

Spark基础编程



摘要

- Spark安装运行
- Spark编程模型
- Spark编程示例



摘要

- Spark安装运行
- Spark编程模型
- Spark编程示例



Spark 安装运行

4

- Spark 系统运行所需的软件环境
- Standalone 模式安装 Spark
- Spark 和集群管理工具的结合

Spark runs on Java 8/11, Scala 2.12, Python 2.7+/3.4+ and R 3.5+. Java 8 prior to version 8u92 support is deprecated as of Spark 3.0.0. Python 2 and Python 3 prior to version 3.6 support is deprecated as of Spark 3.0.0. For the Scala API, Spark 3.0.1 uses Scala 2.12. You will need to use a compatible Scala version (2.12.x).



Spark系统运行的软件环境

□ 操作系统

- ▣ Spark是运行在Java虚拟机上的，因此在Windows、Linux和MacOS上都能够安装Spark。但由于Spark中的一些工具和脚本（如启动脚本）是针对Linux环境编写的，因此建议在Linux操作系统上安装和运行Spark。

□ SSH

- ▣ 主要用于在集群环境下远程管理Spark节点以及Spark节点间的安全共享访问。

□ Java

- ▣ 主要用于运行Spark以及使用Spark提供的Java API进行开发，如Java1.8



Spark系统运行的软件环境

6

□ Scala

- ▣ 除了Java API以外，Spark还向程序员提供了Scala API，如要用Scala开发Spark应用，则需要安装Scala

□ Python

- ▣ 类似的，Spark也提供了Python API，如要用Python开发Spark应用，则需要安装Python

□ HDFS

- ▣ Spark是一个分布式计算引擎，其输入输出数据可以保存在分布式文件系统中。这里推荐使用Hadoop中的HDFS



Standalone模式

7

- **Spark**框架本身自带了完整的资源调度管理服务，可以独立部署到一个集群中，而不需要依赖其他系统来为其提供资源管理调度服务。



Standalone模式的安装

8

- ❑ 软件环境准备
- ❑ 下载编译好的**Spark**包
- ❑ 修改**Spark**配置文件
- ❑ 启动**Spark**
- ❑ 运行测试程序
- ❑ 查看集群状态



软件环境的准备

9

- 安装SSH, Java, HDFS (必须)
- 安装Scala, 建议Scala 2.12 (可选)
- 安装Python, 建议PySpark 3.0.1 (可选)
 - ▣ `pip install pyspark`



下载编译好的Spark包

10

□ 下载地址：

<http://spark.apache.org/downloads.html>

Download Apache Spark™

1. Choose a Spark release:
2. Choose a package type:
3. Download Spark: [spark-3.0.1-bin-hadoop2.7.tgz](#)
4. Verify this release using the 3.0.1 [signatures](#), [checksums](#) and [project release KEYS](#).

Note that, Spark 2.x is pre-built with Scala 2.11 except version 2.4.2, which is pre-built with Scala 2.12. Spark 3.0+ is pre-built with Scala 2.12.

Link with Spark

Spark artifacts are [hosted in Maven Central](#). You can add a Maven dependency with the following coordinates:

```
groupId: org.apache.spark  
artifactId: spark-core_2.12  
version: 3.0.1
```



修改Spark配置文件

11

- Spark的配置文件存放在Spark安装目录下的conf目录中：
 - ▣ spark-env.sh： 主要完成Spark环境变量设置
 - ▣ spark-defaults.conf： Spark默认配置
 - ▣ slaves： 主要完成Worker节点的IP设置



启动Spark

12

- Spark提供了一系列用于启动/停止的脚本
 - ▣ `sbin/start-master.sh`: 启动Master
 - ▣ `sbin/start-slaves.sh`: 启动所有的Worker
 - ▣ `sbin/start-all.sh`: 启动Master和所有的Worker
 - ▣ `sbin/stop-master.sh`: 停止Master
 - ▣ `sbin/stop-slaves.sh`: 停止所有的Worker
 - ▣ `sbin/stop-all.sh`: 停止Master和所有的Worker
- 执行启动脚本后, 可以使用JPS命令查看进程信息。
若Spark正常启动, 那么在Master节点会有一个Master进程, 在每个Worker节点会有Worker进程。



运行测试程序

13

- 启动Spark，可以向Spark集群提交一个测试程序

- ▣ `./bin/run-example SparkPi 10`

- 运行Shell

- ▣ `./bin/spark-shell`

```
1. bin/spark-shell (java)
yuping@mbp ~/Documents/dev/spark-2.2.1-bin-hadoop2.7$ bin/spark-shell
Using Spark's default log4j profile: org/apache/spark/log4j-defaults.properties
Setting default log level to "WARN".
To adjust logging level use sc.setLogLevel(newLevel). For SparkR, use setLogLevel(newLevel).
17/12/10 16:02:31 WARN NativeCodeLoader: Unable to load native-hadoop library for your platform... using builtin-java classes where applicable
Spark context Web UI available at http://192.168.0.4:4040
Spark context available as 'sc' (master = local[*], app id = local-1512892952618).
Spark session available as 'spark'.
Welcome to

  ____ _
 / ___ \
/ /   \ \
/_/     \_\

 version 2.2.1

Using Scala version 2.11.8 (Java HotSpot(TM) 64-Bit Server VM, Java 1.8.0_144)
Type in expressions to have them evaluated.
Type :help for more information.

scala> val textFile = sc.textFile("README.md")
textFile: org.apache.spark.rdd.RDD[String] = README.md MapPartitionsRDD[1] at textFile at <console>:24

scala> 
```




查看Spark状态

14

- 在Spark集群运行期间，可以用浏览器查看Spark状态

□ <http://localhost:4040>

 2.2.1

Jobs | Stages | Storage | Environment | Executors | SQL

Spark shell application UI

Spark Jobs (?)

User: yuping
Total Uptime: 5.7 min
Scheduling Mode: FIFO
Completed Jobs: 3
[Event Timeline](#)

Completed Jobs (3)


Job Id ▾	Description	Submitted	Duration	Stages: Succeeded/Total	Tasks (for all stages): Succeeded/Total
2	count at <console>:26	2017/12/10 16:11:04	23 ms	2/2	<div>2/2</div>
1	first at <console>:26	2017/12/10 16:10:35	24 ms	1/1	<div>1/1</div>
0	count at <console>:26	2017/12/10 16:10:27	0.4 s	2/2	<div>2/2</div>





查看Spark状态

15

 2.2.1

Jobs

Stages

Storage

Environment

Executors

SQL

Spark shell application UI

Stages for All Jobs

Completed Stages: 5

Completed Stages (5)

Stage Id ▾	Description		Submitted	Duration	Tasks: Succeeded/Total	Input	Output	Shuffle Read	Shuffle Write
4	count at <console>:26	+details	2017/12/10 16:11:04	4 ms	<div>1/1</div>			59.0 B	
3	count at <console>:26	+details	2017/12/10 16:11:04	12 ms	<div>1/1</div>	3.7 KB			59.0 B
2	first at <console>:26	+details	2017/12/10 16:10:35	20 ms	<div>1/1</div>	7.4 KB			
1	count at <console>:26	+details	2017/12/10 16:10:27	44 ms	<div>1/1</div>			59.0 B	
0	count at <console>:26	+details	2017/12/10 16:10:27	0.2 s	<div>1/1</div>	3.7 KB			59.0 B



查看Spark状态

16

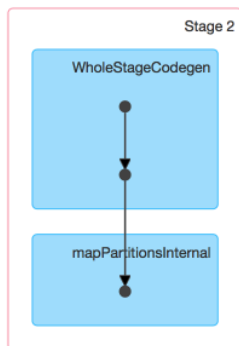
Details for Job 1

Status: SUCCEEDED

Completed Stages: 1

▸ Event Timeline

▾ DAG Visualization



Completed Stages (1)

Stage Id ▾	Description	Submitted	Duration	Tasks: Succeeded/Total	Input	Output	Shuffle Read	Shuffle Write
2	first at <console>:26 +details	2017/12/10 16:10:35	20 ms	1/1	7.4 KB			



Spark和集群管理工具的结合

17

- 管理的难题
- 统一资源管理平台和集装箱思想
- 使用YARN、Mesos或Kubernetes运行Spark
- 使用Docker部署Spark



Spark和集群管理工具的结合

18

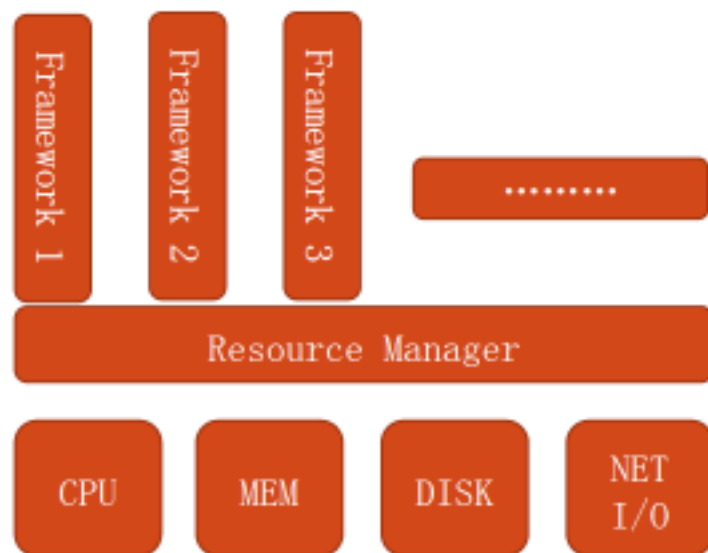
- 不同计算引擎各有所长，真实应用中往往需要同时使用不同的计算框架；
- 不同框架和应用会争抢资源，互相影响，使得管理难度和成本增加；
- 应用往往在单机上开发，在小规模集群上测试，在云上运行。在不同环境下部署时总要经历复杂而且痛苦的配置过程。



Spark和集群管理工具的結合

19

- 统一的资源管理平台将资源独立管理
 - ▣ YARN, Mesos, Kubernetes
- 集装箱思想
 - ▣ 将应用和依赖“装箱”，一次配置，随处部署
 - ▣ Docker
- 通过资源管理可在同一个集群平台上部署不同的计算框架和应用，从而实现多租户资源共享





Spark和集群管理工具的结合

20

- **资源管理**：所有接入的框架要先向它申请资源，申请成功之后，再由平台自身的调度器决定资源交由哪个任务使用
- **资源共享**：通过资源管理可在同一集群平台上部署不同的计算框架和应用，实现多租户资源共享
- **资源隔离**：不同的框架中的不同任务往往需要的资源（内存，CPU，网络IO等）不同，它们运行在同一个集群中，会相互干扰。所以需要实现资源隔离以免任务之间由资源争用导致效率下降
- **提高资源利用效率**：当将各种框架部署到同一个大的集群中，进行统一管理和调度后，由于各种作业交错且作业提交频率大幅度升高，则为资源利用率的提升增加了机会
- **扩展和容错**：统一资源管理平台不能影响到上层框架的可扩展性和容错，同时自身也应具备良好的可扩展性和容错性



