task one

spark和hadoop安装

因为我电脑之前已经安装过spark、hadoop以及pyspark环境所以这里不再赘述，

spark和hadoop启动后jps命令结果

```
mayfly@mayfly:~/workstation/env/spark$ jps
12208 SparkSubmit
13539 Master
13797 Jps
13061 NodeManager
13721 Worker
12395 DataNode
12093 NameNode
12861 ResourceManager
12671 SecondaryNameNode
```

启动时可能遇到的问题

使用sh命令调用start-all.sh可能会提示错误

```
mayfly@mayfly:~/workstation/env/hadoop$ sh sbin/start-all.sh
This script is Deprecated. Instead use start-dfs.sh and start-yarn.sh
sbin/start-all.sh: 111: /home/mayfly/workstation/env/hadoop/sbin/../libexec/hadoop-config.sh: Syntax error: word unexpec
ted (expecting ")")
```

问题解决方式

```
- 输入: bash start-all.sh
- 输入: start-all.sh
```

原因：bash 和 sh 不同：一般的，sh是bash的“子集”（不是子集的部分，具体区别见下的“Things sh has that bash does not”）

1. sh一般设置成bash的软链接；

   在一般的linux操作系统之中，使用sh等于打开了bash的posix的标准模式，等于执行 bash --posix

2. sh与bash的区别相当于
   bash的posix标准模式与bash的区别，其中就有let截取字符串等。

   “sh”并不是一个程序，而是一种标准（POSIX），这种标准，在一定程度上保证了脚本的跨系统性（跨UNIX系统）

spark背景介绍

主要特点

1. 运行速度快：使用DAG(Directed Acyclic Graph，有向无环图)执行引擎，以支持数据流和内存计算，基于内存的执行速度可比Hadoop MapReduce快上百倍，基于磁盘的执行速度也能快十倍
2. 容易使用：支持使用Scala、Java、Python和R语言进行编程，简洁的API设计有助于用户轻松构建并行程序，并且可以通过Spark Shell进行交互式编程
3. 通用性：提供了完整而强大的技术栈，包括SQL查询、流式计算、机器学习和图算法组件，这些组件可以无缝整合在同一个应用中，足以应对复杂的计算
4. 运行模式多样：可运行于独立的集群模式中，或者运行于Hadoop中，也可运行于Amazon EC2等云环境中，并且可以访问HDFS、Cassandra、HBase、Hive等多种数据源

spark和hadoop比较

mapreduce计算模型延迟过高，不能胜任实时快速计算任务，只适用于离线批处理应用场景

Spark最大的特点就是将计算数据、中间结果都存储在内存中，大大减少了IO开销，因而，Spark更适合于迭代运算比较多的数据挖掘与机器学习运算。使用Hadoop进行迭代计算非常耗资源，因为每次迭代都需要从磁盘中写入、读取中间数据，IO开销大。而Spark将数据载入内存后，之后的迭代计算都可以直接使用内存中的中间结果作运算，避免了从磁盘中频繁读取数据。

hadoop缺陷

1. 表达能力有限：计算必须转换为map和reduce两个操作，难以描述复杂的数据处理过程
2. 磁盘IO开销大：每次执行时需要读写磁盘
3. 延迟高：一次计算可能需要分解成一系列按顺序执行的mapreduce任务，任务之间的衔接涉及到IO开销所以延迟较高，而且任务只能串行执行

spark优点

1. 计算模式也属于mapreduce但是不局限于map和reduce操作，还有多种数据集操作类型，编程模型比mapreduce更灵活
2. 提供了内存计算，中间结果直接放到内存中，带来了更高的迭代运算效率
3. 基于DAG的任务调度执行机制，优于mapreduce的迭代执行机制

