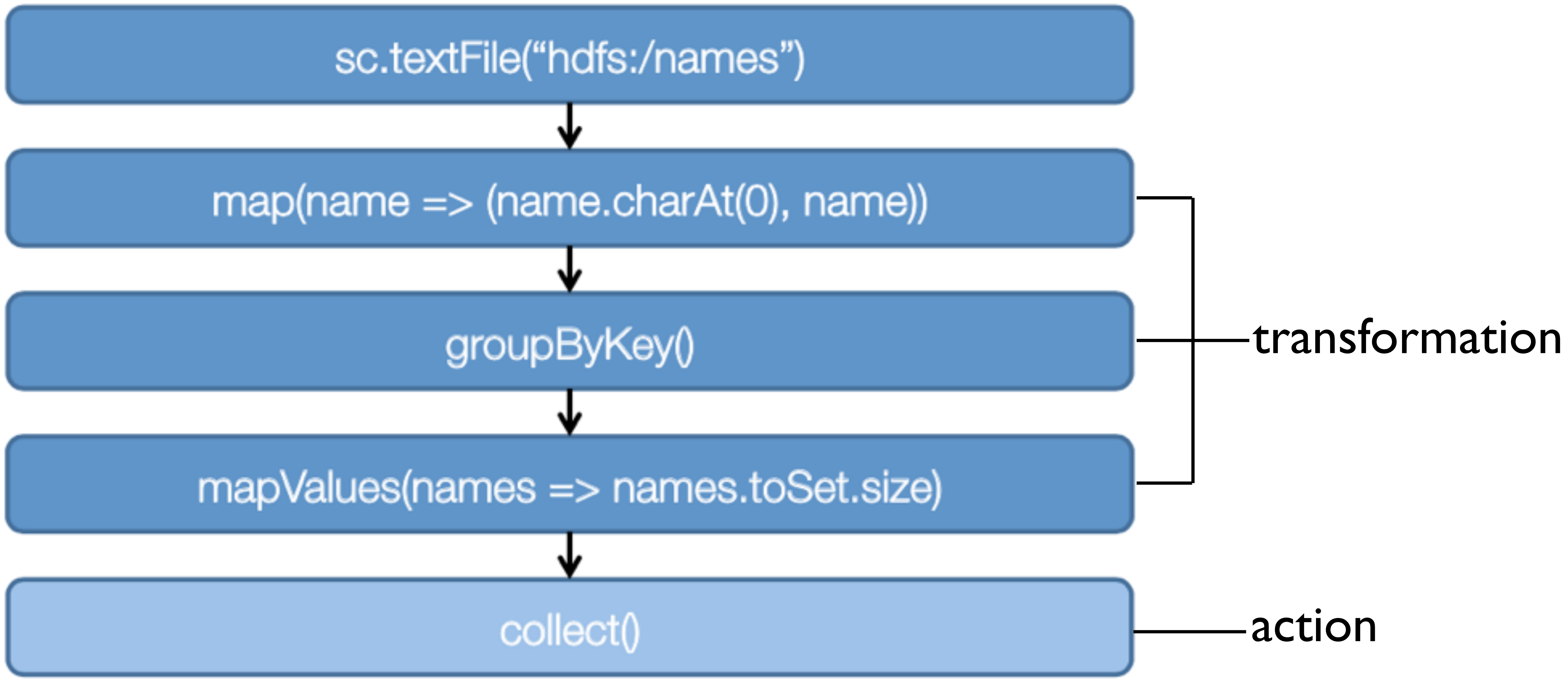
# 案例： 字数统计

```
sc.textFile("hdfs:/names")
    .map(name => (name.charAt(0), name))
    .groupByKey()
    .mapValues(names => names.toSet.size)
    .collect()
```