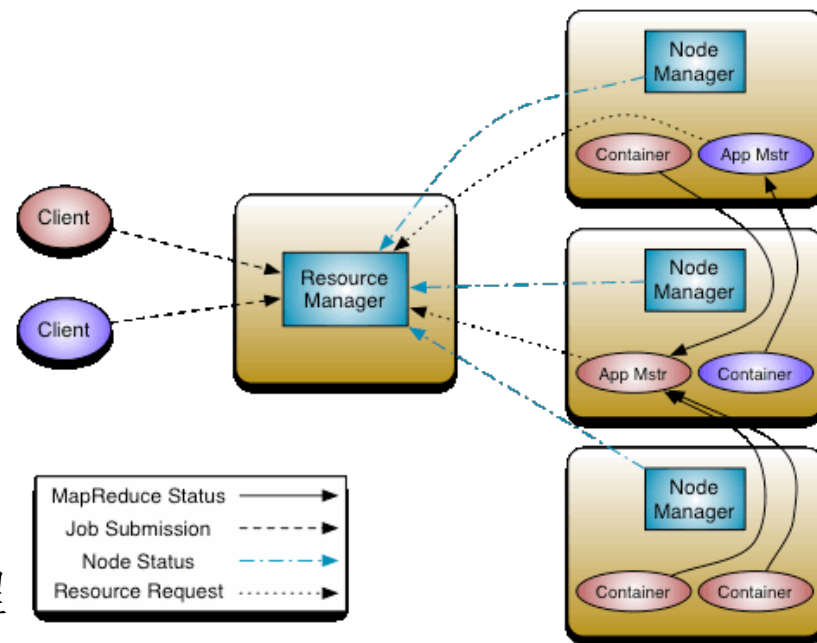
YARN

21

- YARN是Hadoop2.0时代的编程架构，被称为新一代MapReduce。其核心思想是将原MapReduce框架中的JobTracker和TaskTracker重新设计，变成了：

- Resource Manager
- Application Master
- Node Manager

- 除了支持Hadoop2.0以外，实际上，YARN还是一个独立的底层资源管理框架，可用于支持和运行其他各种计算框架，例如Spark





- **ResourceManager** 是一个中心的服务，它负责作业与资源的调度，负责调度、启动每一个 **Job** 所属的 **ApplicationMaster**，监控 **ApplicationMaster** 的存在情况。接收 **JobSubmitter** 提交的作业，按照作业的上下文环境 (**Context**) 信息，以及从 **NodeManager** 收集来的状态信息，启动调度过程，分配一个 **Container** 作为 **ApplicationMaster**
- **ApplicationMaster** 负责一个 **Job** 生命周期内的所有工作，类似老的框架中的 **JobTracker**。但注意每一个 **Job**（不是每一种）都有一个 **ApplicationMaster**，它可以运行在 **ResourceManager** 以外的机器上
- **NodeManager** 功能比较专一，就是负责 **Container** 状态的维护，并向 **ResourceManager** 保持心跳



Spark on YARN

23

Spark
SQL

Spark
Streaming

MLlib
(machine learning)

GraphX

Spark

YARN

HDFS



Spark on YARN

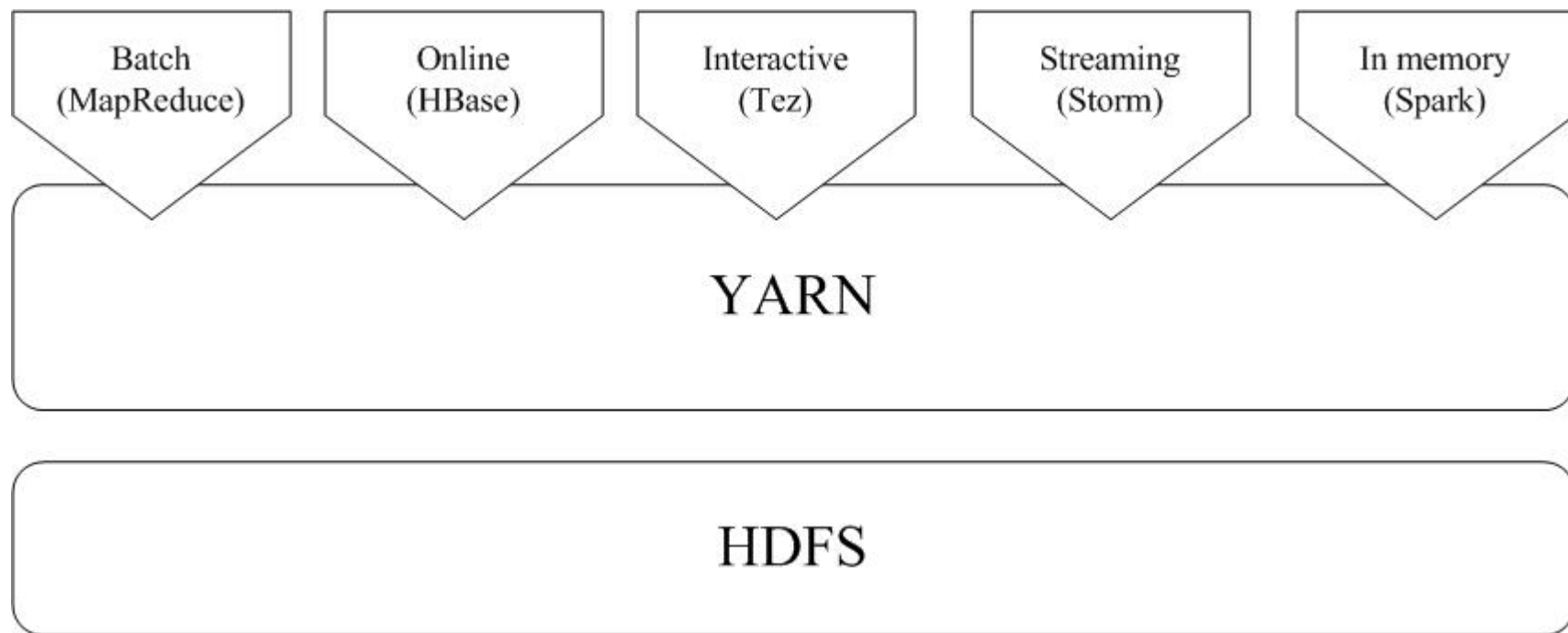
24

- 1.在`spark-env.sh`里面，设置`HADOOP_CONF_DIR`为hadoop配置文件的目录
- 2.在`spark-default.conf`里设置其它可配置选项。具体请参考spark.apache.org/docs/latest/running-on-yarn.html
- 3.试运行
 - 使用`--master`选项选择Spark on YARN的运行模式，共有`yarn-client`和`yarn-cluster`两种选项。
 - 例如：`spark-submit --class path.to.your.Class --master yarn-cluster [options] <app jar>`
 - 或`spark-shell --master yarn-client`
 - 其中，`yarn-cluster`是指yarn的`application master`进程中包含`spark driver`，发起任务的客户端可以在任务初始化完成之后离开，不和yarn维持通信。而`yarn-client`模式中，`spark driver`会运行在客户端，故而客户端不能离开。



Hadoop和Spark的统一部署

25



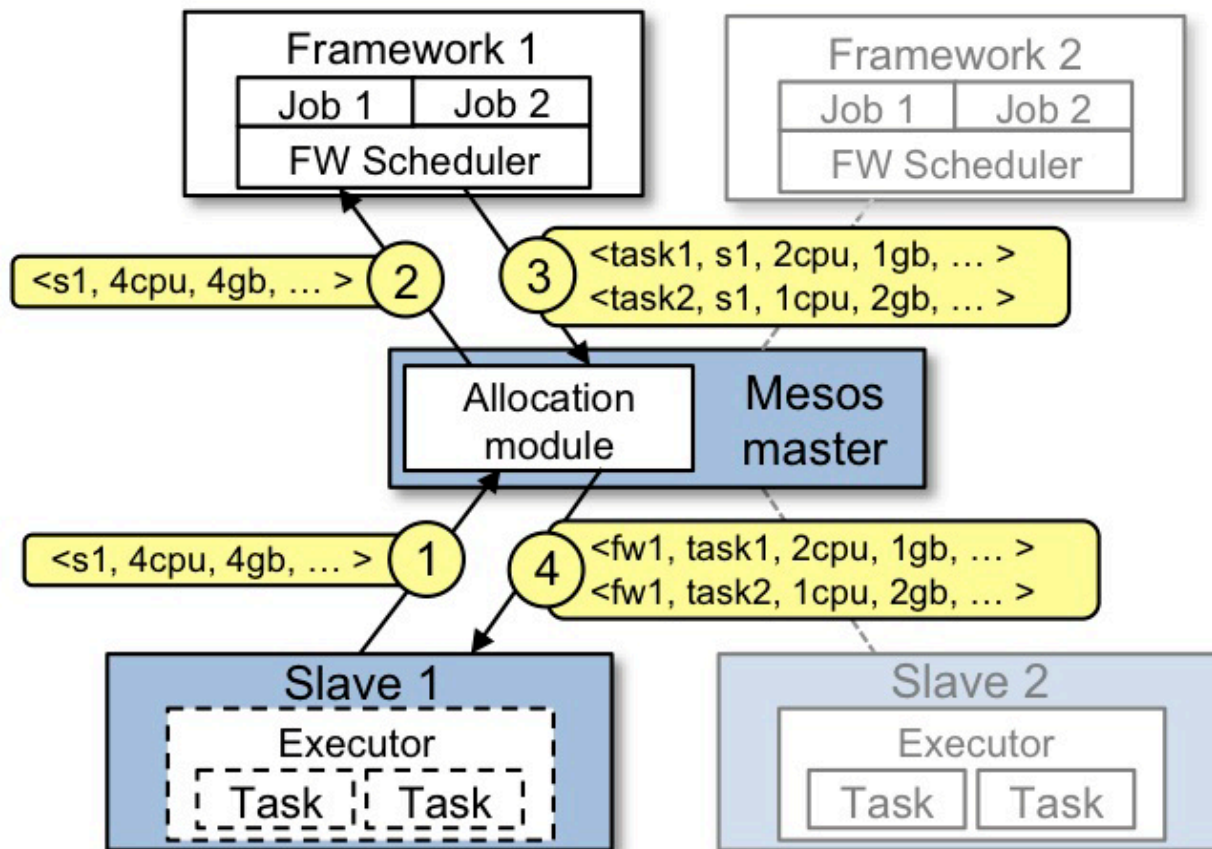


- **Mesos**是**Apache**旗下著名的分布式资源管理框架，被称为分布式系统的内核
- **Mesos**包含两个组件，**Master**和**Slave**
 - ▣ **Master**：作为上层应用和资源间的桥梁，负责向上层应用分配**Slave**上的资源
 - ▣ **Slave**：一方面负责接收**Master**的指令，启动**Executor**运行上层应用的任务，另一方面向**Master**汇报自己的资源存量，以便**Master**调度
- 上层计算框架接入**Mesos**需要自行实现两个组件：**Scheduler**和**Executor**
 - ▣ **Scheduler**：负责接收**Mesos**提供给计算框架的资源，再通过自己的调度器分配给每个任务
 - ▣ **Executor**：而**Mesos**通过调用不同计算框架的**executor**启动该计算框架的任务



