Programming Spark

Spark概述

Hadoop的演进

`Google`发表的三篇论文，开启了`Hadoop 1x`时代的。

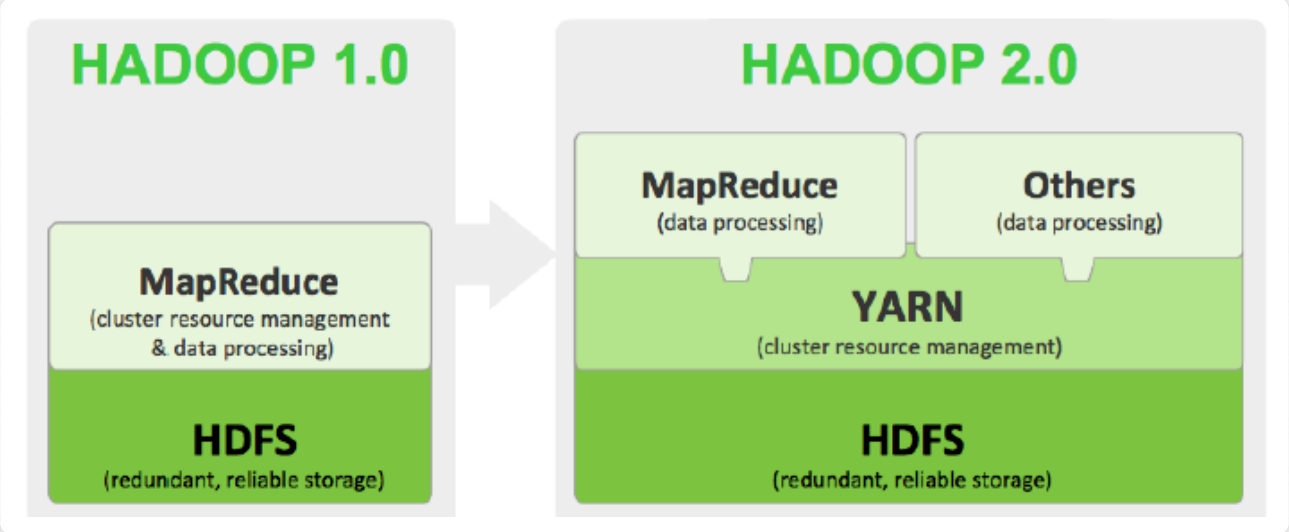
- MapReduce
- GFS
- BigTable

Hadoop 1.0

- MRv1
- JobTracker职责过重
- 缺陷：可扩展性，资源利用率，多框架支持

Hadoop 2.0

- MRv2：解耦
- Yarn

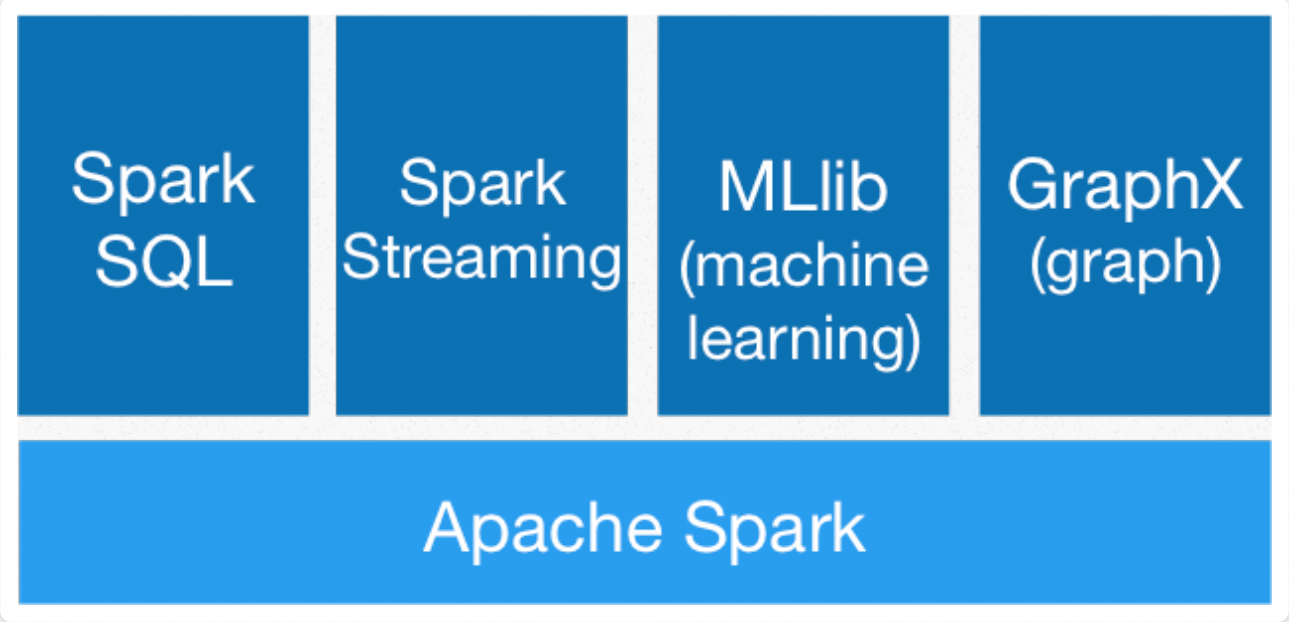


What's Spark?