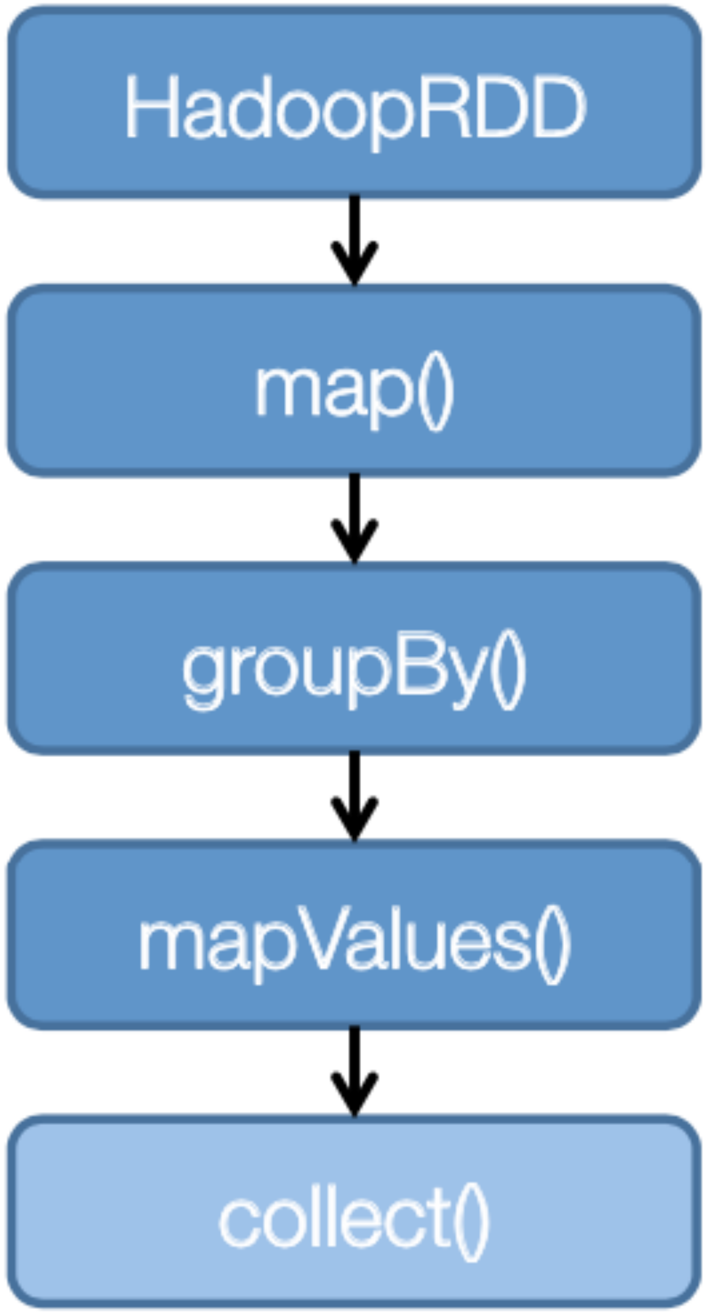


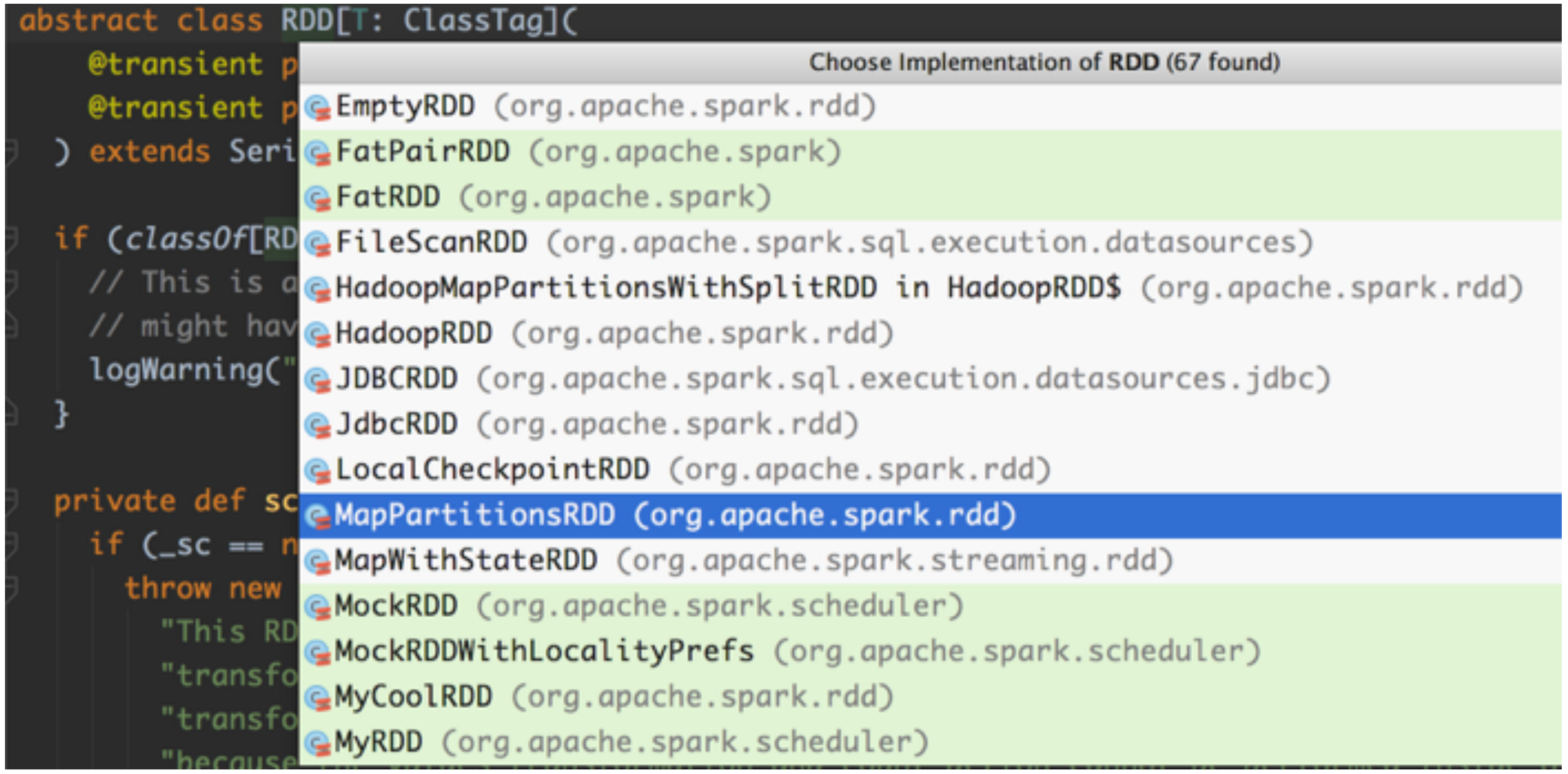
```
res0 = [(A, 2), (P, 1)]
```

# 步骤一：创建RDD


sc.textFile("hdfs:/names")
map(name => (name.charAt(0), name))
groupByKey()
mapValues(names => names.toSet.size)
collect()
transformation
action

```
def textFile(
    path: String,
    minPartitions: Int = defaultMinPartitions): RDD[String] = withScope {
  assertNotStopped()
  hadoopFile(path, classOf[TextInputFormat], classOf[LongWritable], classOf[Text],
    minPartitions).map(pair => pair._2.toString).setName(path)
}
def hadoopFile[K, V](
    path: String,
    inputFormatClass: Class[_ <: InputFormat[K, V]],
    keyClass: Class[K],
    valueClass: Class[V],
    minPartitions: Int = defaultMinPartitions): RDD[(K, V)] = withScope {
  assertNotStopped()
  // A Hadoop configuration can be about 10 KB, which is pretty big, so broadcast it.
  val confBroadcast = broadcast(new SerializableConfiguration(hadoopConfiguration))
  val setInputPathsFunc = (jobConf: JobConf) => FileInputFormat.setInputPaths(jobConf, path)
  new HadoopRDD(
    this,
    confBroadcast,
    Some(setInputPathsFunc),
    inputFormatClass,
    keyClass,
    valueClass,
    minPartitions).setName(path)
}
```