Mesos

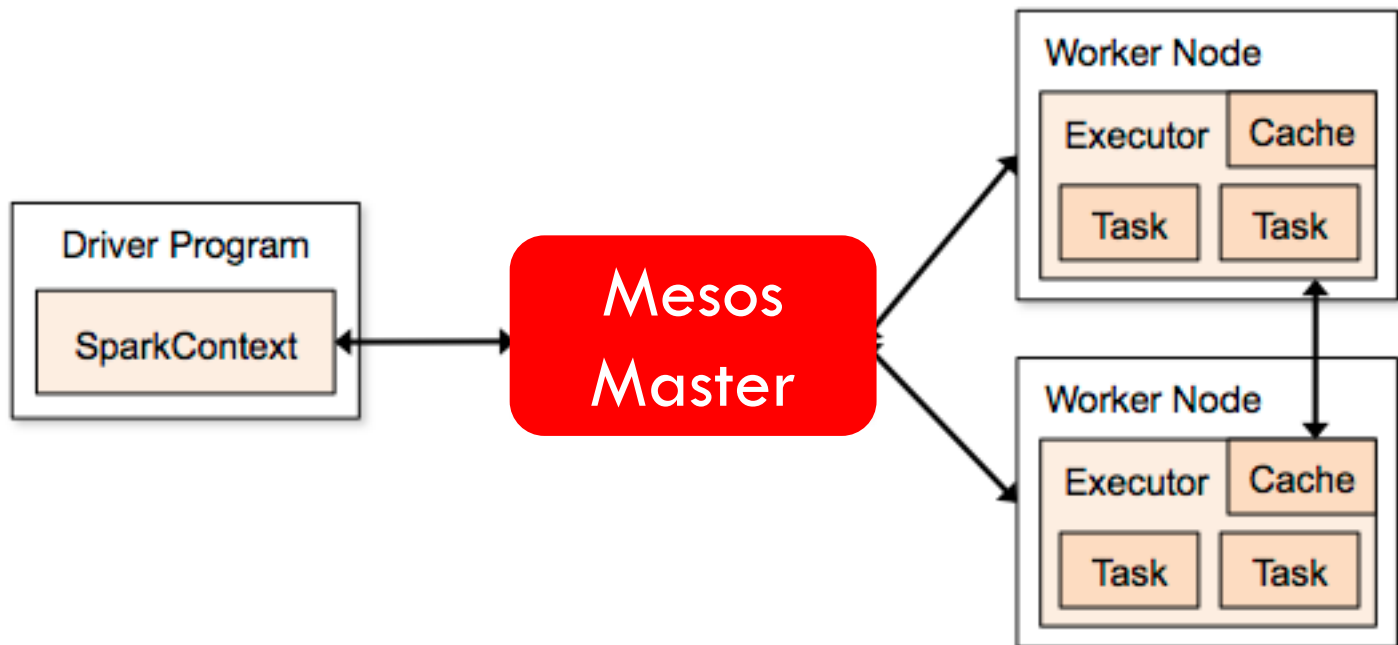
27





Spark on Mesos

28





Spark on Mesos

29

- 1.安装好Mesos之后，在Mesos管理的文件系统（例如HDFS，S3.....）中上传Spark程序包。
 - ▣ `hadoop fs -put spark-2.3.0.tar.gz /path/to/spark-2.3.0.tar.gz`
- 2.在spark-env.sh里设置：
 - ▣ `export MESOS_NATIVE_LIBRARY=<文件libmesos.so所在的目录>`
 - ▣ `export SPARK_EXECUTOR_URI=<第一步上传spark程序包的URL>`
 - ▣ 在spark-default.conf中设置`spark.executor.uri=<第一步上传spark程序包的URL>`.
- 3.调整模式
 - ▣ 在spark-default.conf中设置`spark.mesos.coarse`的ture或false设置Spark on Mesos的运行模式。默认为false，代表fine-grained模式，每个Spark任务开启一个Mesos任务。True则为coarse-grained模式，所有Spark任务都是一个Mesos任务下的小任务
- 4. 试运行/`spark-shell --master mesos://host:5050`



Kubernetes

30

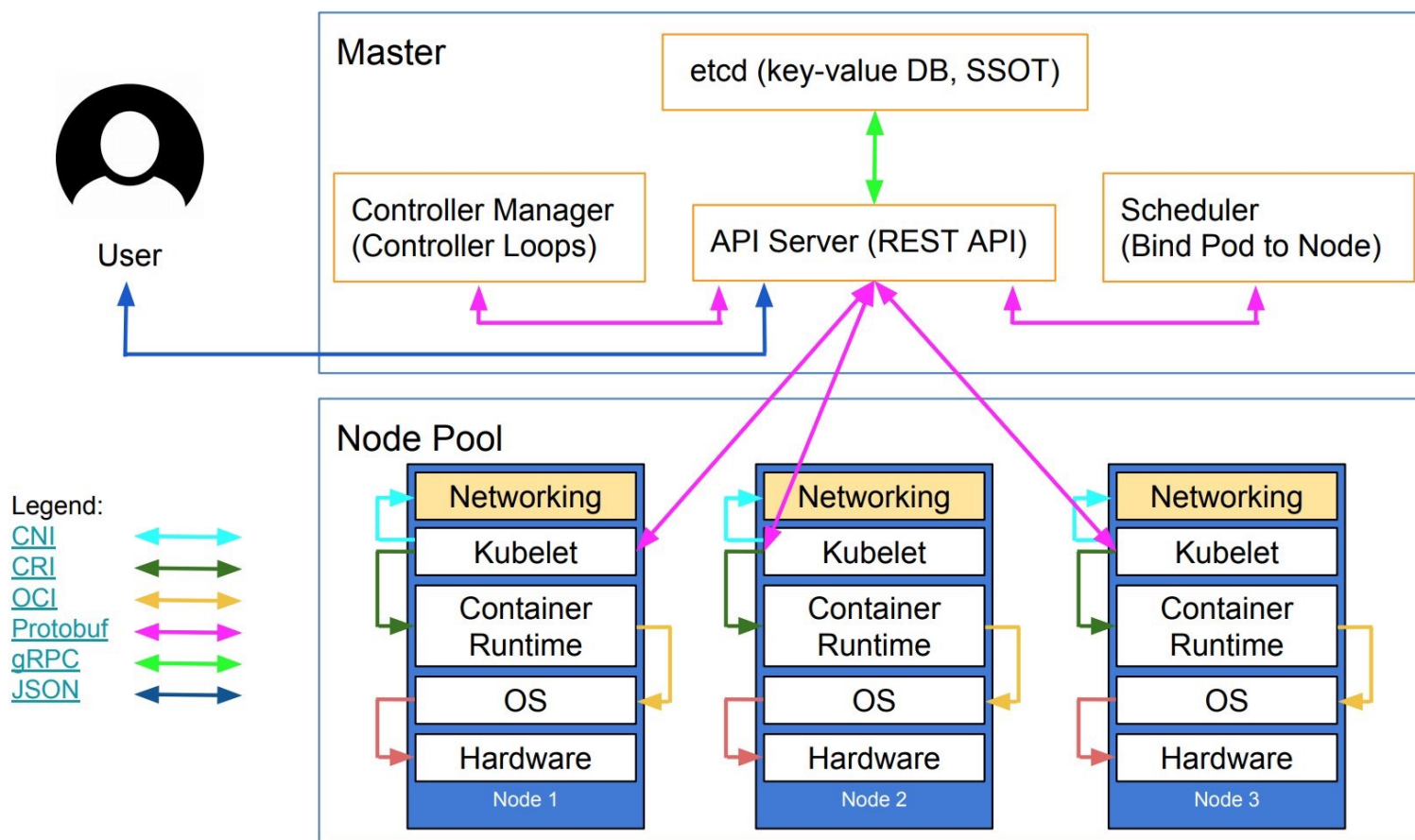
- **Kubernetes**是**Google**基于**Borg**开源的容器编排调度引擎，支持自动化部署、大规模可伸缩、应用容器化管理。在生产环境中部署一个应用程序时，通常要部署该应用的多个实例以便对应用请求进行负载均衡。
- 在**Kubernetes**中，我们可以创建多个容器，每个容器里面运行一个应用实例，然后通过内置的负载均衡策略，实现对这一组应用实例的管理、发现、访问，而这些细节都不需要运维人员去进行复杂的手工配置和处理。



Kubernetes

31

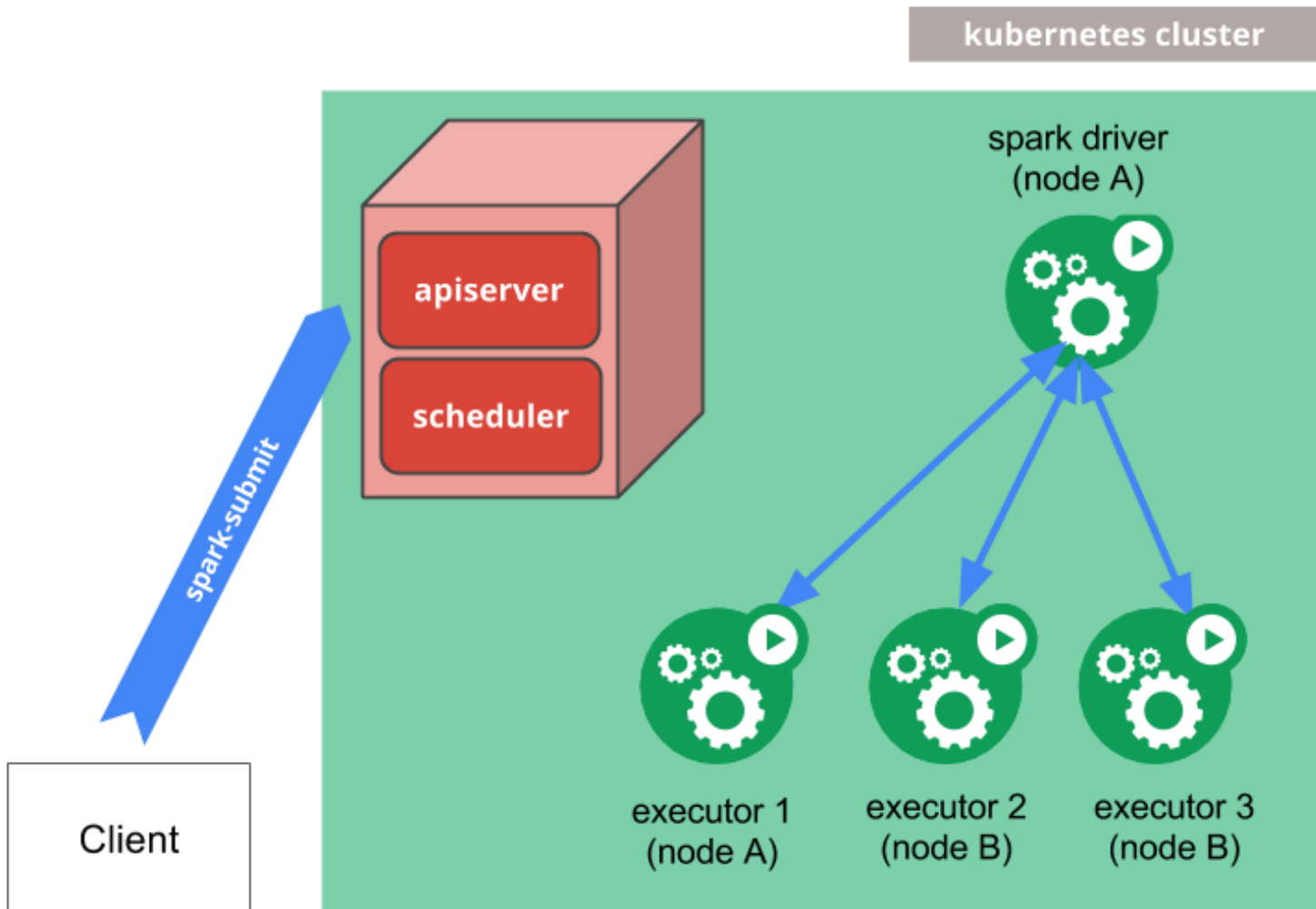
Kubernetes' high-level component architecture