Apache Spark™ is a fast and general engine for large-scale data processing.

- Matei Zaharia
- UC Berkeley, AMPLab
- http://spark.apache.org

Spark技术栈



Spark优势

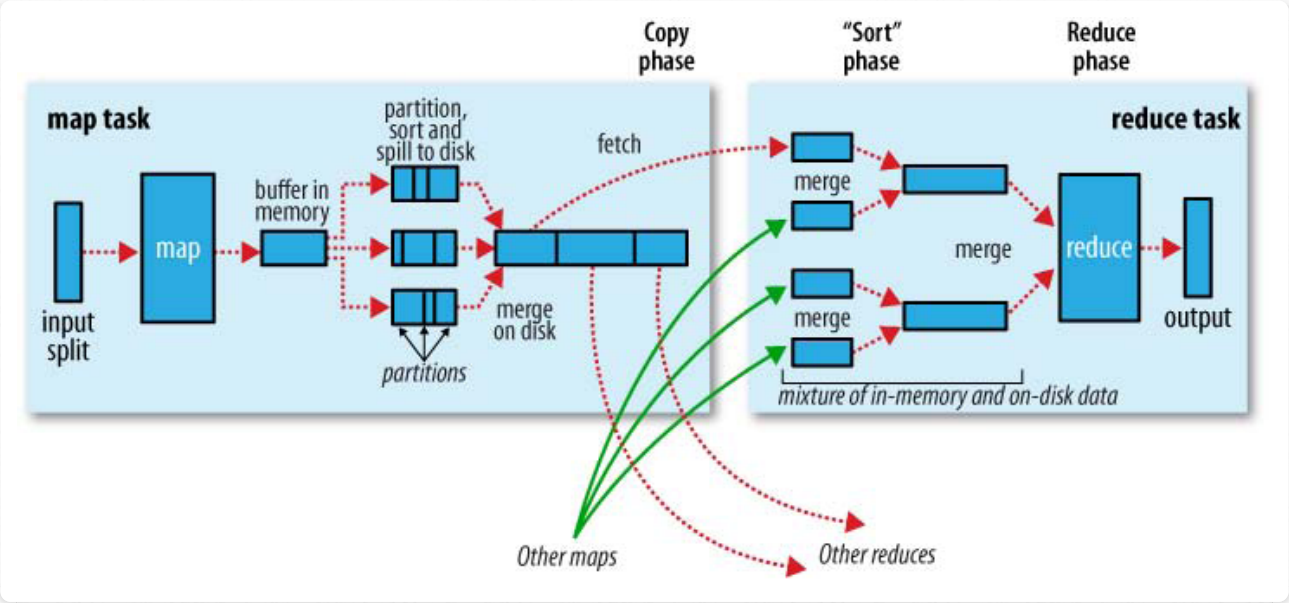
高性能

Run programs up to 100x faster than Hadoop MapReduce in memory, or 10x faster on disk.

- 基于内存的迭代式计算
- 基于`DAG`的执行引擎
- 基于`RDD`的统一抽象模型
- 自动容错机制

MapReduce

- 离线批处理
- 时效性差
- 磁盘读写



易学易用

- 多语言(`Sacla, Java, Python, R`)
- 函数式

```
sc.textFile("hdfs://...").flatMap(_.split(" ")).map((_, 1)).reduceByKey
(_+_)
```

MapReduce

```
class TokenizerMapper extends Mapper<Object, Text, Text, IntWritable> {
  private final static IntWritable one = new IntWritable(1);
  private Text word = new Text();

  public void map(Object key, Text value, Context context)
    throws IOException, InterruptedException {
    StringTokenizer itr = new StringTokenizer(value.toString());
    while (itr.hasMoreTokens()) {
      word.set(itr.nextToken());
      context.write(word, one);
    }
  }
}

class IntSumReducer extends Reducer<Text, IntWritable, Text, IntWritabl
e> {
  private IntWritable result = new IntWritable();

  public void reduce(Text key, Iterable<IntWritable> values, Context co
ntext)
    throws IOException, InterruptedException {
```

```
      int sum = 0;
      for (IntWritable val : values) {
        sum += val.get();
      }
      result.set(sum);
      context.write(key, result);
    }
  }
```