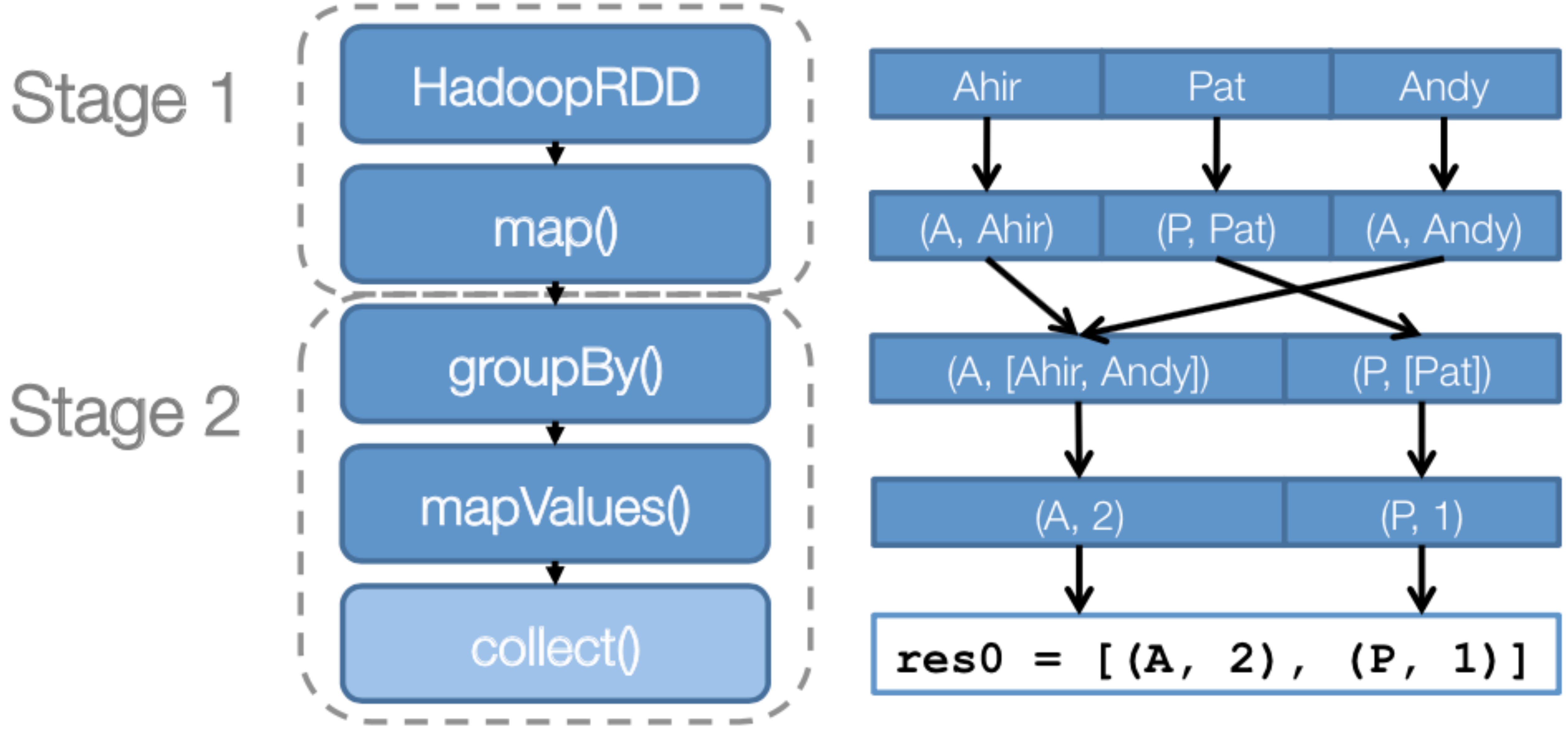


```
abstract class RDD[T: ClassTag](
    @transient p
    @transient p
  ) extends Seri

  if (classOf[RD
    // This is a
    // might hav
    logWarning("
  }

  private def sc
    if (_sc == n
      throw new
        "This RD
        "transfo
        "transfo
        "because
```

Choose Implementation of RDD (67 found)

```
EmptyRDD (org.apache.spark.rdd)
FatPairRDD (org.apache.spark)
FatRDD (org.apache.spark)
FileScanRDD (org.apache.spark.sql.execution.datasources)
HadoopMapPartitionsWithSplitRDD in HadoopRDD$ (org.apache.spark.rdd)
HadoopRDD (org.apache.spark.rdd)
JDBCRDD (org.apache.spark.sql.execution.datasources.jdbc)
JdbcRDD (org.apache.spark.rdd)
LocalCheckpointRDD (org.apache.spark.rdd)
MapPartitionsRDD (org.apache.spark.rdd)
MapWithStateRDD (org.apache.spark.streaming.rdd)
MockRDD (org.apache.spark.scheduler)
MockRDDWithLocalityPrefs (org.apache.spark.scheduler)
MyCoolRDD (org.apache.spark.rdd)
MyRDD (org.apache.spark.scheduler)
```