Spark on Kubernetes

32



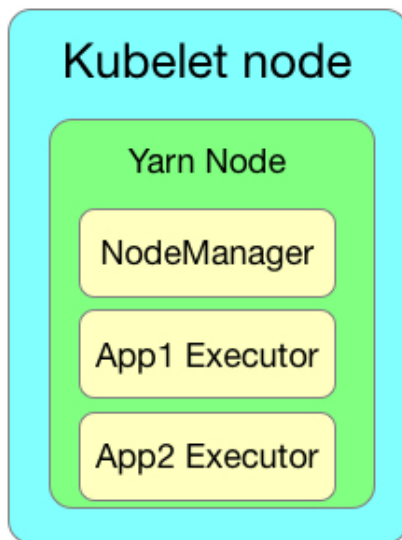


Spark on Kubernetes

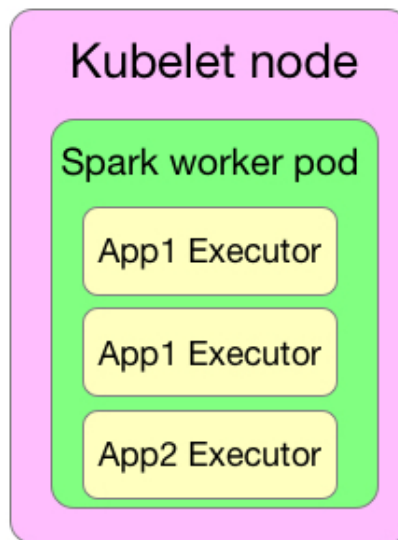
33

Spark on Kubernetes with different schedulers

Yarn



Standalone



Native





摘要

- Spark安装运行
- Spark编程模型
- Spark编程示例



Spark编程模型与编程接口

35

- **Spark**为了解决以往分布式计算框架存在的一些问题(重复计算、资源共享、系统组合),提出了一个分布式数据集的抽象数据模型:
 - ▣ **RDD(Resilient Distributed Datasets):**弹性分布式数据集



Spark编程模型与编程接口

36

- **RDD**是一种分布式的内存抽象，允许在大型集群上执行基于内存的计算（**In-Memory Computing**），同时还保持了**MapReduce**等数据流模型的容错特性。
- **RDD**只读、可分区，这个数据集的全部或部分可以缓存在内存中，在多次计算间重用。
- 简单来说，**RDD**是**MapReduce**模型的一种简单的扩展和延伸。



Spark的基本编程方法与示例

37

- 在一个存储于HDFS的Log文件中，计算出现ERROR的行数，本程序使用scala语言编写，这个语言也是Spark开发和编程的推荐语言。

```
def main(args: Array[String]) { // 定义一个main函数
    // 定义一个sparkConf，提供Spark运行的各种参数，如程序名称、用户名称等
    val conf = new SparkConf().setAppName("Spark Pi")
    // 创建Spark的运行环境，并将Spark运行的参数传入Spark的运行环境中
    val sc = new SparkContext(conf)
    // 调用Spark的读文件函数，从HDFS中读取Log文件，输出一个RDD类型的实例：fileRDD。具体类型：
    RDD[String]
    val fileRDD = sc.textFile("hdfs:///root/Log")
    // 调用RDD的filter函数，过滤fileRDD中的每一行，如果该行中含有ERROR，保留；否则，删除。生成另
    一个RDD类型的实例：filterRDD。具体类型：RDD[String]
    // 注：line=>line.contains("ERROR")表示对每一个line应用contains()函数
    val filterRDD = fileRDD.filter(line=>line.contains("ERROR"))
    // 统计filterRDD中总共有多少行，result为Int类型
    val result = filterRDD.count()
    sc.stop() // 关闭Spark
}
```



RDD的创建

38

□ `val file=sc.textFile("hdfs:///root/Log")`

这句代码创建了一个RDD，那么RDD是怎么创建的？又有那些注意事项？

□ 从形式上看，RDD是一个分区的只读记录的集合。因此，RDD只能通过两种方式创建：

▣ 1、通过从存储器中读取，例如上述代码，从HDFS中读取。例如：

`val rdd= sc.parallelize(1 to 100, 2)`

//生成一个1到100的数组，并行化成RDD

▣ 2、其他RDD的数据上的确定性操作来创建(即Transformation)。例如：

`val filterRDD=file.filter(line=>line.contains("ERROR"))`

//通过file的filter操作生成一个新的filterRDD



RDD的操作

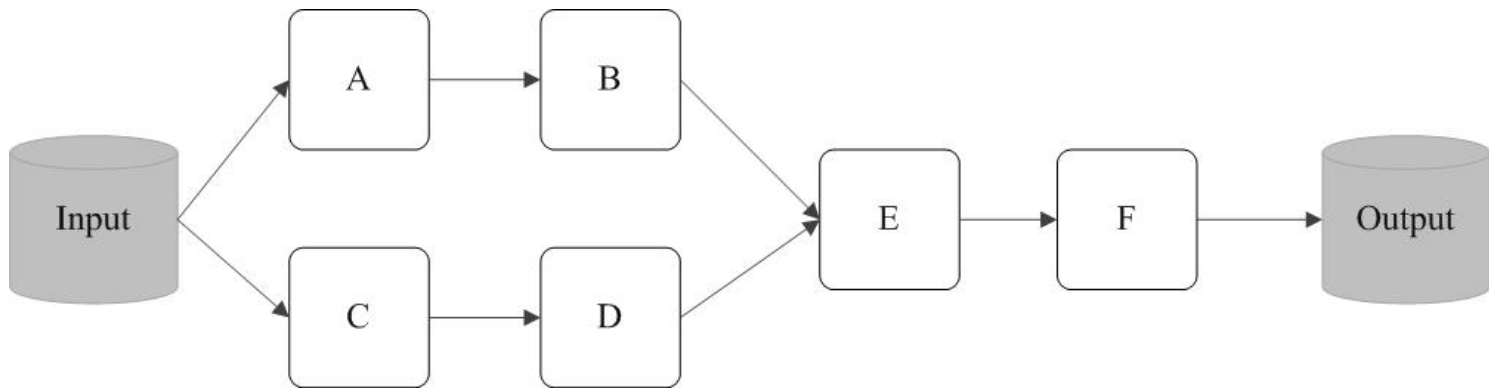
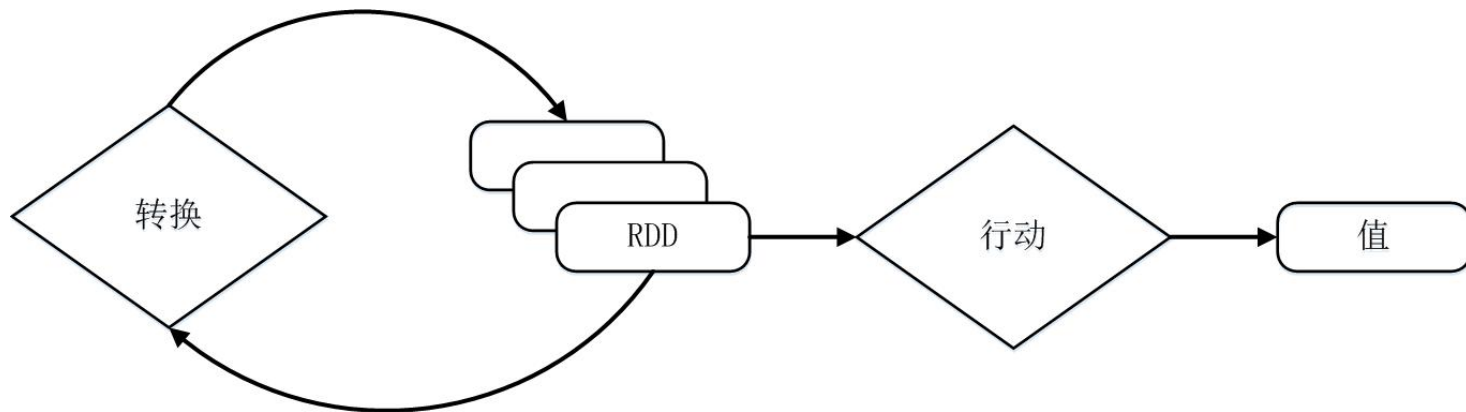
39

- **转换 (transformation)**：这是一种惰性操作，即使用这种方法时，只是定义了一个新的**RDD**，而并不马上计算新的**RDD**内部的值。
 - 例：`val filterRDD=fileRDD.filter(line=>line.contains("ERROR"))`
 - 上述这个操作对于**Spark**来说仅仅记录从**file**这个**RDD**通过**filter**操作变换到**filterRDD**这个**RDD**的变换，并不计算**filterRDD**的结果。
- **动作 (action)**：立即计算这个**RDD**的值，并返回结果给程序，或者将结果写入到外存储中。
 - 例：`val result = filterRDD.count()`
 - 上述操作计算最终的**result**结果是多少，包括前边**transformation**时的变换。