通用

`One Stack to rule them all`构建统一的技术栈。

- `Spark SQL`
- `Spark Streaming`
- `Spark MLlib`
- `Spark GraphX`

近乎完美解决了大数据中三大核心问题。

- `Batch Processing`
- `Streaming Processing`
- `Ad-hoc Query`

易集成，易部署

支持多种部署方式：

- `Local`
- `Standalone`
- `Yarn`
- `Mesos`
- `EC2`

支持多种外部数据源：

- `HDFS`
- `HBase`
- `Hive`
- `Cassandra`
- `S3`

Spark编程模型

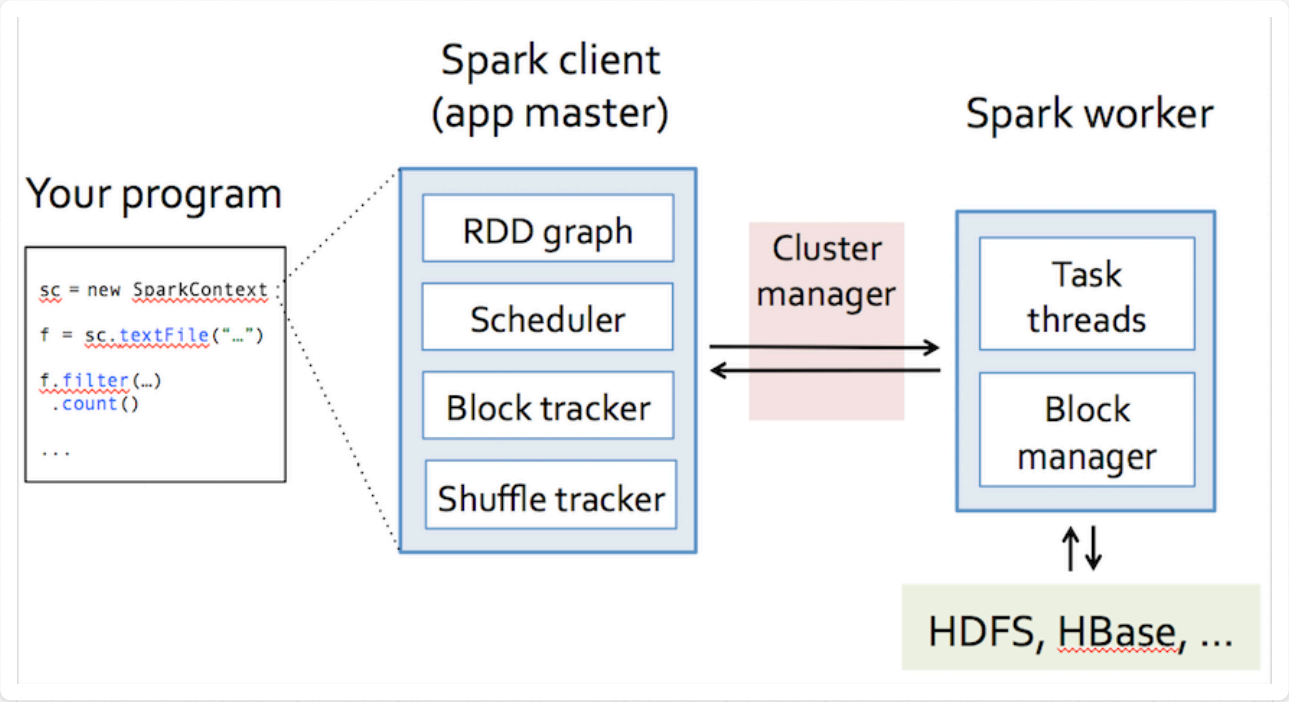
概念

- `Application: [Driver Program] + [Executor]+`
- `Driver Program`: 运行`Application`的`mian`函数，负责创建`SparkContext`；
- `SparkContext`: 构建应用程序的运行环境，负责与`Cluster Maanger`通信，进行资源的申请，任务的分配和调度;
- `Executor`: 运行在`Worker`上的一个进程(`CoarseGrainedExecutorBackend`)，负责执行`Task`；
- `Worker`: 运行一个或多个`Executor`
- `Cluster Manager`： 获取资源的外部服务，`Standalone，Yarn，Mesos`；
- `RDD`: 弹性分布式数据集
- `Job`： `SparkContext`提交的`Action`操作;
- `Stage`: `DAGScheduler`根据`Job`构建基于`Stage(TaskSet)`的`DAG`，并提交给`TaskScheduler`；
- `TaskScheduler`： 将`Taskset`提交给`Worker Node`集群中运行，对`Task`进行调度和管理，并将其分配在合适的`Executor`上执行;
- `DAGScheduler`: 根据`Job`中`RDD`的依赖关系构建出`Stage`的`DAG`，并以此提交`Stage`到`TaskSchuduler`；

组件

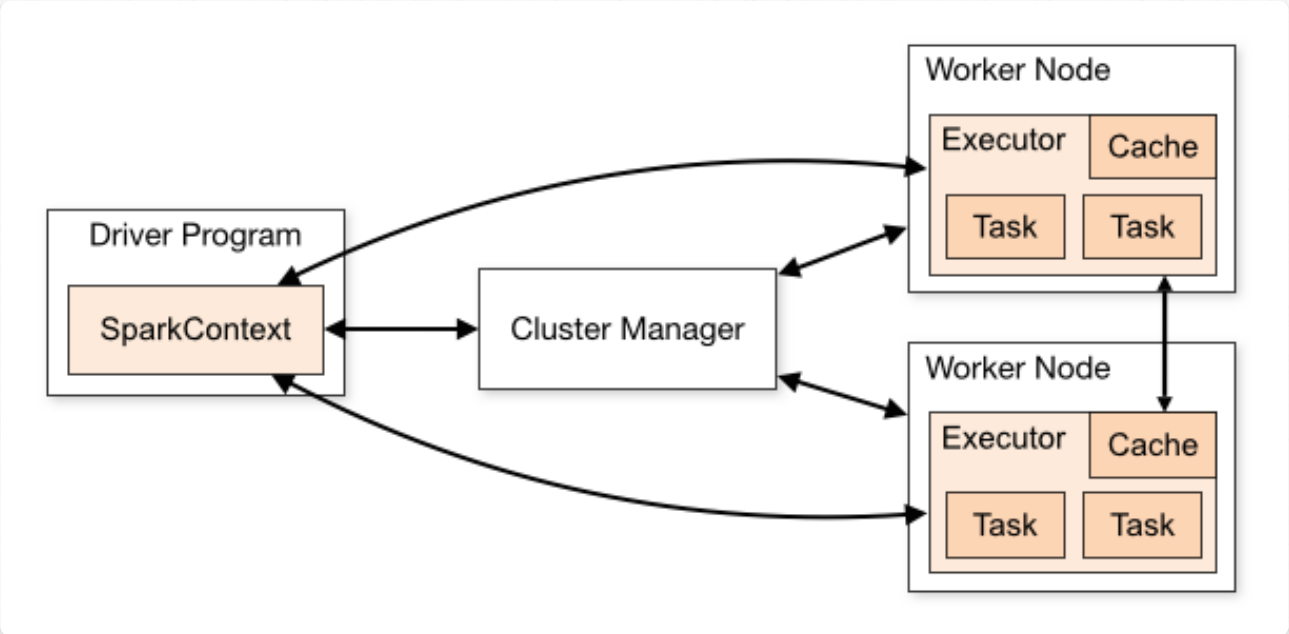
`Program Driver`是`Spark`的核心所在，主要包括三个最重要的组件:

- `RDD`
- `DAGScheduler`
- `TaskScheduler`



部署方式

- `Standalne`
- `Yarn`
- `Mesos`
- `Cloud`



RDD创建方式

- 外部数据集;

```
val lines = ctxt.textFile("hdfs://...")
```

- 并发数据集

```
val words = ctxt.parallelize(List("mapreduce", "spark"))
```

- 由其他`RDD`转换而来

```
val words = lines.map(_.split(" "))
```

RDD编程接口

- `Transformation`: 变换为其它的`RDD`，惰性求值(`Lazy Evaluation`)

```
class RDD[T: ClassTag](
    private var _sc: SparkContext,
    private var deps: Seq[Dependency[_]]) {

  def map[U: ClassTag](f: T => U): RDD[U] = ???
  def flatMap[U: ClassTag](f: T => TraversableOnce[U]): RDD[U] = ???

  def filter(f: T => Boolean): RDD[T] = ???

  def union(other: RDD[T]): RDD[T] = ???
  def intersection(other: RDD[T]): RDD[T] = ???
}
```

```
class PairRDDFunctions[K, V](self: RDD[(K, V)])
    (implicit kt: ClassTag[K], vt: ClassTag[V], ord: Ordering[K] = null
) {
  def join[W](other: RDD[(K, W)]): RDD[(K, (V, W))] = ???
  def groupByKey(): RDD[(K, Iterable[V])] = ???
  def reduceByKey(func: (V, V) => V, numPartitions: Int): RDD[(K, V)] =
 ???
}
```

- `Action`: 返回结果给驱动程序或持久化到外部存储，触发`Job`的提交

```
def count(): Long
def collect(): Array[T]
def reduce(f: (T, T) => T): T
```

```
def saveAsTextFile(path: String): Unit
```

分析WordCount

```
sc.textFile("hdfs://...")
  .flatMap(_.split(" "))
  .map((_, 1))
  .reduceByKey(_+_)
  .foreach(println)
```