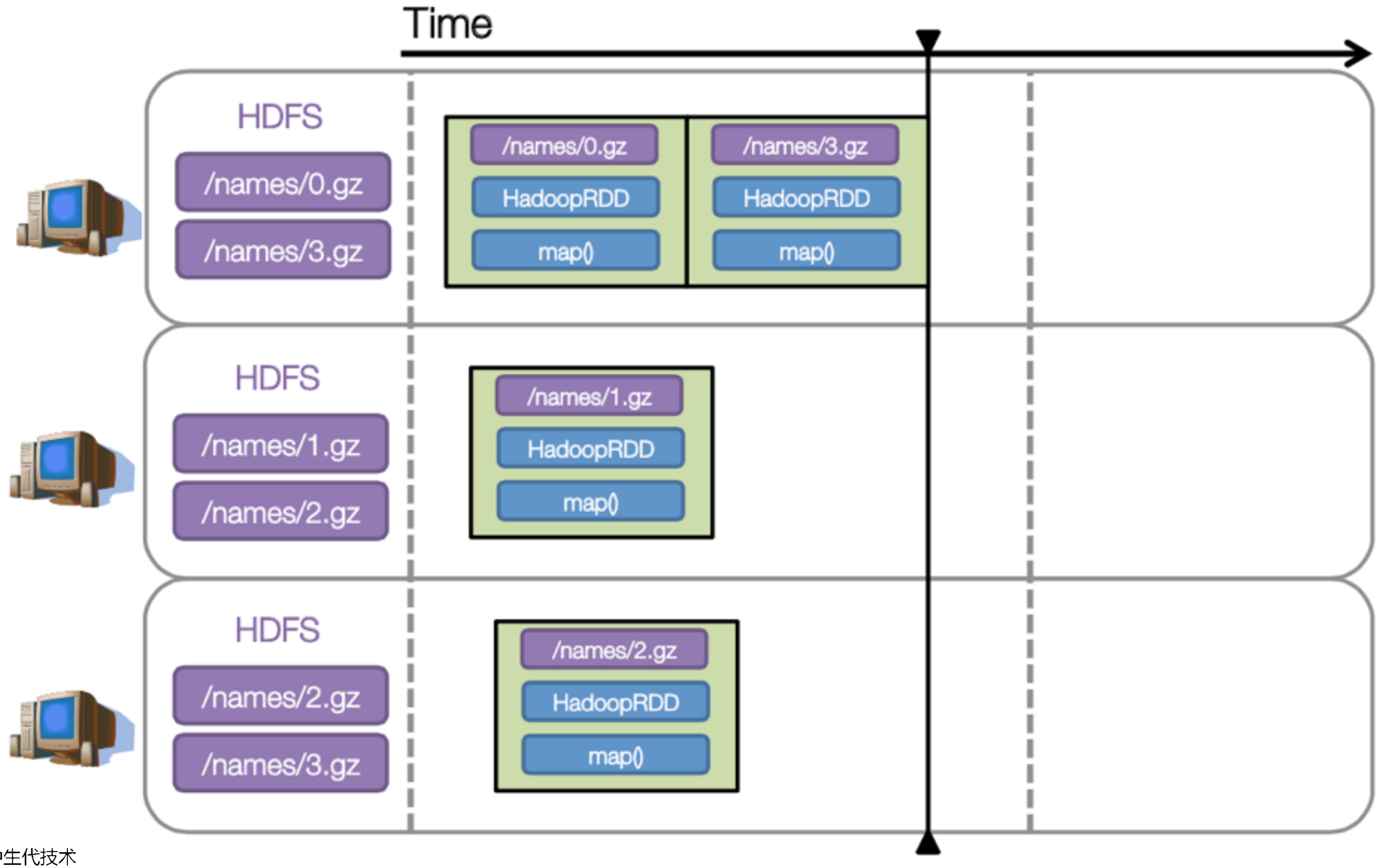
# 步骤二：创建执行计划



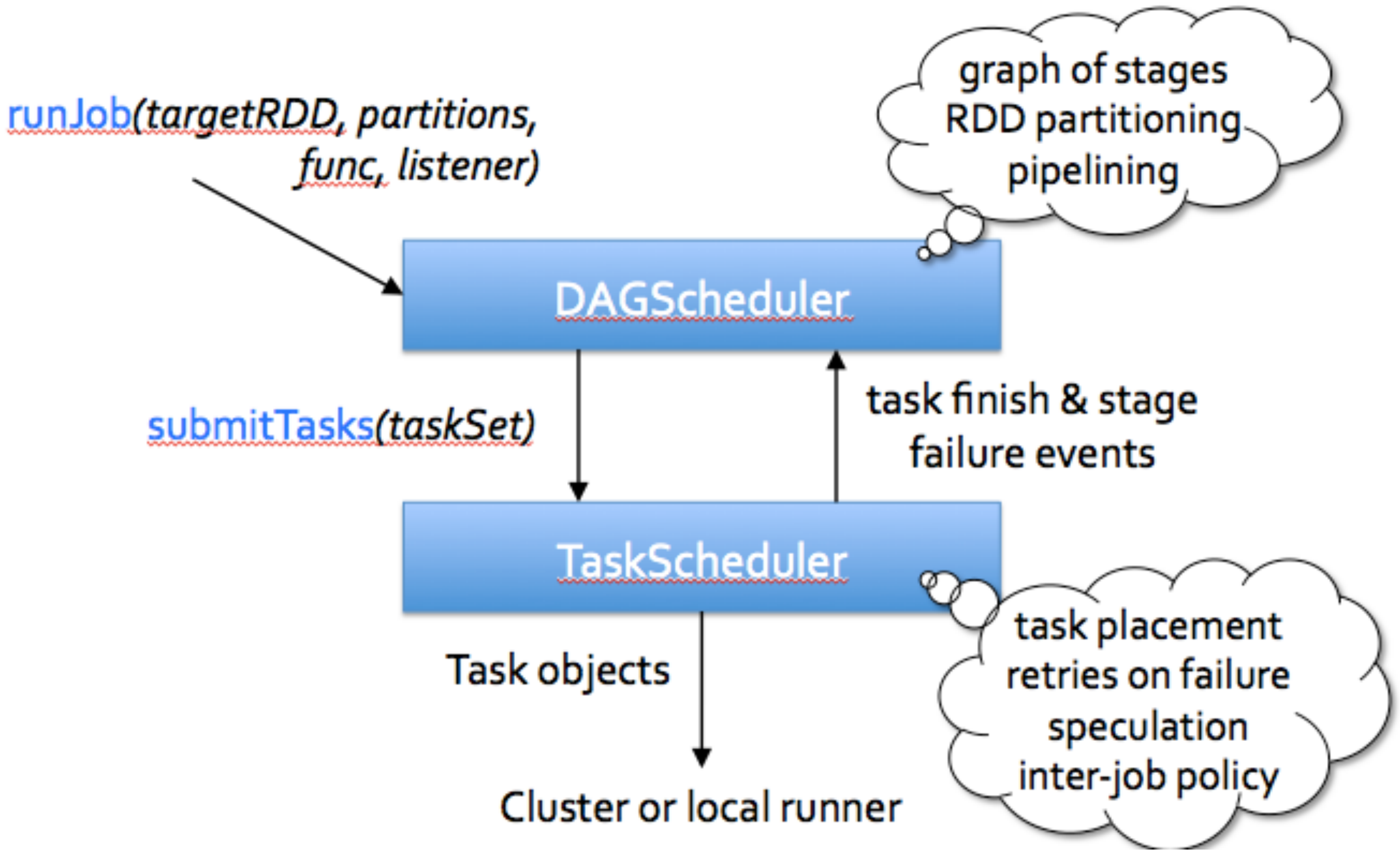
```
res0 = [(A, 2), (P, 1)]
```

groupBy()导致shuffle，形成shuffle dependency(wide dependency)，因而被划分为两个stage。

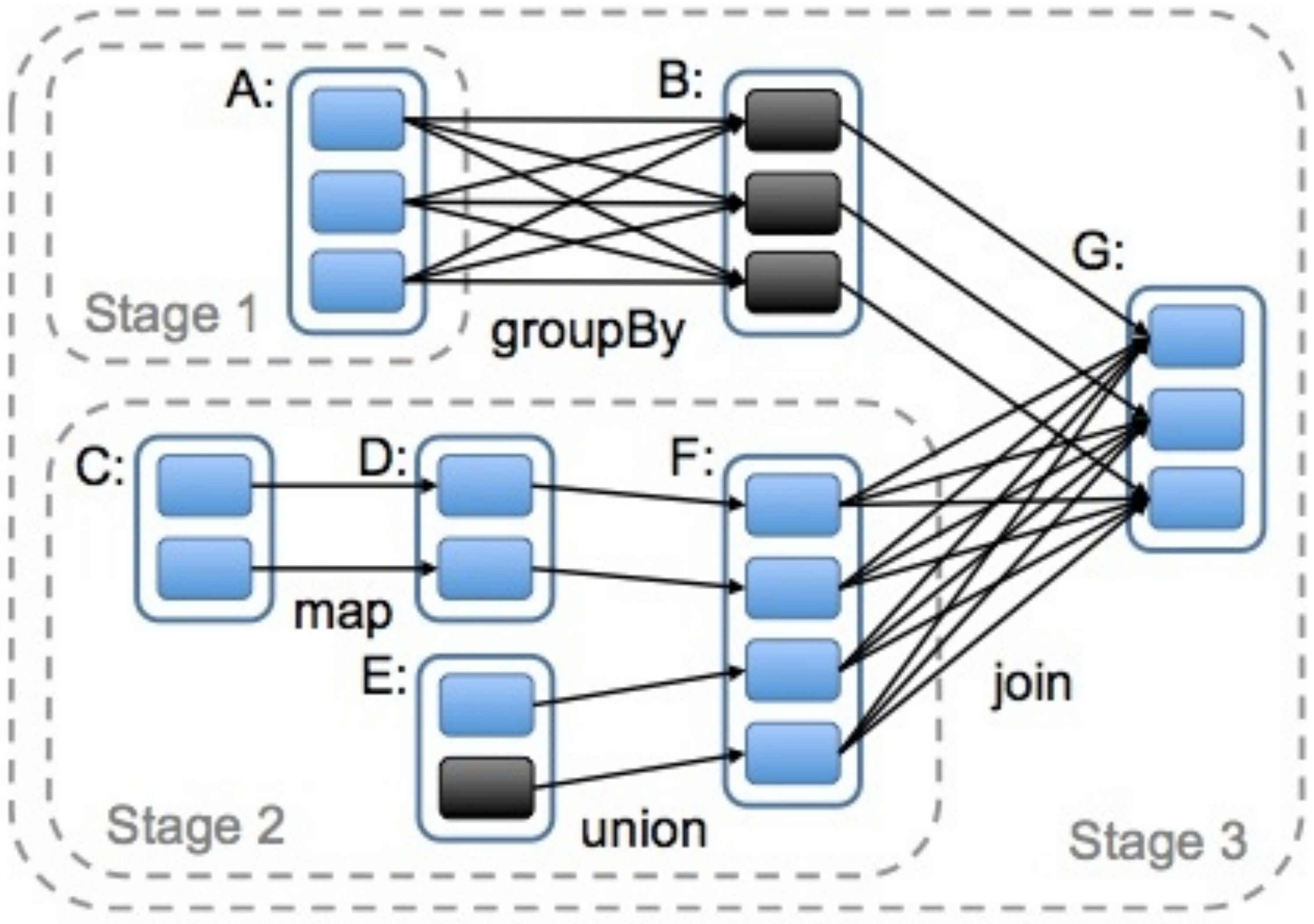
# 步骤三： Schedule tasks



## 步骤三：Schedule tasks



# 步骤三：Schedule tasks



partitions -> stages -> tasks

## 如何改进一下代码？

```
sc.textFile("hdfs:/names")
   .map(name => (name.charAt(0), name))
   .groupByKey()
   .mapValues(names => names.toSet.size)
   .collect()
```

# 如何改进一下代码?

```
sc.textFile("hdfs:/names")
    .map(name => (name.charAt(0), name))
    .groupByKey()
    .mapValues(names => names.toSet.size)