实际应用中，大数据处理的主要类型

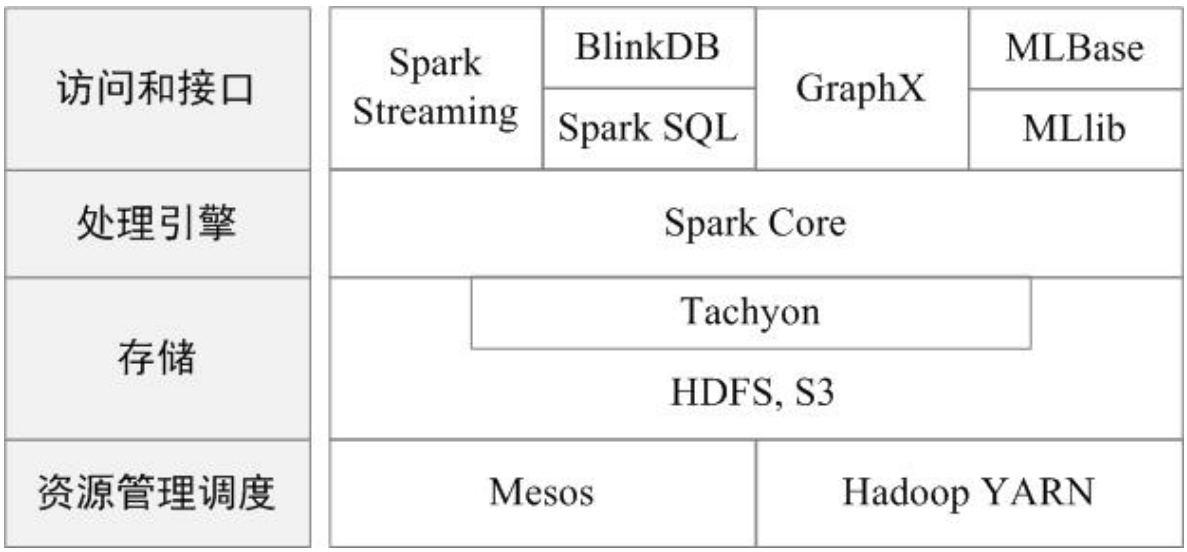
1. 复杂的批量数据处理：时间跨度通常在数十分钟到数小时之间；
2. 基于历史数据的交互式查询：时间跨度通常在数十秒到数分钟之间；
3. 基于实时数据流的数据处理：时间跨度通常在数百毫秒到数秒之间。

使用不同的软件处理这三种类型存在的问题

1. 不同场景之间输入输出数据无法做到无缝共享，通常需要进行数据格式的转换
2. 不同的软件需要不同的开发和维护团队，带来了较高的使用成本
3. 比较难以对同一个集群中的各个系统进行统一的资源协调和分配。

Spark的设计遵循“一个软件栈满足不同应用场景”的理念，逐渐形成了一套完整的生态系统，既能够提供内存计算框架，也可以支持SQL即席查询、实时流式计算、机器学习和图计算等。Spark可以部署在资源管理器YARN之上，提供一站式的大数据解决方案。因此，Spark所提供的生态系统足以应对上述三种场景，即同时支持批处理、交互式查询和流数据处理。

BDAS架构



spark组件

1. Spark Core：Spark Core包含Spark的基本功能，如内存计算、任务调度、部署模式、故障恢复、存储管理等。Spark建立在统一的抽象RDD之上，使其可以以基本一致的方式应对不同的大数据处理场景；通常所说的Apache Spark，就是指Spark Core；
2. Spark SQL：Spark SQL允许开发人员直接处理RDD，同时也可查询Hive、HBase等外部数据源。Spark SQL的一个重要特点是其能够统一处理关系表和RDD，使得开发人员可以轻松地使用SQL命令进行查询，并进行更复杂的数据分析；
3. Spark Streaming：Spark Streaming支持高吞吐量、可容错处理的实时流数据处理，其核心思路是将流式计算分解成一系列短小的批处理作业。Spark Streaming支持多种数据输入源，如Kafka、Flume和TCP套接字等；
4. MLlib（机器学习）：MLlib提供了常用机器学习算法的实现，包括聚类、分类、回归、协同过滤等，降低了机器学习的门槛，开发人员只要具备一定的理论知识就能进行机器学习的工作；
5. GraphX（图计算）：GraphX是Spark中用于图计算的API，可认为是Pregel在Spark上的重写及优化，Graphx性能良好，拥有丰富的功能和运算符，能在海量数据上自如地运行复杂的图算法。