- `val lines = sc.textFile("hdfs://...")`

先生成一个`HadoopRDD`，然后`map`为一个`MapPartitionsRDD`；

- `val words = lines.flatMap(_.split(" "))`

`MapPartitionsRDD`转变为`MapPartitionsRDD`

- `val pairs = words.map((_, 1))`

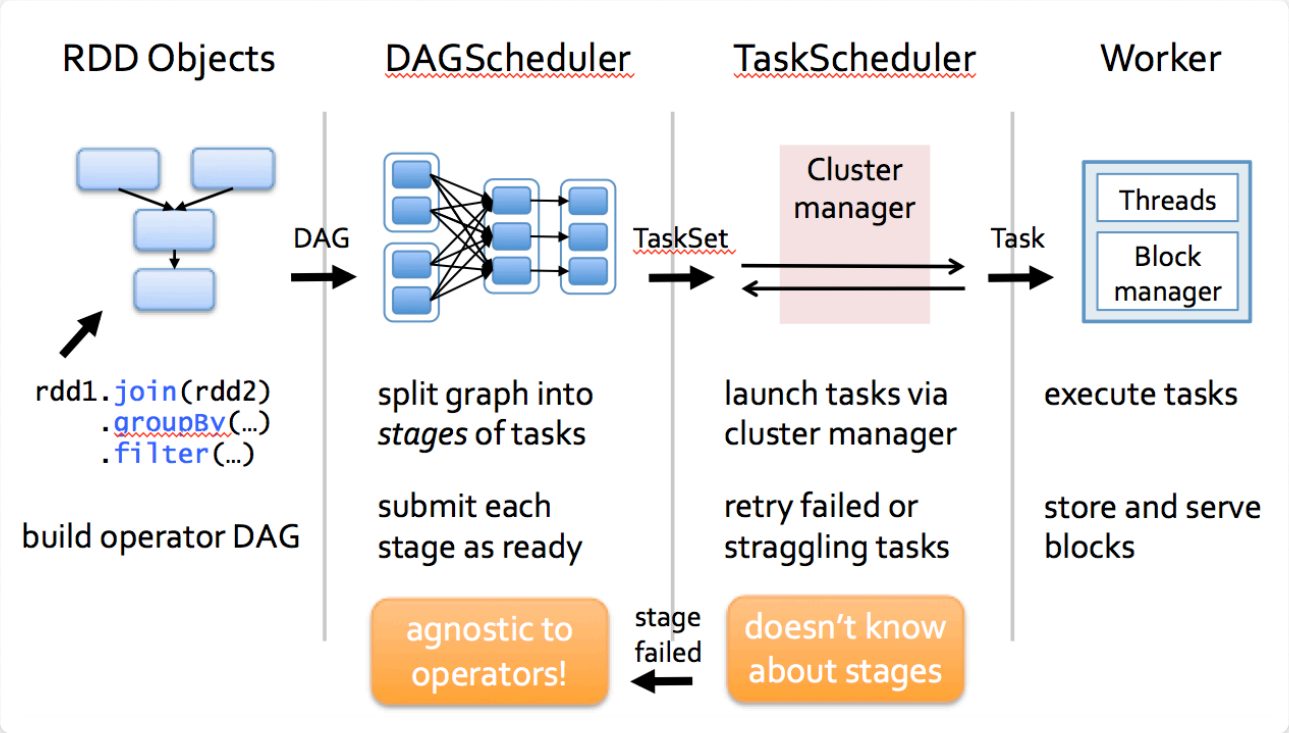
`MapPartitionsRDD`转变为`MapPartitionsRDD`

- `val result = pairs.reduceByKey(_+_)`

```
object RDD {
  implicit def rddToPairRDDFunctions[K, V](rdd: RDD[(K, V)])
    (implicit kt: ClassTag[K], vt: ClassTag[V], ord: Ordering[K] = null
): PairRDDFunctions[K, V] = {
    new PairRDDFunctions(rdd)
  }
}
```

Spark内核

生命周期



剖析RDD

特征

`RDD`，弹性分布式数据集，本质是一个只读的分区记录集合，具有如下几方面的特点：

- 弹性：基于内存计算
- 分区：正交的数据分区(`partitions`)；
- 函数式：只读的，不可变的，并行处理
- 容错：其中分区数据的自动恢复；
- 可序列化

`RDD`具有`5`个最基本的特征：

- Set of partitions(splits)
- List of dependencies on parent RDDs
- Function to compute a partition given parents
- Optional preferred locations
- Optional partitioning info (Partitioner)

参阅`RDD`源代码：

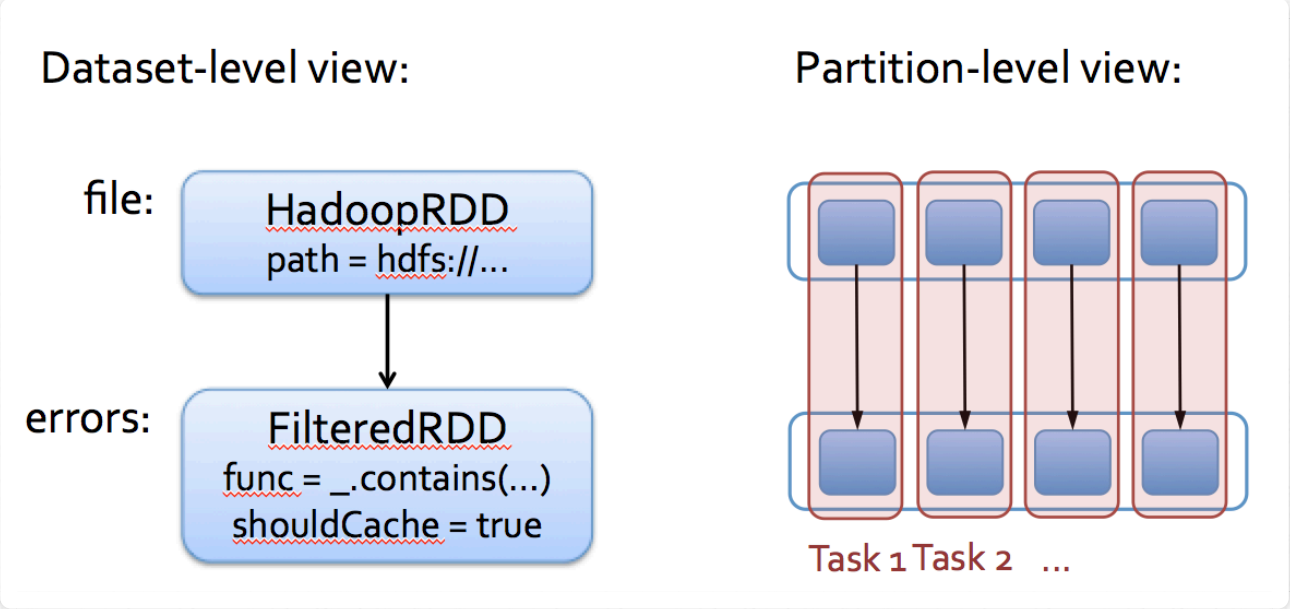
```
abstract class RDD[T: ClassTag](
    private var sc: SparkContext,
    private var deps: Seq[Dependency[_]]
  ) {
```

```
  def compute(split: Partition, context: TaskContext): Iterator[T]

  protected def getPartitions: Array[Partition]
  protected def getDependencies: Seq[Dependency[_]] = deps
  protected def getPreferredLocations(split: Partition): Seq[String] =
Nil
  val partitioner: Option[Partitioner] = None
}
```

RDD例子

```
val file = sc.textFile("hdfs://...")
val errors = file.filter(_.contains("ERROR"))

errors.cache()
errors.count()
```