    .collect()
```

```
sc.textFile("hdfs:/names")
    .distinct(numPartitions = 6)
    .map(name => (name.charAt(0), 1))
    .reduceByKey(_ + _)
    .collect()
```

中生代技术
FRESHMAN TECHNOLOGY

## 如何改进一下代码？

```
sc.textFile("hdfs:/names")
   .map(name => (name.charAt(0), name))
   .groupByKey()
   .mapValues(names => names.toSet.size)
   .collect()
```

```
sc.textFile("hdfs:/names")
   .distinct(numPartitions = 6)
   .map(name => (name.charAt(0), 1))
   .reduceByKey(_ + _)
   .collect()
```

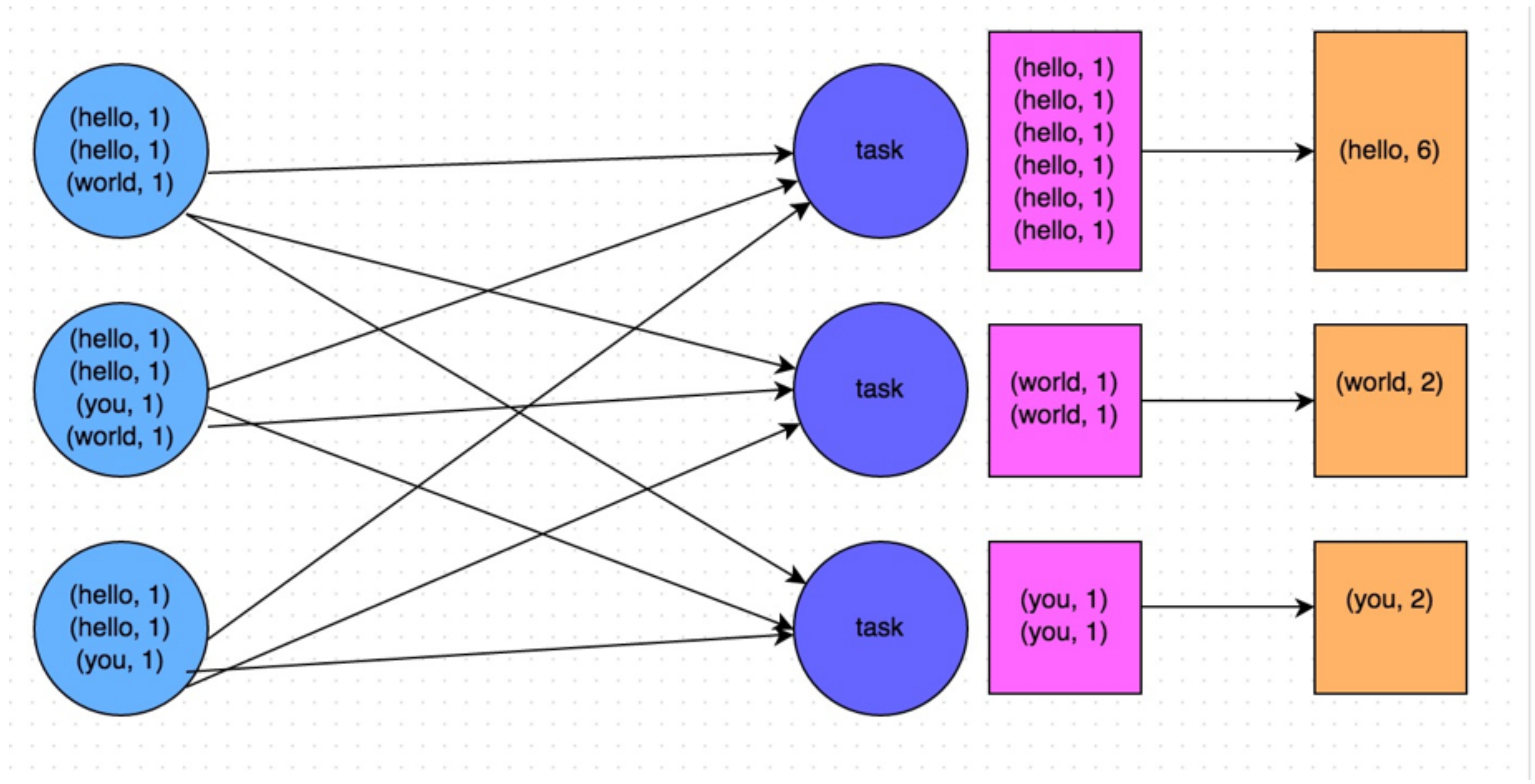
- 为并发执行确保足够的分区
- 最少的内存消耗 (groupBys和sorting)
- 减少或避免数据的shuffle
- 合理地使用标准库提供的API