spark运行架构

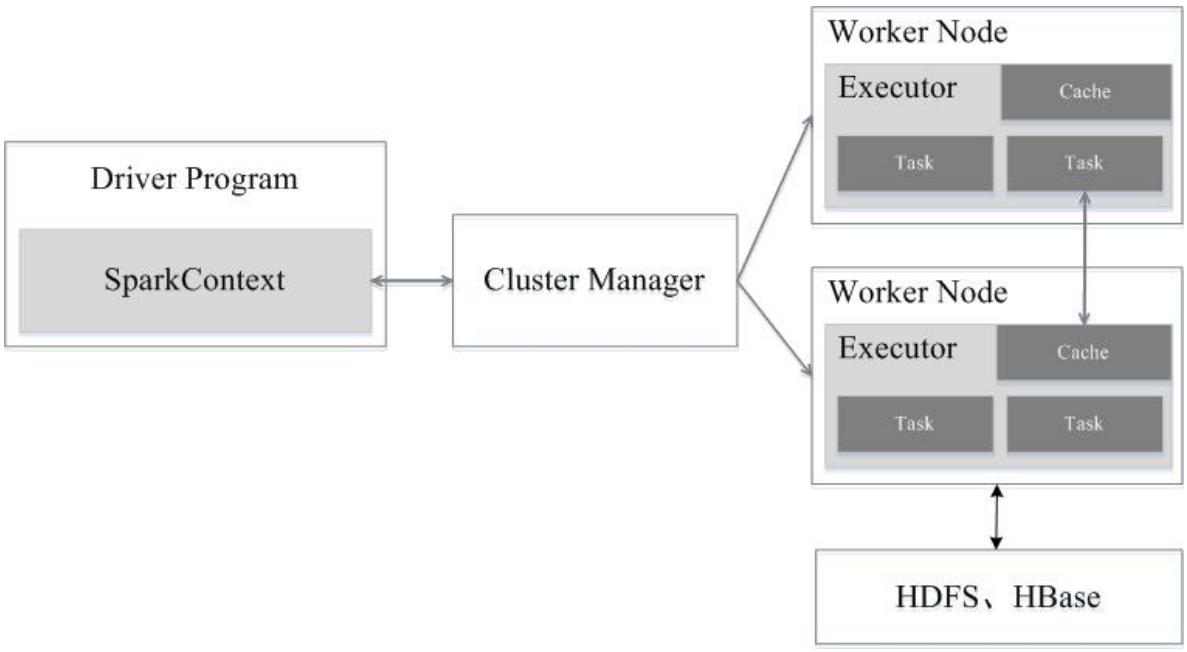
基本概念

- RDD：是弹性分布式数据集（Resilient Distributed Dataset）的简称，是分布式内存的一个抽象概念，提供了一种高度受限的共享内存模型；
- DAG：是Directed Acyclic Graph（有向无环图）的简称，反映RDD之间的依赖关系；
- Executor：是运行在工作节点（Worker Node）上的一个进程，负责运行任务，并为应用程序存储数据；
- 应用：用户编写的Spark应用程序；
- 任务：运行在Executor上的工作单元；
- 作业：一个作业包含多个RDD及作用于相应RDD上的各种操作；
- 阶段：是作业的基本调度单位，一个作业会分为多组任务，每组任务被称为“阶段”，或者也被称为“任务集”。