RDD运行时



HadoopRDD

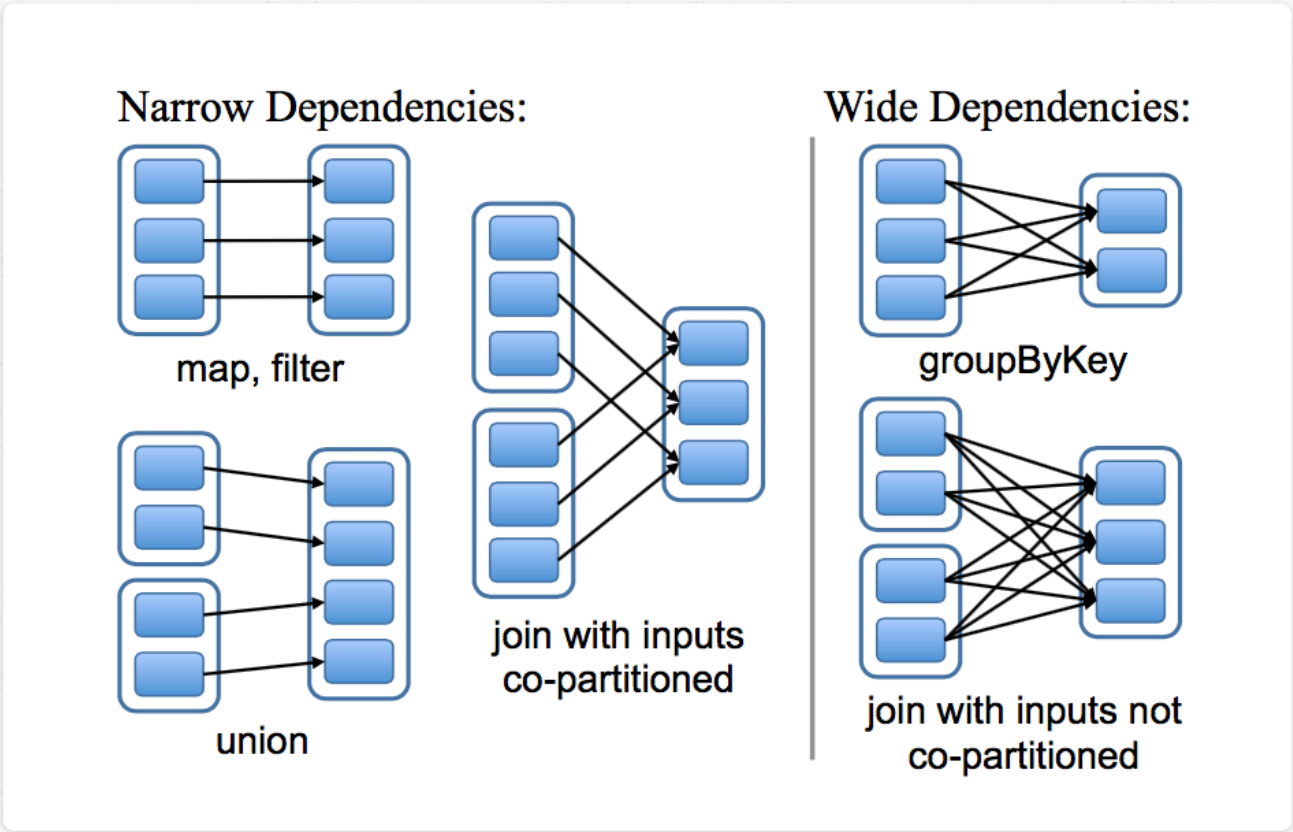
- partitions = one per HDFS block
- dependencies = none
- compute(partition) = read corresponding block
- preferredLocations(part) = HDFS block location
- partitioner = none

MapPartitionsRDD

- partitions = same as parent RDD
- dependencies = *“one-to-one”* on parent
- compute(partition) = compute parent and filter it
- preferredLocations(part) = none(ask parent)
- partitioner = none

依赖关系

- `Narrow Dependency`：一个父`RDD`之多被一个子`RDD`引用；
- `Shuffle/Wide Dependency`：一个父`RDD`被多个子`RDD`引用；



区分的意义

- `Narrow Dependency`可以支持在同一个`Cluster Node`上以`Pipeline`的形式执行多条命令；
- `Narrow Dependency`的数据容错性会更有效，它只需重新计算丢失了的父分区即可，并且可以并行地在不同节点上重计算。