## 为什么使用reduceByKey?

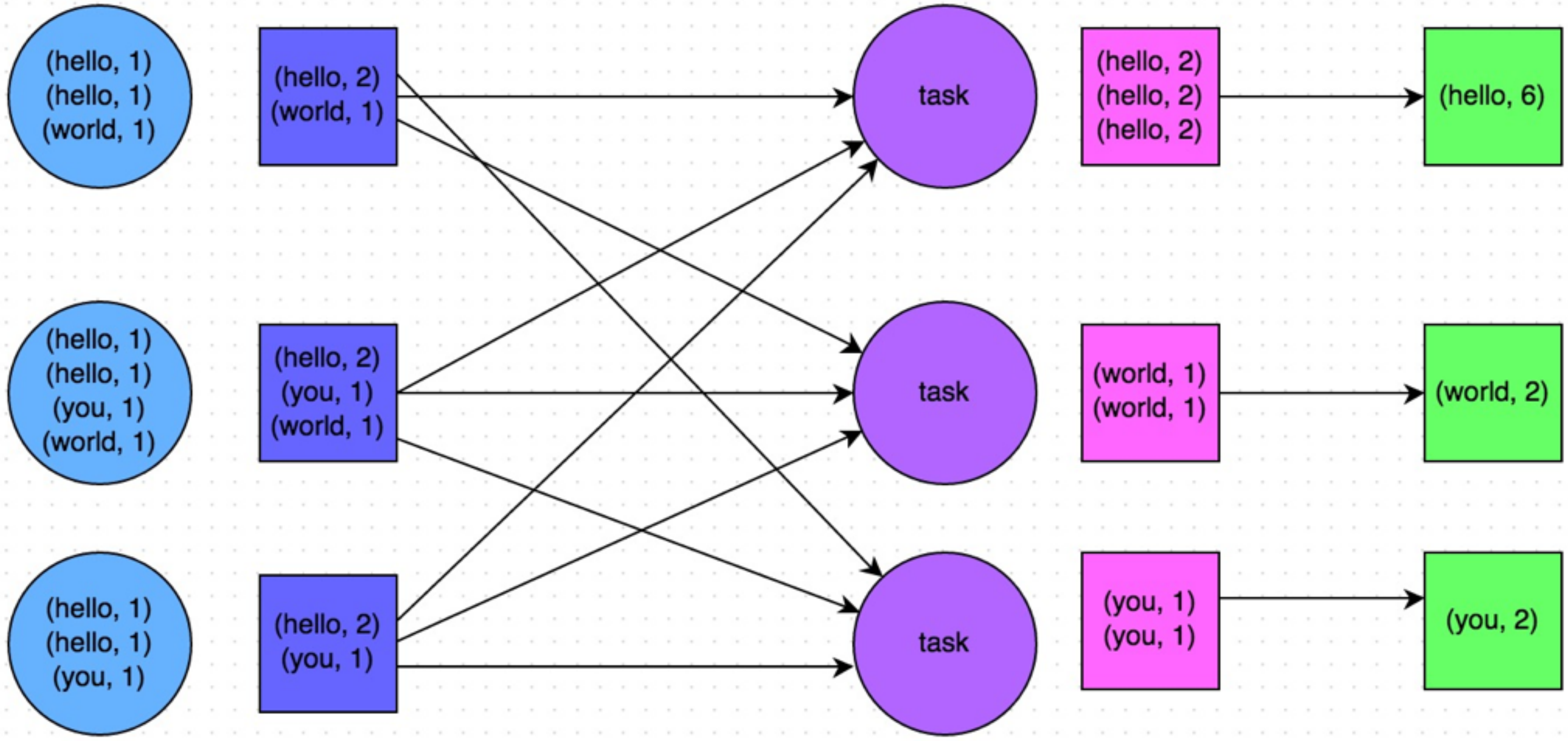
reduceByKey和aggregateByKey算子都会使用用户自定义的函数对每个节点本地的相同key进行预聚合。