架构设计

Spark运行架构

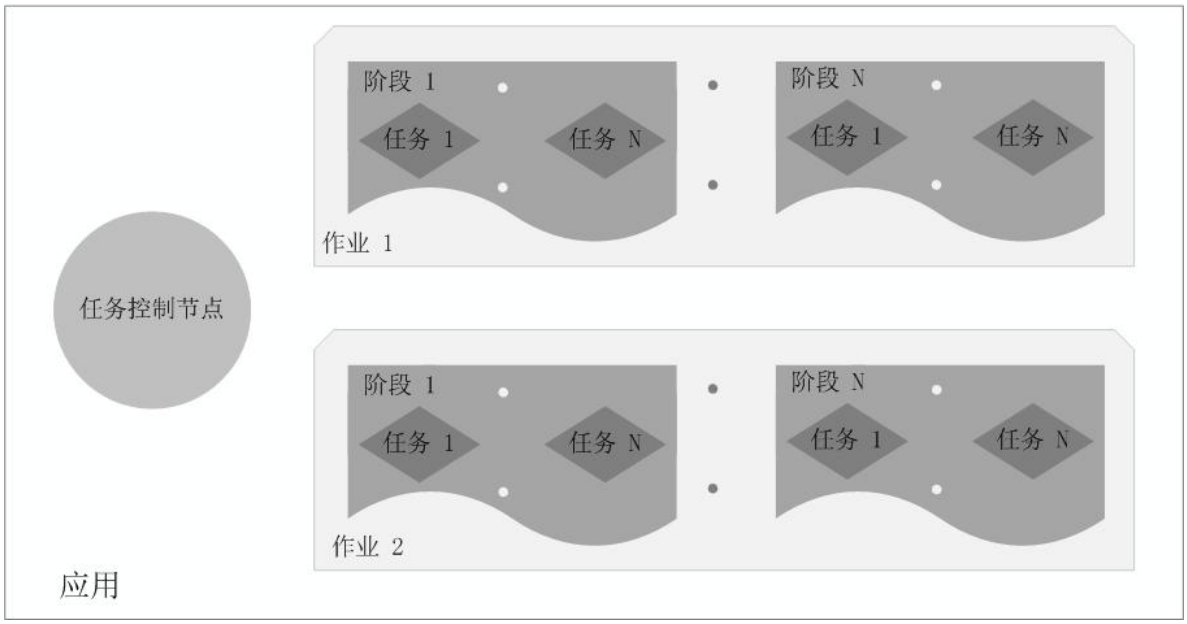
Spark运行架构包括集群资源管理器（Cluster Manager）、运行作业任务的工作节点（Worker Node）、每个应用的任务控制节点（Driver）和每个工作节点上负责具体任务的执行进程（Executor）。其中，集群资源管理器可以是Spark自带的资源管理器，也可以是YARN或Mesos等资源管理框架。



与Hadoop MapReduce计算框架相比，Spark所采用的Executor有两个优点：

1. 利用多线程来执行具体的任务（Hadoop MapReduce采用的是进程模型），减少任务的启动开销；
2. Executor中有一个BlockManager存储模块，会将内存和磁盘共同作为存储设备，当需要多轮迭代计算时，可以将中间结果存储到这个存储模块里，下次需要时，就可以直接读该存储模块里的数据，而不需要读写到HDFS等文件系统里，因而有效减少了IO开销；或者在交互式查询场景下，预先将表缓存到该存储系统上，从而可以提高读写IO性能。