RDD的操作

40

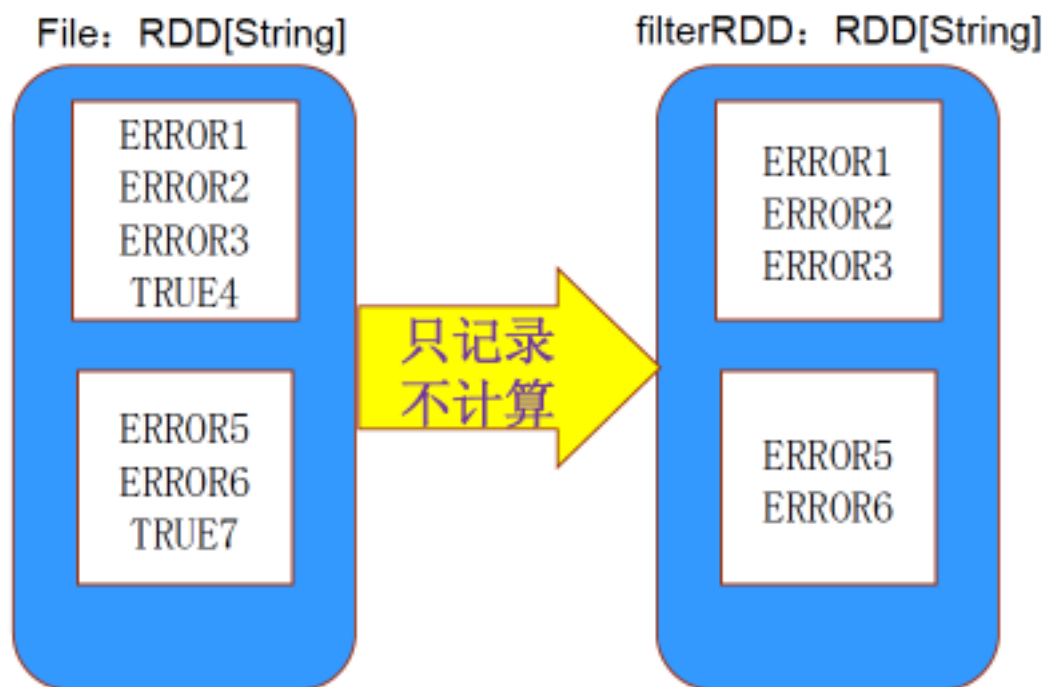




RDD的transformation图示

41

- `val filterRDD=fileRDD.filter(line=>line.contains("ERROR"))`
- 设fileRDD中包含以下7行数据：
- "ERROR1 ERROR2 ERROR3 TRUE4 ERROR5 ERROR6 TRUE7"

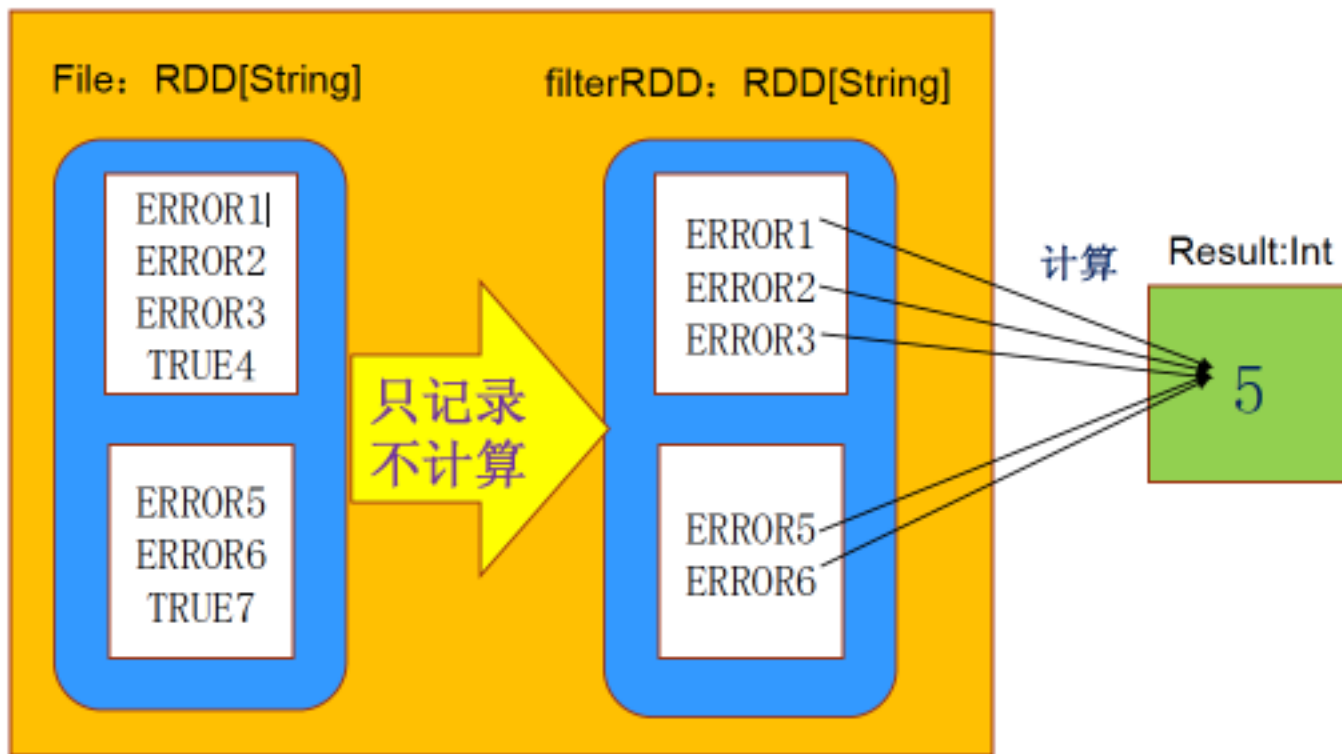




RDD的 action 图示

42

□ 例： `val result = filterRDD.count()`





Spark支持的一些常用transformation操作

43

Transformation	Meaning
<code>map(func)</code>	Return a new distributed dataset formed by passing each element of the source through a function <i>func</i> .
<code>filter(func)</code>	Return a new dataset formed by selecting those elements of the source on which <i>func</i> returns true.
<code>flatMap(func)</code>	Similar to map, but each input item can be mapped to 0 or more output items (so <i>func</i> should return a Seq rather than a single item).
<code>mapPartitions(func)</code>	Similar to map, but runs separately on each partition (block) of the RDD, so <i>func</i> must be of type <code>Iterator<T> => Iterator<U></code> when running on an RDD of type T.
<code>mapPartitionsWithinIndex(func)</code>	Similar to mapPartitions, but also provides <i>func</i> with an integer value representing the index of the partition, so <i>func</i> must be of type <code>(Int, Iterator<T>) => Iterator<U></code> when running on an RDD of type T.
<code>sample(withReplacement, fraction, seed)</code>	Sample a fraction <i>fraction</i> of the data, with or without replacement, using a given random number generator seed.
<code>union(otherDataset)</code>	Return a new dataset that contains the union of the elements in the source dataset and the argument.



Spark支持的一些常用transformation操作

Transformation	Meaning
<code>intersection(<i>otherDataset</i>)</code>	Return a new RDD that contains the intersection of elements in the source dataset and the argument.
<code>distinct([<i>numTasks</i>])</code>	Return a new dataset that contains the distinct elements of the source dataset.
<code>groupByKey([<i>numTasks</i>])</code>	When called on a dataset of (K, V) pairs, returns a dataset of (K, Iterable<V>) pairs.
<code>reduceByKey(<i>func</i>, [<i>numTasks</i>])</code>	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 <i>func</i> , which must be of type (V,V) => V. Like in groupByKey, the number of reduce tasks is configurable through an optional second argument.
<code>aggregateByKey(<i>zeroValue</i>)(<i>seqOp</i>, <i>combOp</i>, [<i>numTasks</i>])</code>	When called on a dataset of (K, V) pairs, returns a dataset of (K, U) pairs where the values for each key are aggregated using the given combine functions and a neutral "zero" value. Allows an aggregated value type that is different than the input value type, while avoiding unnecessary allocations. Like in groupByKey, the number of reduce tasks is configurable through an optional second argument.
<code>sortByKey([<i>ascending</i>], [<i>numTasks</i>])</code>	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.



Spark支持的一些常用transformation操作

Transformation	Meaning
<code>join(otherDataset, [numTasks])</code>	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 supported through <code>leftOuterJoin</code> , <code>rightOuterJoin</code> , and <code>fullOuterJoin</code> .
<code>cogroup(otherDataset, [numTasks])</code>	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 <code>groupWith</code> .
<code>cartesian(otherDataset)</code>	When called on datasets of types T and U, returns a dataset of (T, U) pairs (all pairs of elements).
<code>pipe(command, [envVars])</code>	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.
<code>coalesce(numPartitions)</code>	Decrease the number of partitions in the RDD to <code>numPartitions</code> . Useful for running operations more efficiently after filtering down a large dataset.
<code>repartition(numPartitions)</code>	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.
<code>repartitionAndSortWithinPartitions(partitioner)</code>	Repartition the RDD according to the given partitioner and, within each resulting partition, sort records by their keys. This is more efficient than calling <code>repartition</code> and then sorting within each partition because it can push the sorting down into the shuffle machinery.



Spark支持的一些常用action操作

46

Action	Meaning
<code>reduce(func)</code>	Aggregate the elements of the dataset using a function <i>func</i> (which takes two arguments and returns one). The function should be commutative and associative so that it can be computed correctly in parallel.
<code>collect()</code>	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.
<code>count()</code>	Return the number of elements in the dataset.
<code>first()</code>	Return the first element of the dataset (similar to <code>take(1)</code>).
<code>take(n)</code>	Return an array with the first <i>n</i> elements of the dataset.
<code>takeSample(withReplacement, num, [seed])</code>	Return an array with a random sample of <i>num</i> elements of the dataset, with or without replacement, optionally pre-specifying a random number generator seed.
<code>takeOrdered(n, [ordering])</code>	Return the first <i>n</i> elements of the RDD using either their natural order or a custom comparator.



Spark支持的一些常用action操作

47

Action	Meaning
<code>saveAsTextFile(path)</code>	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 <code>toString</code> on each element to convert it to a line of text in the file.
<code>saveAsSequenceFile(path)</code> (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 <code>Int</code> , <code>Double</code> , <code>String</code> , etc).
<code>saveAsObjectFile(path)</code> (Java and Scala)	Write the elements of the dataset in a simple format using Java serialization, which can then be loaded using <code>SparkContext.objectFile()</code> .
<code>countByKey()</code>	Only available on RDDs of type <code>(K, V)</code> . Returns a hashmap of <code>(K, Int)</code> pairs with the count of each key.
<code>foreach(func)</code>	Run a function <i>func</i> 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.