groupByKey算子则不会进行预聚合，全量的数据会在集群的各个节点之间分发和传输，性能相对来说比较差。

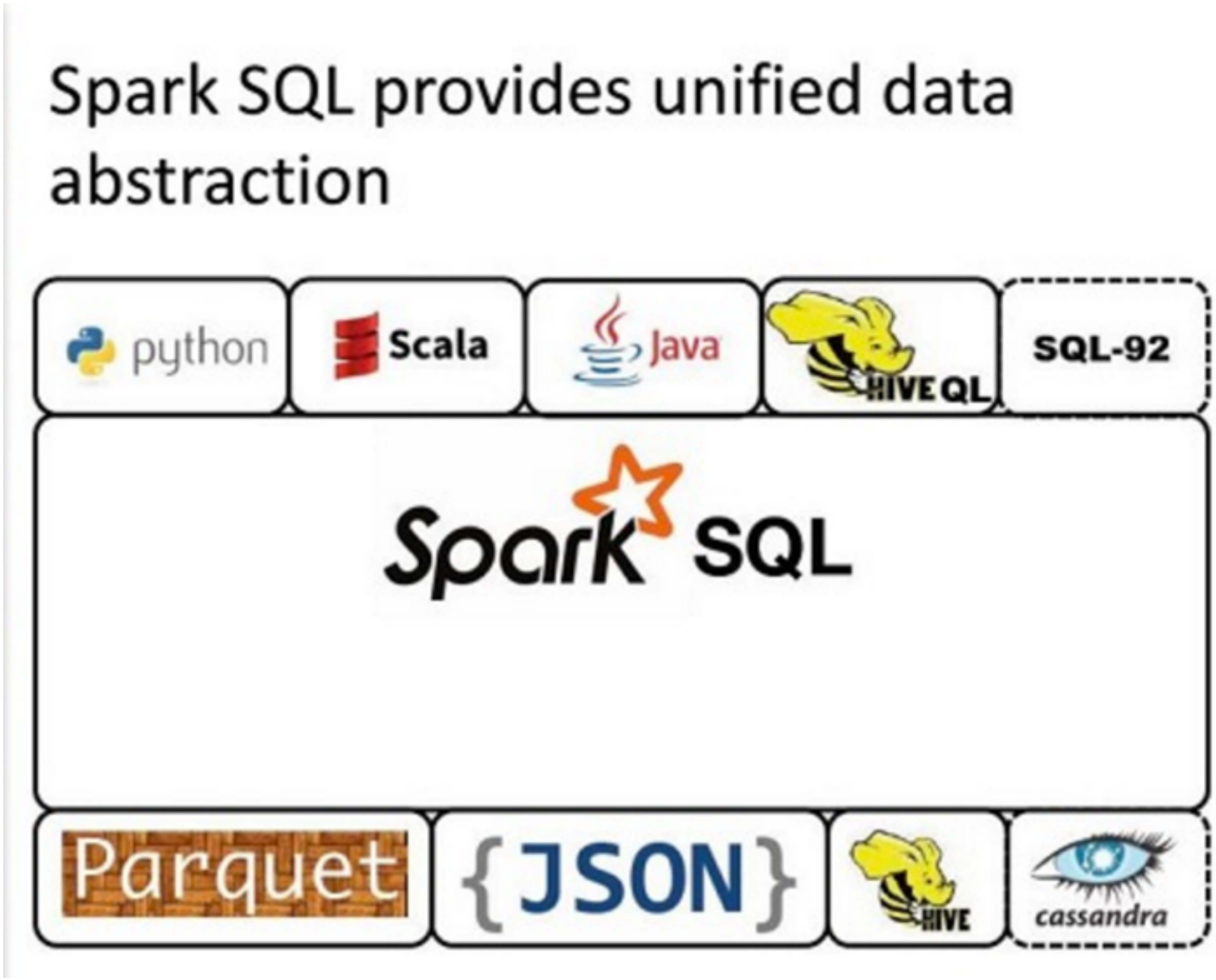
# 使用groupByKey的情况



# 使用reduceByKey的情况

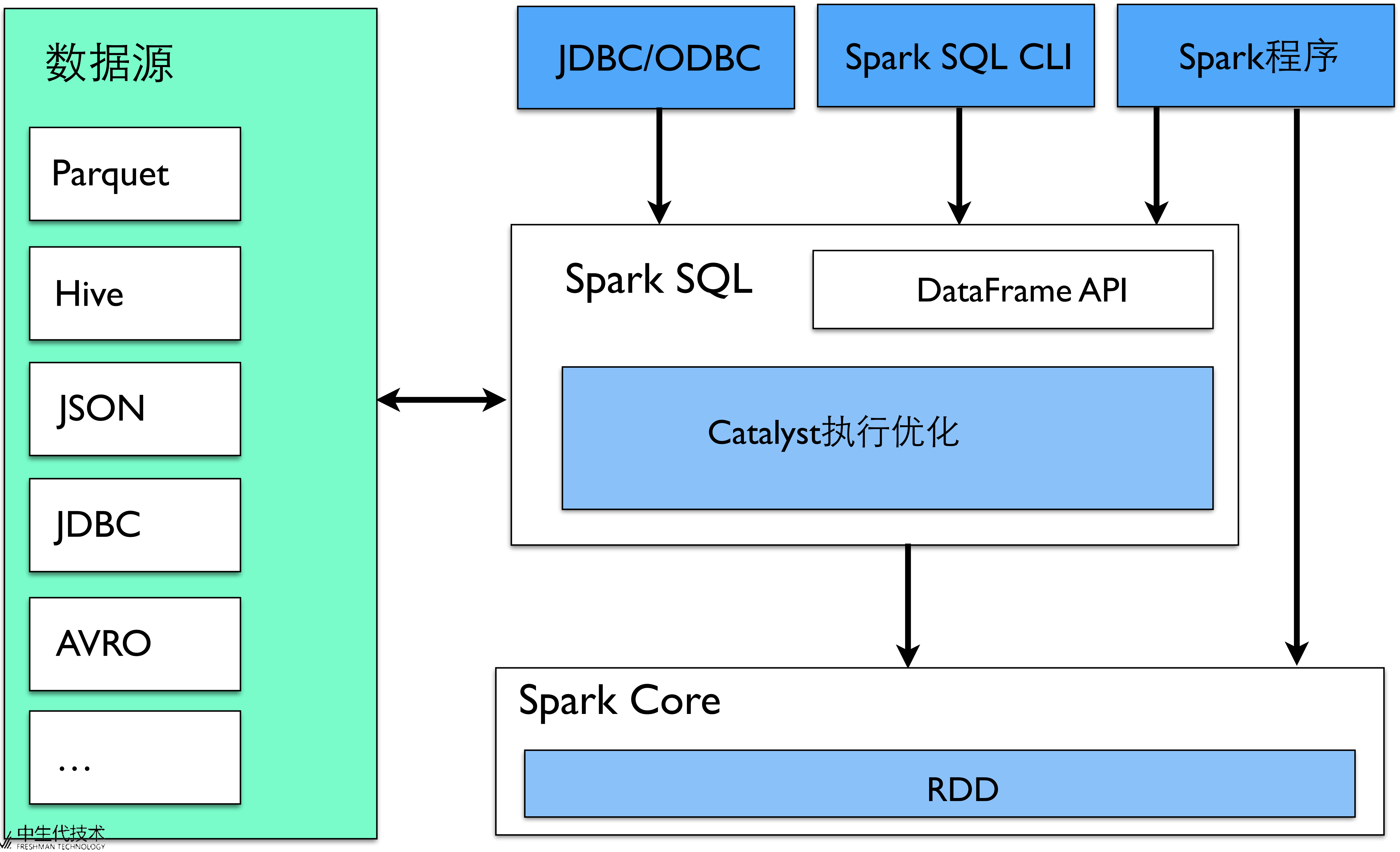


# Spark SQL

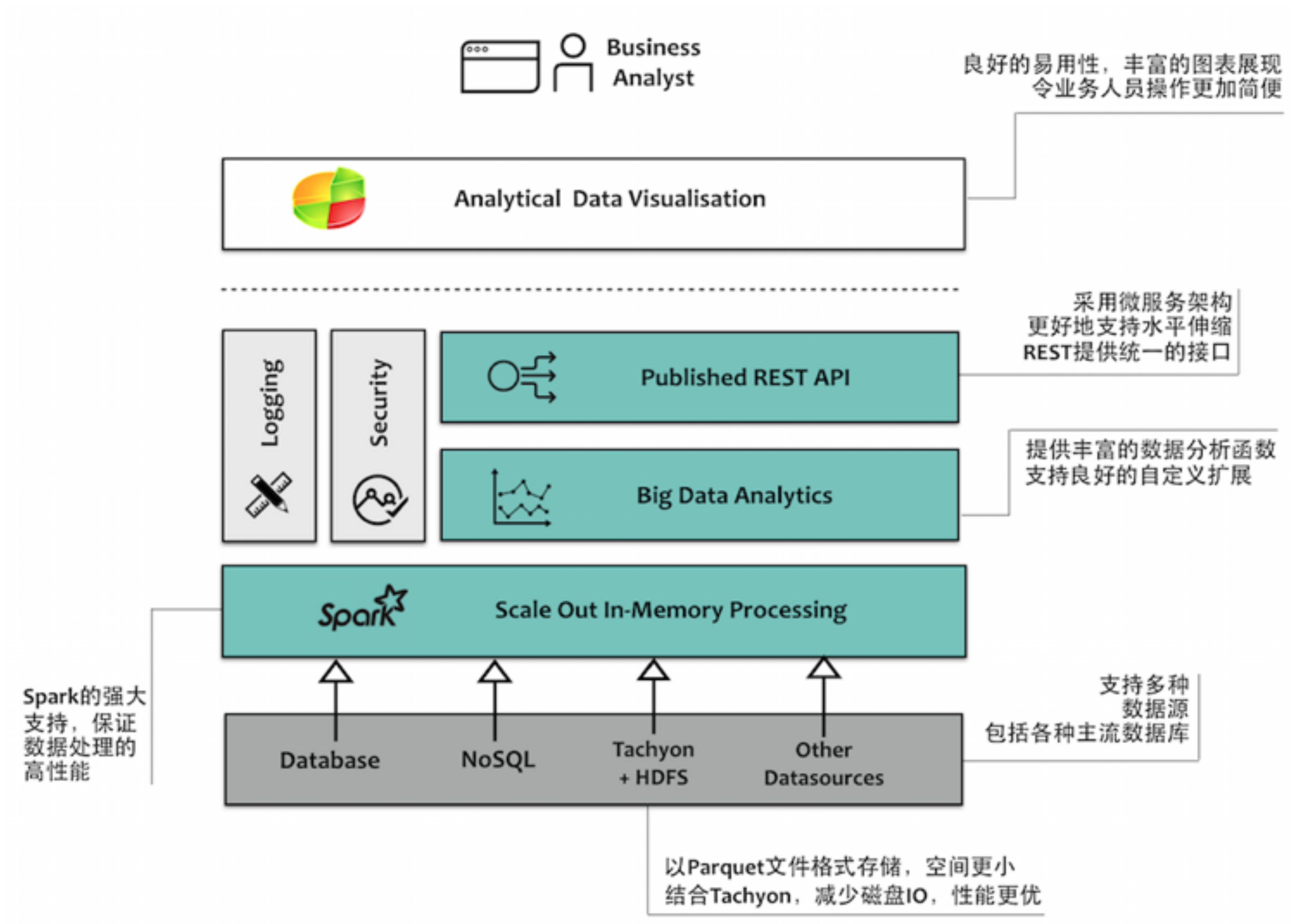


Spark SQL provides unified data abstraction

# Spark SQL整体架构



# 我们产品的架构



中生代技术 FRESHMAN TECHNOLOGY

## 我们使用Spark/Spark SQL的方式

# 启动Spark SQL执行分析

```
object GlobalContext extends DataSetConfiguration {

  implicit class SparkContextImplicits(sparkContext: SparkContext) {
    def setHadoopConfiguration(key: String, value: String): SparkContext = {
      sparkContext.hadoopConfiguration.set(key, value)
      sparkContext
    }
  }

  val sparkContext = new SparkContext(new SparkConf().setAll(sparkConfig))
    .setHadoopConfiguration("fs.hdfs.impl", classOf[DistributedFileSystem].getName)
    .setHadoopConfiguration("fs.file.impl", classOf[LocalFileSystem].getName)

  def initSparkContext = sparkContext
}
```

1. 定义一个全局的SparkContext

# 启动Spark SQL执行分析

```
trait SparkSqlSupport extends UdfRegistrationSupport with MortLogger {
  self =>

  implicit lazy val sqlContext = newSqlContext

  def load(tableName: String)(implicit sqlContext: SQLContext): DataFrame

  def load(tableName: String, dataSetId: Option[ID])(implicit sqlContext: SQLContext): DataFrame
    registerDynamicFunctions(dataSetId)
    load(tableName)
  }

  def newSqlContext: SQLContext = newSqlContext(registerUdf)

  def newSqlContext(block: => SQLContext => Any): SQLContext =
    new SQLContext(GlobalContext.sparkContext) {
      self =>
      block(self)
    }
}
```

2. 加载SQLContext

# 启动Spark SQL执行分析

```
trait SparkSqlSupport extends UdfRegistrationSupport with MortLogger {
  self =>

   def executeAnalysis(tableName: String,
                       statement: String, dataSetId: Option[ID] = None,
                       dynamicFunctions: List[DynamicFunction] = Nil)