Spark中各种概念之间的相互关系



在Spark中

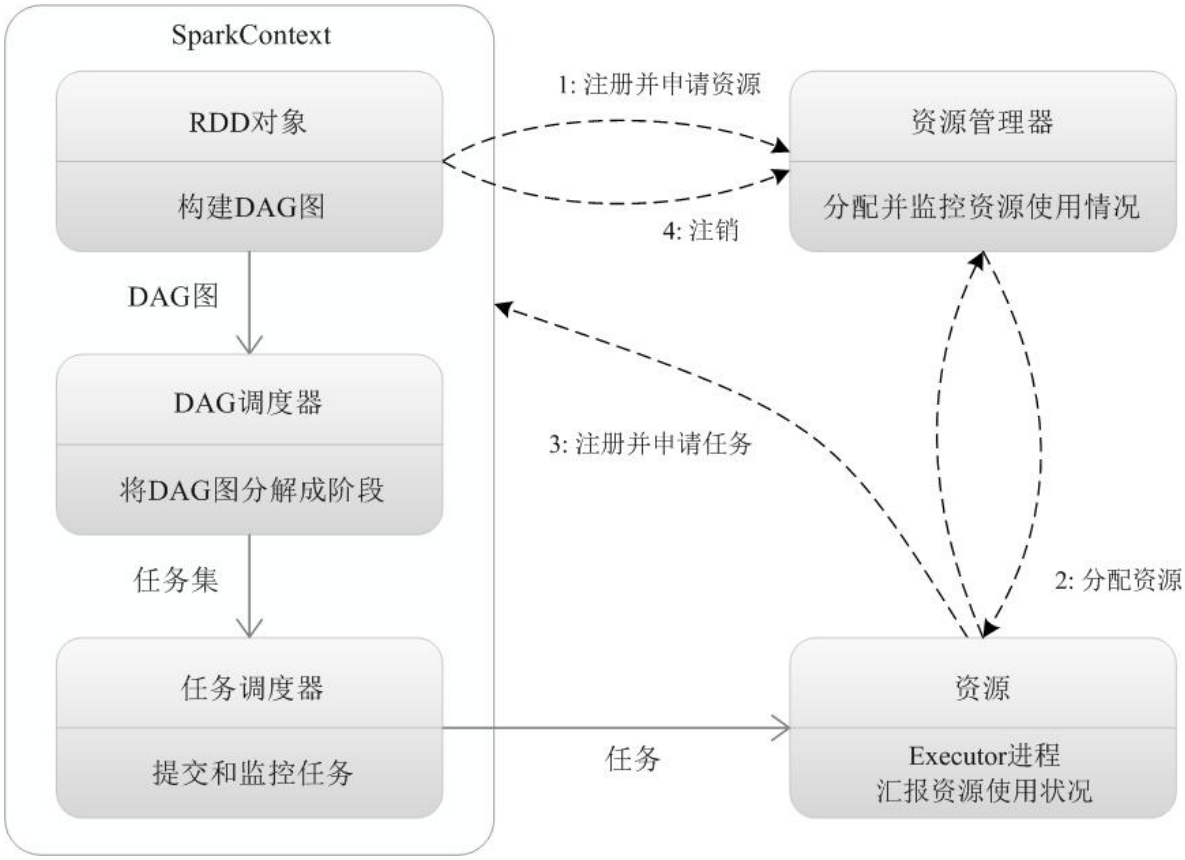
一个应用（Application）由一个任务控制节点（Driver）和若干个作业（Job）构成

一个作业由多个阶段（Stage）构成，一个阶段由多个任务（Task）组成

当执行一个应用时，任务控制节点会向集群管理器（Cluster Manager）申请资源，启动Executor，并向Executor发送应用程序代码和文件，然后在Executor上执行任务

运行结束后，执行结果会返回给任务控制节点，或者写到HDFS或者其他数据库中。

spark运行基本流程



1. 当一个Spark应用被提交时，首先需要为这个应用构建起基本的运行环境，即由任务控制节点（Driver）创建一个SparkContext，由SparkContext负责和资源管理器（Cluster Manager）的通信以及进行资源的申请、任务的分配和监控等。SparkContext会向资源管理器注册并申请运行Executor的资源；
2. 资源管理器为Executor分配资源，并启动Executor进程，Executor运行情况将随着“心跳”发送到资源管理器上；
3. SparkContext根据RDD的依赖关系构建DAG图，DAG图提交给DAG调度器（DAGScheduler）进行解析，将DAG图分解成多个“阶段”（每个阶段都是一个任务集），并且计算出各个阶段之间的依赖关系，然后把一个个“任务集”提交给底层的任务调度器（TaskScheduler）进行处理；Executor向SparkContext申请任务，任务调度器将任务分发给Executor运行，同时，SparkContext将应用程序代码发放给Executor；
4. 任务在Executor上运行，把执行结果反馈给任务调度器，然后反馈给DAG调度器，运行完毕后写入数据并释放所有资源。