RDD的容错实现

48

□ 在RDD中，存在两种容错的方式：

▣ Lineage（血统系统、依赖系统）

- RDD提供一种基于粗粒度变换的接口，这使得RDD可以通过记录RDD之间的变换，而不需要存储实际的数据就可以完成数据的恢复，使得Spark具有高效的容错性。

▣ CheckPoint（检查点）

- 对于很长的lineage的RDD来说，通过lineage来恢复耗时较长。因此，在对包含宽依赖的长血统的RDD设置检查点操作非常有必要。
- 由于RDD的只读特性使得Spark比常用的共享内存更容易完成checkpoint。

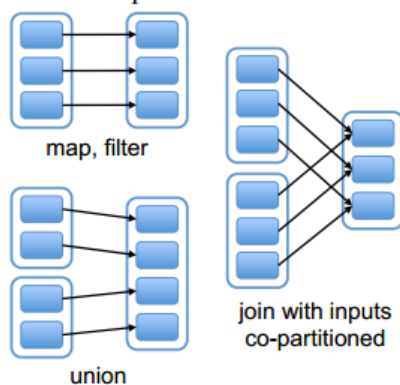


RDD之间的依赖关系

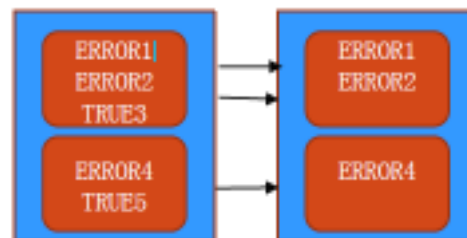
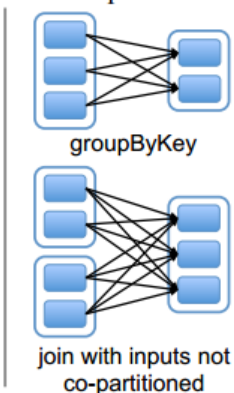
49

- 在Spark中存在两种类型的依赖：
 - 窄依赖：父RDD中的一个Partition最多被子RDD中的一个Partition所依赖
 - 例：`val filterRDD = fileRDD.filter(line=>line.contains("ERROR"))`
 - 宽依赖：父RDD中的一个Partition被子RDD中的多个Partition所依赖

Narrow Dependencies:



Wide Dependencies:





根据RDD分区的依赖关系划分阶段

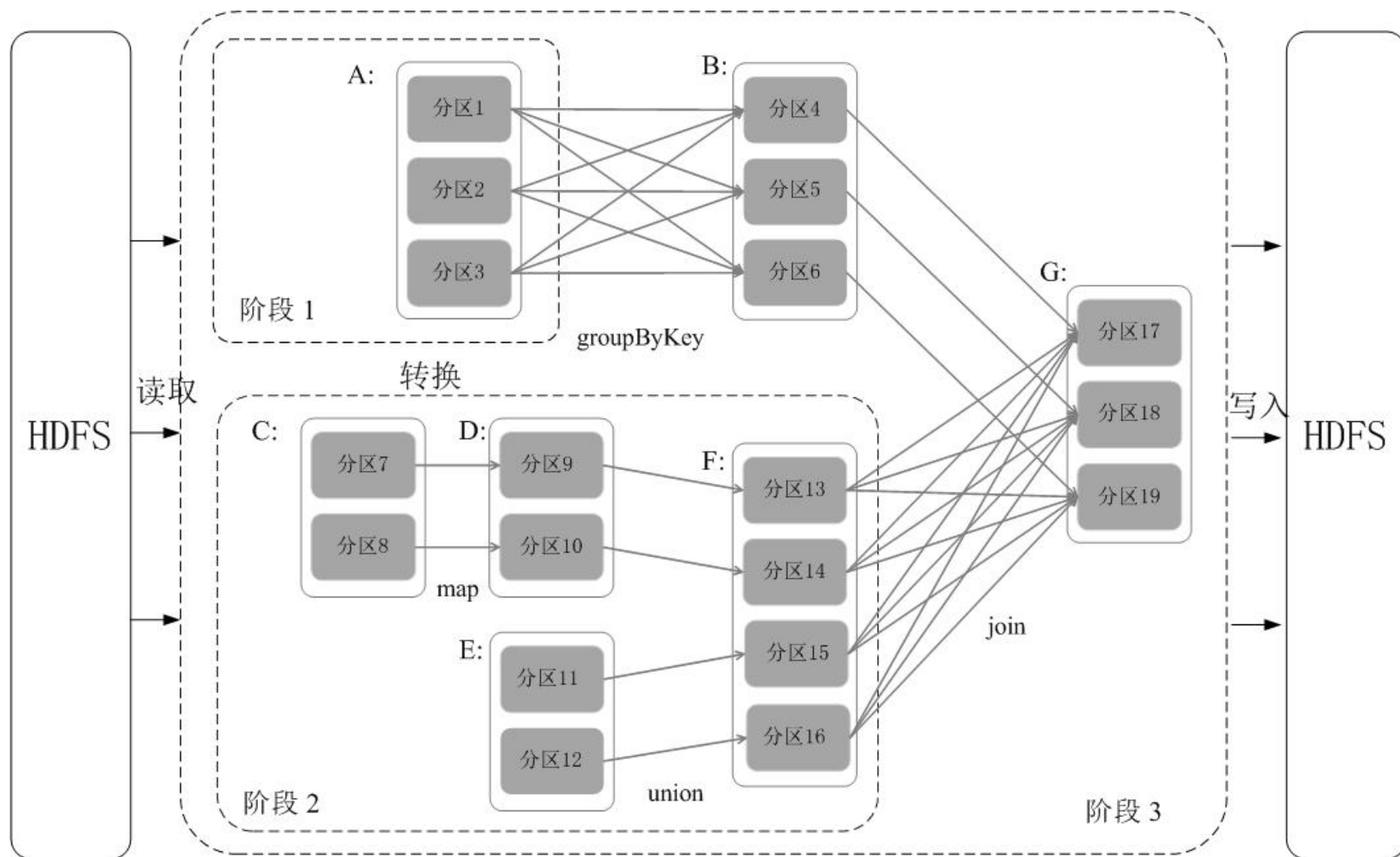
50

- **Spark**通过分析各个**RDD**的依赖关系生成了**DAG**，再通过分析各个**RDD**中的分区之间的依赖关系来决定如何划分阶段，具体划分方法是：在**DAG**中进行反向解析，遇到宽依赖就断开，遇到窄依赖就把当前的**RDD**加入到当前的阶段中；将窄依赖尽量划分在同一个阶段中，可以实现流水线计算。
- 把一个**DAG**图划分成多个“阶段”以后，每个阶段都代表了一组关联的、相互之间没有**Shuffle**依赖关系的任务组成的任务集合。每个任务集合会被提交给任务调度器（**TaskScheduler**）进行处理，由任务调度器将任务分发给**Executor**运行。



根据RDD分区的依赖关系划分阶段

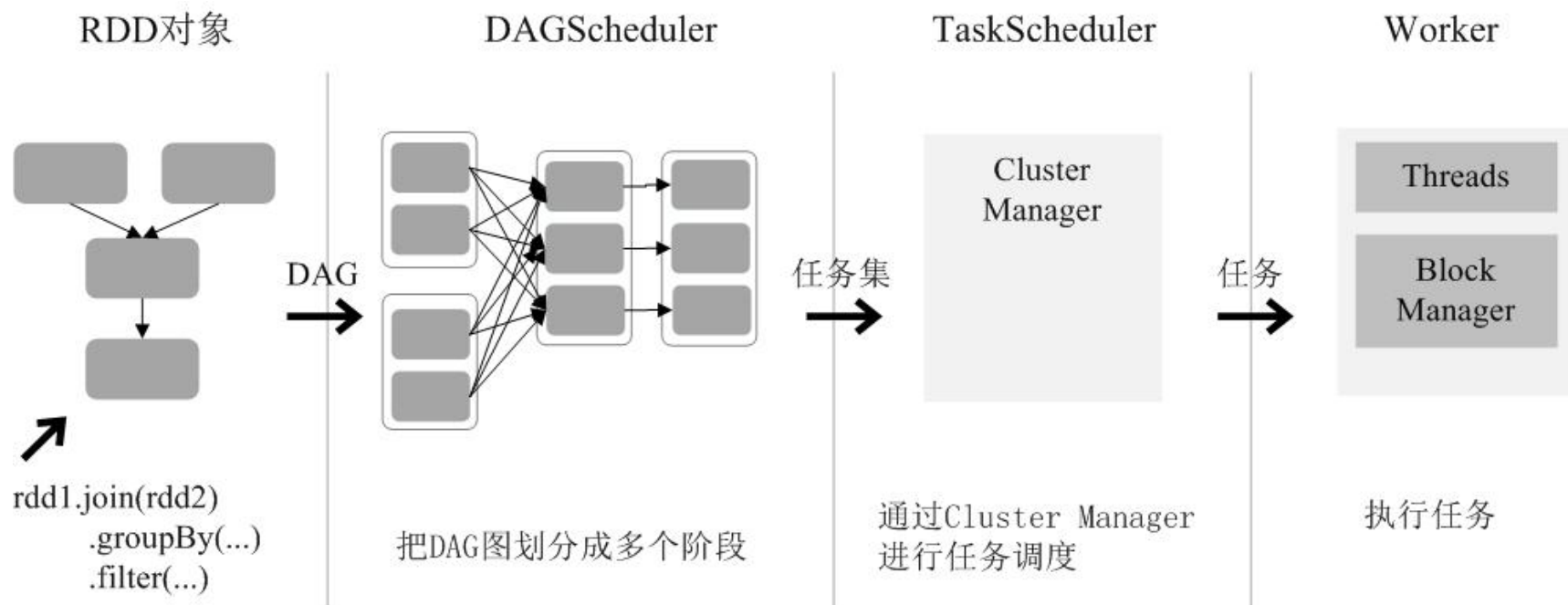
51





RDD的运行

52





RDD持久化

53

- **Spark**提供了三种对持久化**RDD**的存储策略：
 - ▣ 未序列化的**Java**对象，存在内存中
 - 性能表现最优，可以直接访问在**JAVA**虚拟机内存里的**RDD**对象。
 - ▣ 序列化的数据，存于内存中
 - 取消**JVM**中的**RDD**对象，将对象的状态信息转换为可存储形式，减小**RDD**的存储开销，但使用时需要反序列化恢复。
 - 在内存空间有限的情况下，这种方式可以让用户更有效的使用内存，但是这么做的代价是降低了性能。
 - ▣ 磁盘存储
 - 适用于**RDD**太大难以在内存中存储的情形，但每次重新计算该**RDD**都会带来巨大的额外开销。



RDD持久化

54

```
scala> val list = List("Hadoop","Spark","Hive")
```

```
list: List[String] = List(Hadoop, Spark, Hive)
```

```
scala> val rdd = sc.parallelize(list)
```

```
rdd: org.apache.spark.rdd.RDD[String] = ParallelCollectionRDD[22] at parallelize at  
<console>:29
```

`scala> rdd.cache()` //会调用`persist(MEMORY_ONLY)`，但是，语句执行到这里，并不会缓存rdd，这时rdd还没有被计算生成

`scala> println(rdd.count())` //第一次行动操作，触发一次真正从头到尾的计算，这时才会执行上面的`rdd.cache()`，把这个rdd放到缓存中

3

`scala> println(rdd.collect().mkString(","))` //第二次行动操作，不需要触发从头到尾的计算，只需要重复使用上面缓存中的rdd