(implicit sqlContext: SQLContext): DataFrame = {
        registerDynamicFunctions(dataSetId, dynamicFunctions)

        val dataFrame = load(tableName)(sqlContext)
        dataFrame.registerTempTable(tableName)
        sqlContext.sql(normalize(statement)(dataFrame))
    }

  }
```

3. 执行分析的核心方法

# 启动Spark SQL执行分析

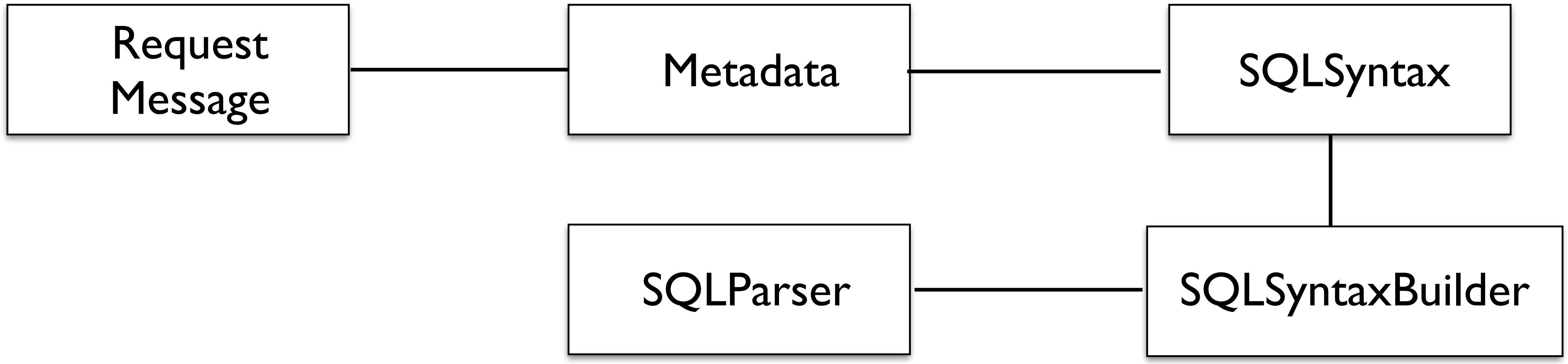
```
trait DataAnalysisActor
  extends Actor
  with DisposableActor
  with ActorExceptionHandler
  with SparkSqlSupport
  with CostTimeLogger {

  override def mortReceive: Receive = {
    case AnalysisExecutionContext(tableName, sql, dataSetId) =>
      val analysisResult: AnalysisResult = logTimeConsuming("analysis execution time")(log.info) {
        executeAnalysis(tableName, sql, dataSetId).toAnalysisResult
      }

      sender ! ExecutionSuccess(analysisResult)
  }
}
```

4. 通过Actor执行分析

# 使用Spark SQL提供的UDF与UDAF

```
trait UdfRegistrationSupport {

  type UDAF = UserDefinedAggregateFunction

  def registerUdf(sqlContext: SQLContext) = {

    UdfContext.udfs.udfs.filter(_.function.nonEmpty).foreach { implicit udf =>
      udf.udfType match {
        case FunctionType.UDAF =>
          udfInstanceOf[UDAF].foreach {
            sqlContext.udf.register(udf.name, _)
          }

        case _ =>
      }
    }

    sqlContext.udf.register("add", add)
  }
}
```

# 对分析元数据进行语法树解析



```
object SQLParser extends SQLSyntaxBuilder {
  def parse(implicit metaList: List[SQLMeta]): String = {
    withSQL {
      select(fields, from, join, where, groupBy, orderBy)
    }
  }
  def withSQL(statement: => SelectClauseSyntax): String = {
    try {
      statement.evaluate
    } catch {
      case ex: ResourceNotExistException => throw new BadRequestException(ex.message)
      case ex: BadRequestException => throw ex
      case ex: Exception => throw new BadRequestException(ex.getMessage)
    }
  }
}
```

扫描二维码或搜索「逸言」
关注微信公众号