spark运行架构的特点

1. 每个应用都有自己专属的Executor进程，并且该进程在应用运行期间一直驻留。Executor进程以多线程的方式运行任务，减少了多进程任务频繁的启动开销，使得任务执行变得非常高效和可靠；
2. Spark运行过程与资源管理器无关，只要能够获取Executor进程并保持通信即可；
3. Executor上有一个BlockManager存储模块，类似于键值存储系统（把内存和磁盘共同作为存储设备），在处理迭代计算任务时，不需要把中间结果写入到HDFS等文件系统，而是直接放在这个存储系统上，后续有需要时就可以直接读取；在交互式查询场景下，也可以把表提前缓存到这个存储系统上，提高读写IO性能；
4. 任务采用了数据本地性和推测执行等优化机制。数据本地性是尽量将计算移到数据所在的节点上进行，即“计算向数据靠拢”，因为移动计算比移动数据所占的网络资源要少得多。而且，Spark采用了延时调度机制，可以在更大的程度上实现执行过程优化。比如，拥有数据的节点当前正被其他的任务占用，那么，在这种情况下是否需要将数据移动到其他的空闲节点呢？答案是不一定。因为，如果经过预测发现当前节点结束当前任务的时间要比移动数据的时间还要少，那么，调度就会等待，直到当前节点可用。

RDD的设计与运行原理

Spark的核心是建立在统一的抽象RDD之上，使得Spark的各个组件可以无缝进行集成，在同一个应用程序中完成大数据计算任务。RDD的设计理念源自AMP实验室发表的论文《Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing》。

RDD设计背景

在实际应用中，存在许多迭代式算法（比如机器学习、图算法等）和交互式数据挖掘工具，这些应用场景的共同之处是，不同计算阶段之间会重用中间结果，即一个阶段的输出结果会作为下一个阶段的输入。

RDD提供了一个抽象的数据架构，我们不必担心底层数据的分布式特性，只需将具体的应用逻辑表达为一系列转换处理，不同RDD之间的转换操作形成依赖关系，可以实现管道化，从而避免了中间结果的存储，大大降低了数据复制、磁盘IO和序列化开销。

RDD概念

一个**RDD**就是**一个分布式对象集合**，**本质上是一个只读的分区记录集合**，每个**RDD可以分成多个分区**，每个**分区就是一个数据集片段**，并且**一个RDD的不同分区可以被保存到集群中不同的节点上**，从而可以在集群中的不同节点上进行并行计算。

RDD提供了一种高度受限的共享内存模型，即**RDD是只读的记录分区的集合**，不能直接修改，只能基于稳定的物理存储中的数据集来创建RDD，或者通过在其他RDD上执行确定的转换操作（如map、join和groupBy）而创建得到新的RDD（也就是RDD不能修改，只能新建）

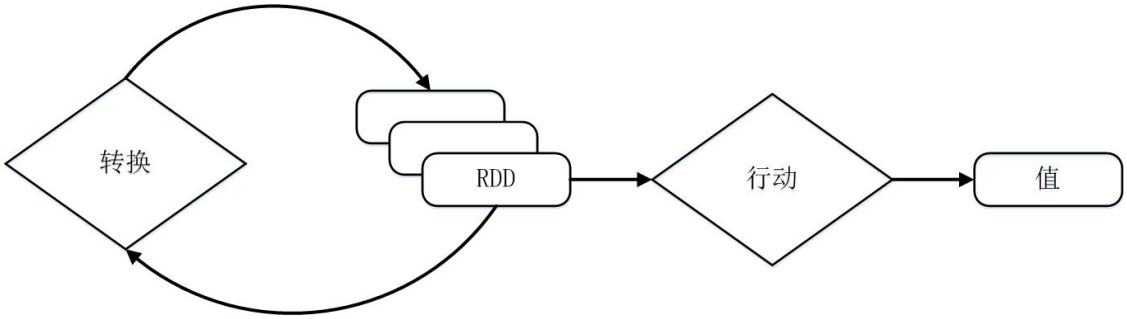
RDD操作

- 行动：执行计算并指定输出的形式，(count,collect,etc)接受RDD返回非RDD(即输出一个值或结果)
- 转换：指定RDD之间的相互依赖关系，(map,filter,groupBy,join,etc)接受RDD并返回新的RDD，**RDD提供的转换接口都是比较简单的粗粒度数据转换操作，因此RDD不适合针对某个数据项的细粒度修改**