Hadoop,Spark,Hive



完整的存储级别介绍

55

Storage Level	Meaning
MEMORY_ONLY	将RDD作为非序列化的Java对象存储在jvm中。如果RDD不适合存在内存中，一些分区将不会被缓存，从而在每次需要这些分区时都需重新计算它们。这是系统默认的存储级别。
MEMORY_AND_DISK	将RDD作为非序列化的Java对象存储在jvm中。如果RDD不适合存在内存中，将这些不适合存在内存中的分区存储在磁盘中，每次需要时读出它们。
MEMORY_ONLY_SER	将RDD作为序列化的Java对象存储（每个分区一个byte数组）。这种方式比非序列化方式更节省空间，特别是用到快速的序列化工具时，但是会更耗费cpu资源—密集的阅读操作。
MEMORY_AND_DISK_SER	和MEMORY_ONLY_SER类似，但不是在每次需要时重复计算这些不适合存储到内存中的分区，而是将这些分区存储到磁盘中。
DISK_ONLY	仅仅将RDD分区存储到磁盘中
MEMORY_ONLY_2, MEMORY_AND_DISK_2, etc.	和上面的存储级别类似，但是复制每个分区到集群的两个节点上面
OFF_HEAP (experimental)	以序列化的格式存储RDD到Tachyon中。相对于MEMORY_ONLY_SER，OFF_HEAP减少了垃圾回收的花费，允许更小的执行者共享内存池。这使其在拥有大量内存的环境下或者多并发应用程序的环境中具有更强的吸引力。



RDD 内部设计

56

- 每个RDD都包含：
 - ▣ 一组RDD分区（**partition**），即数据集的原子组成部分
 - ▣ 对父RDD的一组依赖，这些依赖描述了RDD的**Lineage**
 - ▣ 一个函数，即在父RDD上执行何种计算
 - ▣ 元数据，描述分区模式和数据存放的位置



RDD 内部接口

57

操作	含义
<code>partitions()</code>	返回一组Partition对象
<code>preferredLocations(p)</code>	根据数据存放的位置， 返回分区p在哪些节点访问更快
<code>dependencies()</code>	返回一组依赖
<code>iterator(p, parentIters)</code>	按照父分区的迭代器，逐个计算分区p的元素
<code>partitioner()</code>	返回RDD是否hash/range分区的元数据信息



分区

58

- **RDD**是弹性分布式数据集，通常**RDD**很大，会被分成很多个分区，分别保存在不同的节点上。**RDD**分区的一个分区原则是使得分区的个数尽量等于集群中的**CPU**核心（**core**）数目。
- 对于不同的**Spark**部署模式而言，都可以通过设置**spark.default.parallelism**这个参数的值，来配置默认的分区数目，一般而言：
 - ▣ 本地模式：默认为本地机器的**CPU**数目，若设置了**local[N]**,则默认为**N**；
 - ▣ **Apache Mesos**：默认的分区数为**8**；
 - ▣ **Standalone**或**YARN**：在“集群中所有**CPU**核心数目总和”和“**2**”二者中取较大值作为默认值；



分区

59

- 因此，对于`parallelize`而言，如果没有在方法中指定分区数，则默认为`spark.default.parallelism`

```
scala>val array = Array(1, 2, 3, 4, 5)
```

```
array: Array[Int] = Array(1, 2, 3, 4, 5)
```

```
scala>val rdd = sc.parallelize(array,2) #设置两个分区
```

```
rdd: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[13] at  
parallelize at <console>:29
```

- 对于`textFile`而言，如果没有在方法中指定分区数，则默认为`min(defaultParallelism,2)`，其中，`defaultParallelism`对应的就是`spark.default.parallelism`。
- 如果是从HDFS中读取文件，则分区数为文件分片数(比如，128MB/片)。



Spark编程接口

60

- Spark用Scala语言实现了RDD的API
- Scala是一种基于JVM的静态类型、函数式、面向对象的语言
- Scala具有简洁（特别适合交互式使用）、有效（因为是静态类型）等优点
- Spark支持多种语言的API:
 - ▣ Scala
 - ▣ Python
 - ▣ Java
 - ▣ R



摘要

- Spark安装运行
- Spark编程模型
- Spark编程示例



Spark编程示例

62

- Word count
- K-Means 聚类



WordCount MapReduce代码

63

□ Map类代码

//定义Map类实现字符串分解

```
public static class TokenizerMapper extends Mapper<Object, Text, Text, IntWritable> {  
    private final static IntWritable one = new IntWritable(1);  
    private Text word = new Text();  
    //实现map()函数  
    public void map(Object key, Text value, Context context) throws IOException, InterruptedException {  
        //将字符串拆解成单词  
        StringTokenizer itr = new StringTokenizer(value.toString());  
        while (itr.hasMoreTokens()) {  
            word.set(itr.nextToken()); //将分解后的一个单词写入word类  
            context.write(word, one); //收集<key, value>  
        }  
    }  
}
```



WordCount MapReduce 代码

64

□ Reduce 类代码

//定义Reduce类规约同一key的value

```
public static class IntSumReducer extends Reducer <Text, IntWritable, Text,
IntWritable>{

    private IntWritable result = new IntWritable();//实现reduce()函数

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

        int sum = 0;//遍历迭代values，得到同一key的所有value
        for (IntWritable val : values) { sum += val.get(); }

        result.set(sum);//产生输出对<key, value>

        context.write(key, result);

    }

}
```

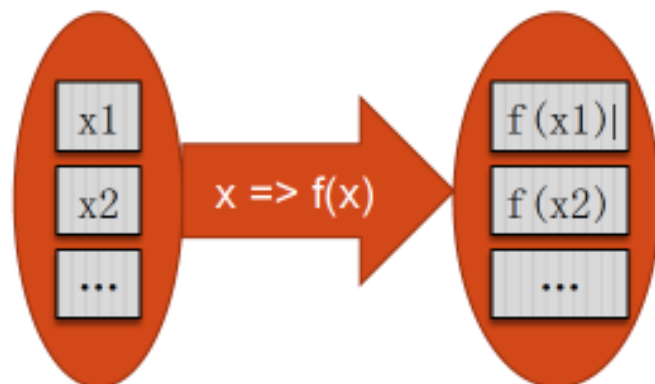



WordCount Spark Scala代码

65

```
val file = spark.textFile("hdfs://...")  
val counts = file.flatMap(line => line.split(" ")) //分词  
                  .map(word => (word, 1)) //对应mapper的工作  
                  .reduceByKey(_ + _) //相同key的不同value之间进行"+"运算  
counts.saveAsTextFile("hdfs://...")
```

这里，map操作表示对列表中的每个元素应用一个函数，在scala中，函数可以写成“ $x \Rightarrow f(x)$ ”的形式，也可以更简洁地写成 $f(_)$ 。例如 $line \Rightarrow line.split(" ")$ 可以简写为 $_.split(" ")$ 。flatMap是在map之后增加了一个“扁平化”的操作，将map之后可能形成的形如 $List(List(1,2), List(3,4))$ 的数据集“扁平”为 $List(1,2,3,4)$ 。





WordCount Spark Java代码

66

```
JavaRDD<String> file = spark.textFile("hdfs://...");

JavaRDD<String> words = file.flatMap(new FlatMapFunction<String, String>(){
    public Iterable<String> call(String s) {
        return Arrays.asList(s.split(" "));
    }
}); // 对应 flatMap(line => line.split(" ")) 操作

JavaPairRDD<String, Integer> pairs = words.mapToPair( new PairFunction<String, String,
Integer>(){
    public Tuple2<String, Integer> call(String s){
        return new Tuple2<String, Integer>(s, 1);
    }
}); // 对应 map(word => (word, 1))

JavaPairRDD<String, Integer> counts = pairs.reduceByKey(new Function2<Integer, Integer>(){
    public Integer call(Integer a, Integer b) {
        return a + b;
    }
}); // 对应 reduceByKey(_ + _) counts.saveAsTextFile("hdfs://...");
```



WordCount Spark Java8代码

67

Java8引入了**Lambda**表达式（“闭包”或“匿名方法”），**Lambda**表达式允许通过表达式来代替功能接口。

```
JavaRDD<String> lines = spark.read().textFile(args[0]).javaRDD();
JavaRDD<String> words = lines.flatMap(s -> Arrays.asList(SPACe.split(s)).
iterator());
JavaPairRDD<String, Integer> ones = words.mapToPair(s -> new Tuple2<>(s,
1));
JavaPairRDD<String, Integer> counts = ones.reduceByKey((i1, i2) -> i1 + i2);
List<Tuple2<String, Integer>> output = counts.collect();
for (Tuple2<?, ?> tuple : output) {
    System.out.println(tuple._1() + ": " + tuple._2());
}
```



K-Means 算法示例

68

- 1. 基于MapReduce的K-Means聚类算法
- 2. 基于Spark的K-Means聚类算法
- 3. 基于Spark的K-Means聚类算法代码



基于MapReduce的K-Means聚类算法

69

□ 算法设计思路

- ▣ 将所有数据分布到不同的**MapReduce**节点上，每个节点只对自己的数据进行计算
- ▣ 每个**Map**节点能够读取上一次迭代生成的**cluster centers**，并判断自己的各个数据点应该属于哪一个**cluster**
- ▣ **Reduce**节点综合每个属于每个**cluster**的数据点，计算出新的**cluster centers**



基于MapReduce的K-Means聚类算法

70

□ Map阶段的处理

- 在Map类的初始化方法**setup**中读取全局的聚类中心信息
- 对Map方法收到的每一个数据点 p ，计算 p 与所有聚类中心间的距离，并选择一个距离最小的中心作为 p 所属的聚类，输出 $\langle \text{ClusterID}, (p,1) \rangle$ 键值对
- 对每个Map节点上即将传递到Reduce节点的每一个 $\langle \text{ClusterID}, (p,1) \rangle$ 键值对，用Combiner进行数据优化，合并相同ClusterID下的所有数据点并求取这些点的均值 p_m 以及数据点个数 n



基于MapReduce的K-Means聚类算法

71

□ Reduce阶段的处理

- 经过Map和Combine后从Map节点输出的所有ClusterID相同的中间结果 $\langle \text{ClusterID}, [(pm1, n1), (pm2, n3)...] \rangle$ ，计算新的均值 pm ，输出 $\langle \text{ClusterID}, pm \rangle$
- 所有输出的 $\langle \text{ClusterID}, (pm, n) \rangle$ 形成新的聚类中心，供下一次迭代计算



基于MapReduce的K-Means聚类算法

72

□ 性能分析

▣ 优点

- 相比于单机运行的K-Means算法，MapReduce通过并行计算每个MapReduce节点上的数据到cluster centers的距离，显著提高了K-Means算法的效率

▣ 不足

- MapReduce每次执行迭代操作都被作为独立作业重新进行处理，需要重新初始化和读写、传输数据
- MapReduce每次迭代很可能存在大量不变数据，而每次都要重新载入和处理
- MapReduce每次迭代需要一个额外的MapReduce Job用来检测迭代终止条件
- MapReduce下一轮迭代必须等待上一轮迭代终止，必须重新从分布式文件系统载入数据