继承关系

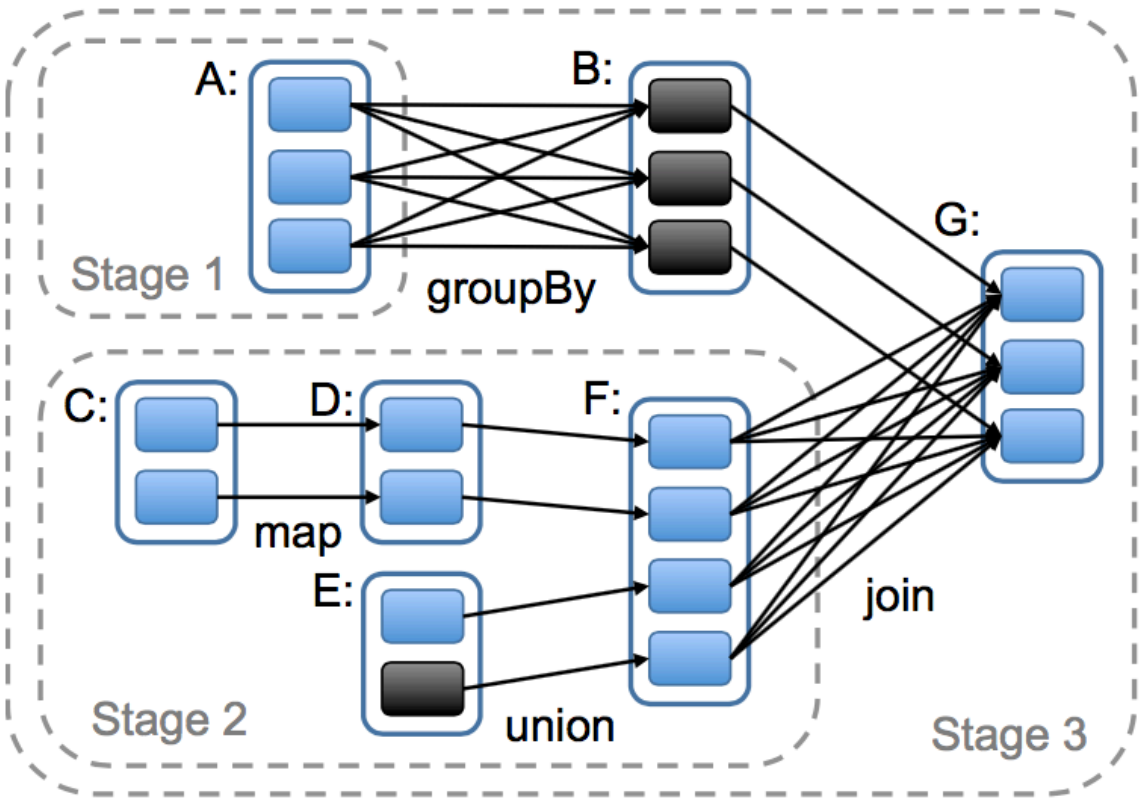
```
abstract class Dependency[T] extends Serializable {
  def rdd: RDD[T]
}
```

存在两个子类:

- `NarrowDependency`
- `ShuffleDependency`

Stage DAG

- `Wide Dependency`是划分`Stage`的边界;
- `Narrow Dependency`的`RDD`被放在同一个`Stage`之中;



思考`2`个问题:

1. 如何确定`Stage`的起始边界?
   - 开始: 读取外部数据, 或读取`Shuffle`数据;
   - 结束: 发生`Shuffle`, 或者`Job`结束;
2. 如何确定`Final Stage`?
   - 触发`Action`的`RDD`所在的`Stage`
3. 如何表示一个`Stage`?

```
private[spark] abstract class Stage(
    val id: Int,
```

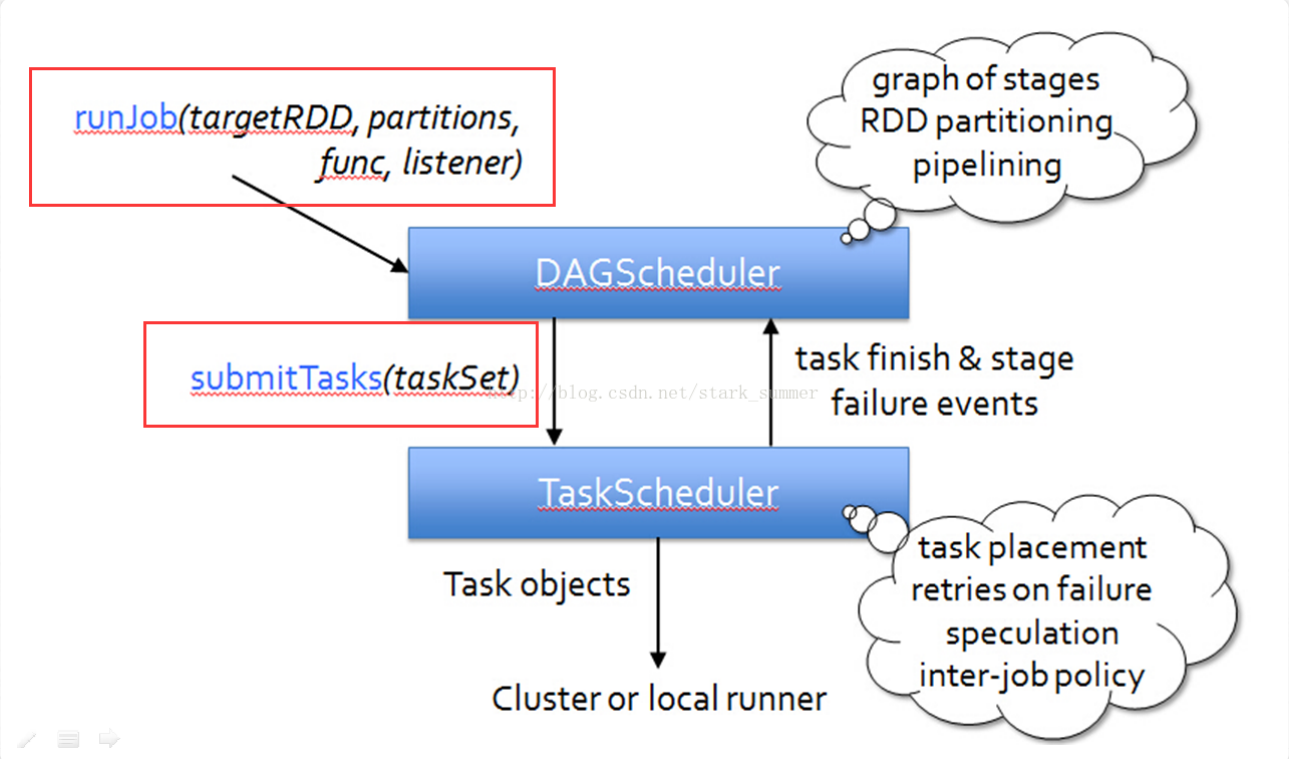
```
    val rdd: RDD[_],
    val numTasks: Int)
```