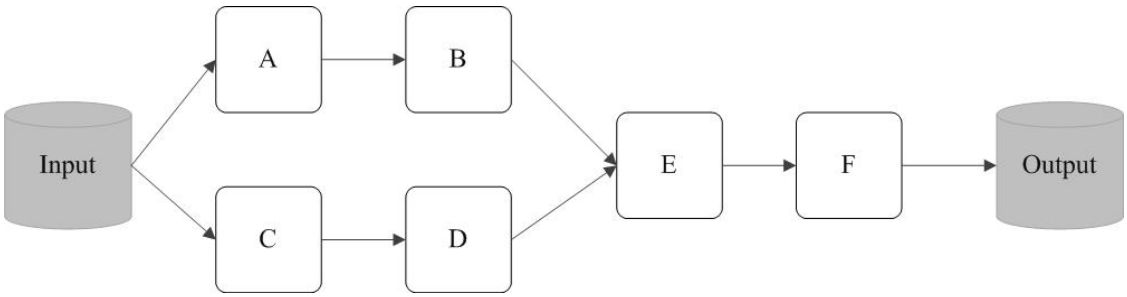
RDD的执行过程

1. RDD读入外部数据源（或内存中的集合）进行创建
2. RDD经过一系列转换操作，每次都产生不同的RDD，供下一个转换使用
3. 最后一个RDD经**行动**操作进行处理，并输出到外部数据源

   RDD采用了惰性调用，即RDD的执行过程中，真正的计算发生在RDD的**行动**操作，对于行动之前的所有转换操作，spark只是记录下转换操作应用的一些基础数据集以及RDD生成的轨迹，即相互之间的依赖关系，而不会出发真正的计算



执行过程举例



从输入中逻辑上生成A和C两个RDD，经过一系列“转换”操作，逻辑上生成了F（也是一个RDD），之所以说是逻辑上，是因为这时候计算并没有发生，Spark只是记录了RDD之间的生成和依赖关系。当F要进行输出时，也就是当F进行“行动”操作的时候，Spark才会根据RDD的依赖关系生成DAG，并从起点开始真正的计算。

上述这一系列处理称为一个“**血缘关系（Lineage）**”，即DAG拓扑排序的结果。采用惰性调用，**通过血缘关系连接起来的一系列RDD操作就可以实现管道化（pipeline）**，避免了多次转换操作之间数据同步的等待，而且不用担心有过多的中间数据，因为这些具有血缘关系的操作都管道化了，一个操作得到的结果不需要保存为中间数据，而是直接管道式地流入到下一个操作进行处理。同时，这种通过血缘关系把一系列操作进行管道化连接的设计方式，也使得管道中每次操作的计算变得相对简单，保证了每个操作在处理逻辑上的单一性

RDD特性

总体而言，Spark采用RDD以后能够实现高效计算的主要原因如下：

1. 高效的容错性。现有的分布式共享内存、键值存储、内存数据库等，为了实现容错，必须在集群节点之间进行数据复制或者记录日志，也就是在节点之间会发生大量的数据传输，这对于数据密集型应用而言会带来很大的开销。在RDD的设计中，数据只读，不可修改，如果需要修改数据，必须从父RDD转换到子RDD，由此在不同RDD之间建立了血缘关系。所以，RDD是一种天生具有容错机制的特殊集合，不需要通过数据冗余的方式（比如检查点）实现容错，而只需通过RDD父子依赖（血缘）关系重新计算得到丢失的分区来实现容错，无需回滚整个系统，这样就避免了数据复制的高开销，而且重算过程可以在不同节点之间并行进行，实现了高效的容错。此外，RDD提供的转换操作都是一些粗粒度的操作（比如map、filter和join），RDD依赖关系只需要记录这种粗粒度的转换操作，而不需要记录具体的数据和各种细粒度操作的日志（比如对哪个数据项进行了修改），这就大大降低了数据密集型应用中的容错开销；
2. 中间结果持久化到内存。数据在内存中的多个RDD操作之间进行传递，不需要“落地”到磁盘上，避免了不必要的读写磁盘开销；
3. 存放的数据可以是Java对象，避免了不必要的对象序列化和反序列化开销。