基于Spark的K-Means聚类算法

73

- **Spark并行化K-Means算法设计思路**
 - ▣ 将所有数据分布到集群中所有节点的**RDD**内存数据结构中，每个节点计算自己内存中的数据
 - ▣ 每个节点读取上一次迭代生成的**cluster centers**，并判断自己内存中的数据点应该属于哪一个**cluster**
 - ▣ 汇总每个节点更新的信息，综合每个属于每个**cluster**的数据点，计算出新的**cluster centers**



基于Spark的K-Means聚类算法

74

□ Spark并行化K-Means算法流程

- ▣ 1.从HDFS上读取数据转化为RDD，将RDD中的每个数据对象转化为向量形成新的RDD存入缓存，随机抽样K个向量作为全局初始聚类中心
- ▣ 2.计算RDD中的每个向量p到聚类中心cluster centers的距离，将向量划分给最近的聚类中心，生成以<ClusterID, (p, 1)>为元素的新的RDD
- ▣ 3.聚合新生成的RDD中Key相同的<ClusterID, (p, 1)>键值对，将相同ClusterID下的所有向量相加并求取向量个数n，生成新的RDD



基于Spark的K-Means聚类算法

75

- Spark并行化K-Means算法流程
 - ▣ 4. 对生成的RDD中每一个元素 $\langle \text{ClusterID}, (pm, n) \rangle$ ，计算ClusterID聚类的新的聚类中心，生成以 $\langle \text{ClusterID}, pm/n \rangle$ 为元素的新的RDD
 - ▣ 5. 判断是否达到最大迭代次数或者迭代是否收敛，不满足条件则重复步骤2到步骤5，满足则结束，输出最后的聚类中心



基于Spark的K-Means聚类算法

76

□ 性能分析

- **Spark**的大部分操作都是在内存中完成的，相比于**MapReduce**每次从分布式文件系统中获取数据要高效
- **Spark**的所有迭代操作都在一个**Job**中完成，相比于**MapReduce**没有重启多次**Job**带来的开销
- **Spark**任务执行结束直接退出，不需要另外一个**Job**来检测迭代终止条件



基于Spark的K-Means聚类算法Scala代码

77

□ Scala示例

▣ 读取数据和初始化聚类中心

```
val lines = sc.textFile("data/mllib/kmeans_data.txt" )  
val data = lines.map(s => s.split(" ").map(_.toDouble)).cache()  
val kPoints= data.takeSample(false, K, 42).map(s => spark.util.Vector(s))  
//takeSample(Boolean, Int, Long)采样函数， false表示不使用替换方法采样，  
K表示样本数， 42表示随机种子
```

▣ 划分数据给聚类中心

```
val closest = data.map// 产生<ClusterID, (p, 1)>键值对  
(p =>  
  ( closestPoint(spark.util.Vector(p), kPoints),  
    //closestPoint计算最近的聚类中心， 产生ClusterID (spark.util.Vector(p), 1)  
  ))
```



基于Spark的K-Means聚类算法Scala代码

78

□ Scala示例

▣ 聚合生成新的聚类中心

//同一个聚类下所有向量相加并统计向量个数

```
val pointStats= closest.reduceByKey{  
    case ((x1, y1), (x2, y2)) => (x1 + x2, y1 + y2) //产生(pm, n)  
    } //将同一clusterID的所有(p, 1)的两个分量分别相加，得到<ClusterID, (pm, n)>  
//计算生成新的聚类中心
```

```
val newPoints= pointStats.map {  
    pair => (pair._1, pair._2._1 / pair._2._2).collectAsMap()  
//由<ClusterID, (pm, n)>产生(ClusterID, pm/n)。其中，pair._1表示聚类的ClusterID，  
pair._2._1表示聚类中所有向量之和pm，pair._2._2表示聚类中所有向量的个数n
```



基于Spark的K-Means聚类算法Java代码

79

□ Java示例

▣ 读取数据和初始化聚类中心

```
JavaRDD<String> data = sc.textFile(path);

JavaRDD<Vector> parsedData= data.map(
    new Function<String, Vector>() {
        public Vector call(String s) {
            String[] sarray= s.split(" ");
            Vector<Double> values = new Vector<Double>();
            for (inti=0; i< sarray.length; i++)
                values.add(Double.parseDouble(sarray[i]));
            return values;
        }
    }); //读取数据

parsedData.cache();//缓存数据

final List<Vector> kPoints= parsedData.takeSample(false, 2, 42); //初始化聚类中心。

//其中takeSample为采样函数，false表示不使用替换方法采样，2表示样本数，42表示随机种子
```



基于Spark的K-Means聚类算法Java代码

80

□ 划分数据给聚类中心

```
JavaPairRDD<Integer, Tuple2<Vector, Integer>>
```

```
    closest = parsedData.mapToPair(
```

```
        new PairFunction<Vector, Integer, Tuple2<Vector, Integer>>() {
```

```
            public Tuple2<Integer, Tuple2<Vector, Integer>> call(Vector p) {
```

```
                int clusterID= kmeans.closestPoint(p, kPoints);
```

```
                //closestPoint计算最近的聚类中心，产生ClusterID
```

```
                Tuple2<Vector, Integer> pair = new Tuple2<Vector, Integer>(p, 1);
```

```
                return new Tuple2(clusterID, pair);
```

```
            }
```

```
        }
```

```
    );
```




基于Spark的K-Means聚类算法Java代码

81

□ 聚合同一个聚类中的向量

```
JavaPairRDD<Integer, Tuple2<Vector, Integer>> pointStats= closest.reduceByKey(  
    new Function2<Tuple2<Vector, Integer>, Tuple2<Vector, Integer>, Tuple2<Vector, Integer>>() {  
        public Tuple2<Vector, Integer> call(Tuple2<Vector, Integer> tuple1, Tuple2<Vector,  
Integer> tuple2) {  
            Vector newv= new Vector<Double>(); //newv统计聚类中向量之和  
            int count = 0; //count统计聚类中向量个数  
            for (int i= 0; i< tuple1._1().size() && i< tuple2._1().size(); i++) {  
                newv.add((double) tuple1._1().get(i) + (double) tuple2._1().get(i));  
                count = tuple1._2() + tuple2._2();  
            }  
            return new Tuple2<Vector, Integer>(newv, count);  
        }  
    }  
);
```



基于Spark的K-Means聚类算法Java代码

82

□ 生成新的聚类中心

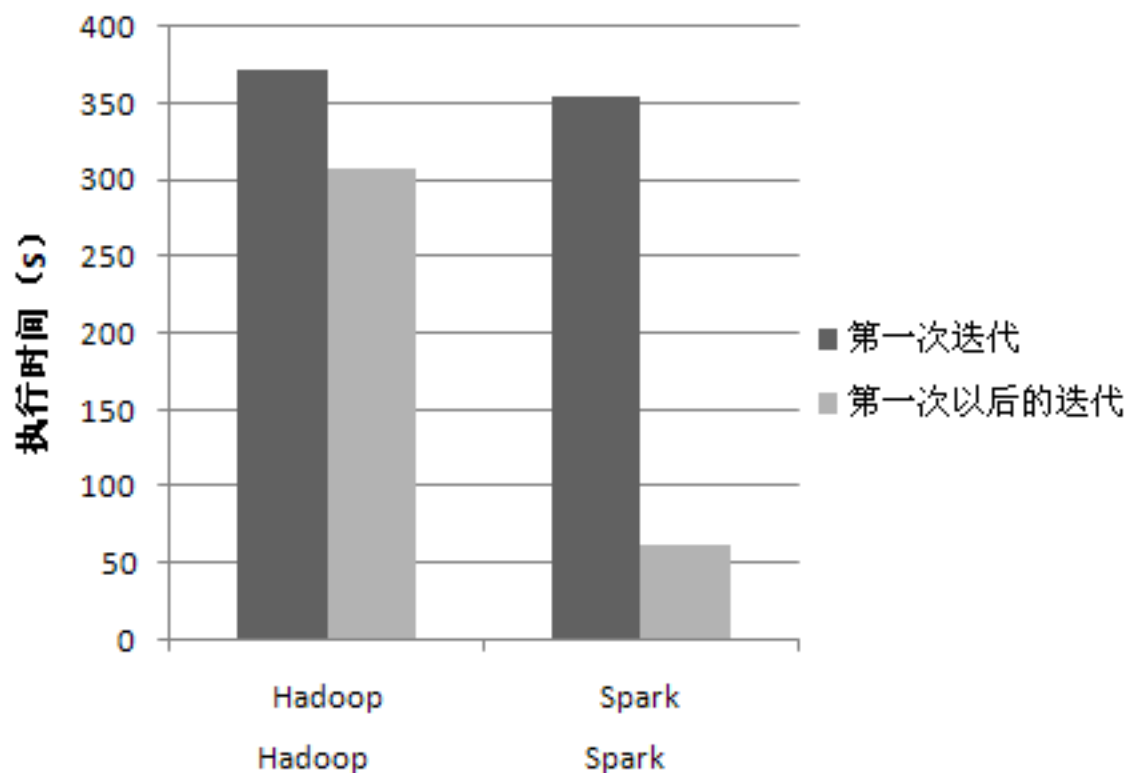
```
JavaPairRDD<Integer, Vector> newPointsStats= pointStats.mapToPair(  
    new PairFunction<Tuple2<Integer, Tuple2<Vector, Integer>>, Integer, Vector>() {  
        public Tuple2<Integer, Vector> call(Tuple2<Integer, Tuple2<Vector, Integer>> tuple) {  
            int clusterID= tuple._1();           //clusterID表示聚类的类别  
            Vector newv= new Vector<Double>(); //newv表示聚类中的向量之和  
            int count = tuple._2()._2();         //count表示聚类中向量个数  
            for (inti= 0; i< tuple._2()._1().size(); i++) {  
                newv.add((double) tuple._2()._1().get(i) * 1.0 / count);  
            }  
            return new Tuple2<Integer, Vector>(clusterID, newv);  
        }  
    });  
Map<Integer, Vector> newPoint= newPointsStats.collectAsMap();
```



基于Spark的K-Means聚类算法代码

83

□ 性能比较





参考文献

84

- Spark官方网站<http://spark.apache.org/>
- Spark主要开发者Matei Zaharia的博士论文：Zaharia M. An architecture for fast and general data processing on large clusters[R]. Technical Report No. UCB/EECS-2014-12, 3 Feb 2014. <http://www.eecs.berkeley.edu/Pubs/TechRpts/2014/EECS-2014-12.html>, 2014.
- Mesos官方网站<http://mesos.apache.org/>
- Kubernetes官方网站<https://kubernetes.io>
- Docker官方网站<https://www.docker.com/>
- Hadoop官方网站中关于YARN的介绍
<http://hadoop.apache.org/docs/current/hadoop-yarn/hadoop-yarn-site/YARN.html>