其中`rdd`为`Stage`中最后一个`RDD`，可反向推演出完整的`Stage`。

作业提交

作业提交由`RDD`的`Action`触发，交由`DAGScheduler`根据`RDD`的依赖关系绘制`Stage DAG`，最后按照`DAG`的广度优先遍历算法依次将`Stage(TaskSet)`提交给`TaskScheduler`进行调度和处理；`TaskScheduler`将`TaskSet`中的任务提交至集群`Worker Node`上的`Executor`执行；`Executor`维护了线程池，由单独的线程处理`Task`的运行。

事件流



任务提交源码剖析

1. sc.runJob->dagScheduler.runJob->submitJob
2. DAGScheduler::submitJob会创建JobSummitted的event发送给内嵌类**eventProcessActor**
3. eventProcessActor在接收到JobSubmmitted之后调用**processEvent**处理函数
4. job到stage的转换，生成finalStage并提交运行，关键是调用**submitStage**
5. 在submitStage中会计算stage之间的依赖关系，依赖关系分为**宽依赖**和**窄依赖**两种
6. 如果计算中发现当前的stage没有任何依赖或者所有的依赖都已经准备完毕，则提交task
7. 提交task是调用函数**submitMissingTasks**来完成
8. task真正运行在哪个worker上面是由TaskScheduler来管理，也就是上面的submitMissingTasks会调用TaskScheduler::submitTasks
9. TaskSchedulerImpl中会根据Spark的当前运行模式来创建相应的backend,如果是在单机运行则创建LocalBackend
10. LocalBackend收到TaskSchedulerImpl传递进来的**ReceiveOffers**事件
11. receiveOffers->executor.launchTask->TaskRunner.run

Spark部署模式

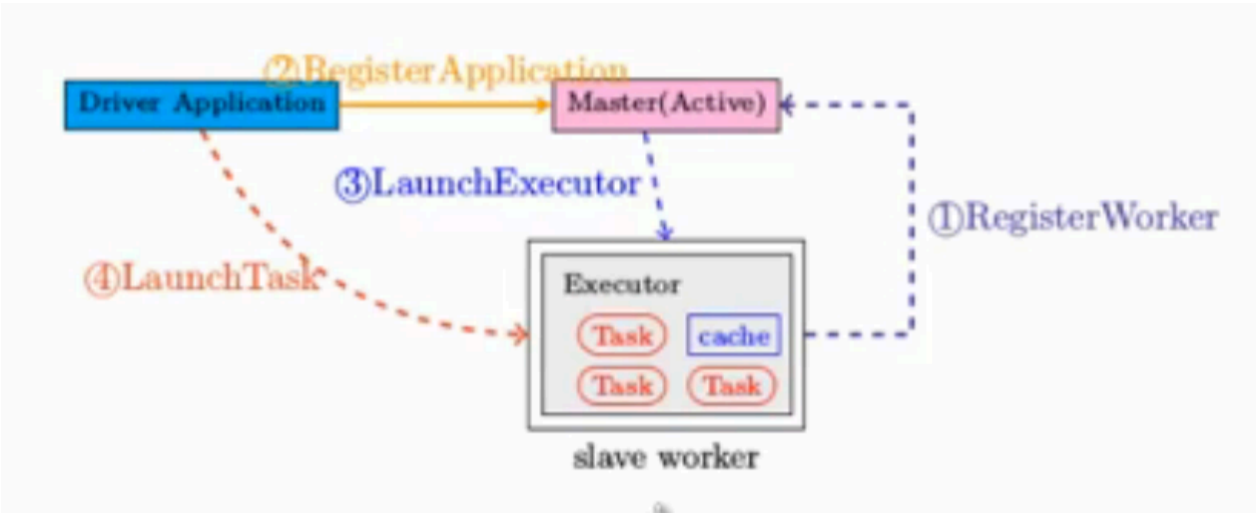
运行模式

- `--master: spark://host:port, mesos://host:port, yarn, yarn-cluster, yarn-client, local`
- `--deploy-mode: client/cluster`

Spark Standalne

`Driver`运行在哪里？

- `spark-shell`，运行在`Master`上；
- `spark-submit`，运行在客户端上；



Spark on Yarn

Spark on Mesos

Spark SQL

Spark Streaming

About Me

刘光聪，程序员，敏捷教练，开源软件爱好者，具有多年大型遗留系统的重构经验，对`OO`，`FP`，`DSL`等领域具有浓厚的兴趣。

- GitHub: https://github.com/horance-liu
- Email: horance@outlook.com