RDD之间的依赖关系

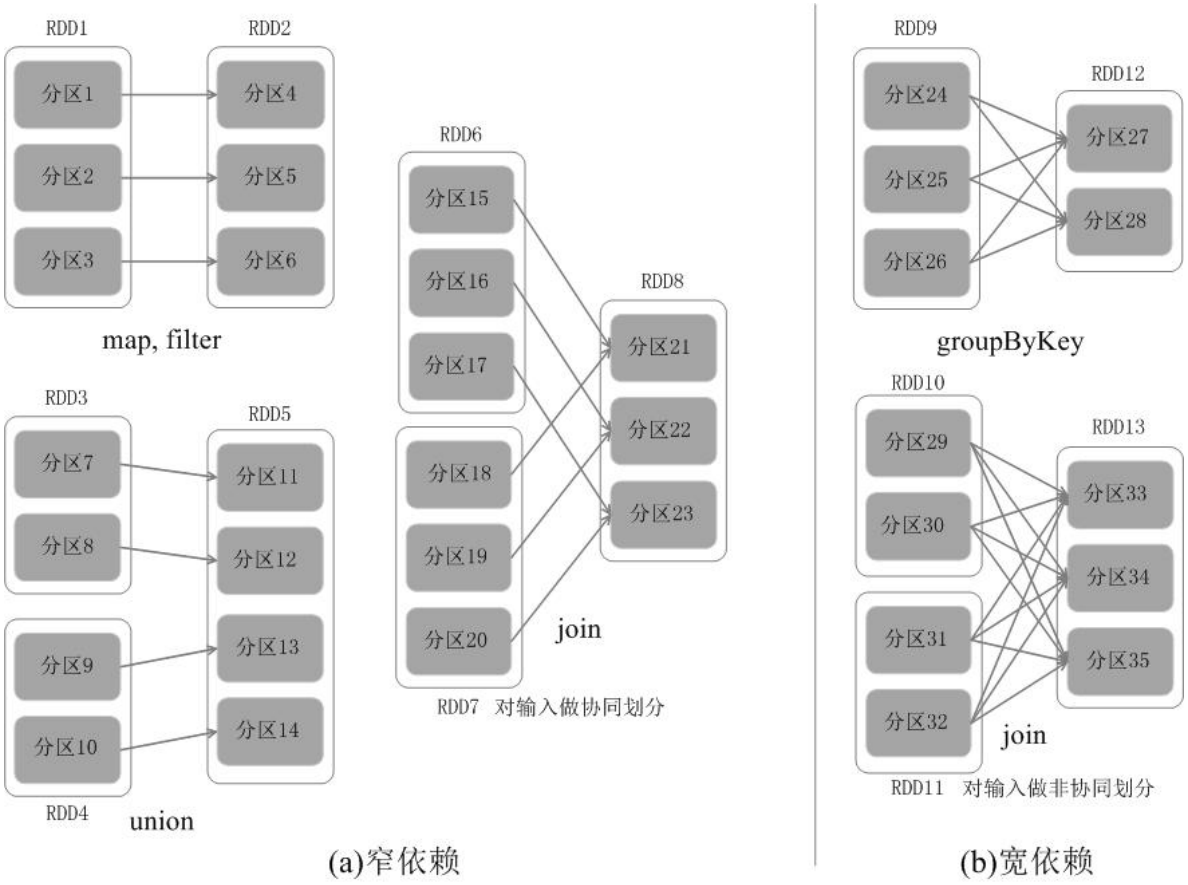
RDD中不同的操作会使得不同RDD中的分区会产生不同的依赖。RDD中的依赖关系分为窄依赖（Narrow Dependency）与宽依赖（Wide Dependency）

- 窄依赖：表现为一个父RDD的分区对应于一个子RDD的分区，或多个父RDD的分区对应于一个子RDD的分区，如果父RDD的一个分区只被一个子RDD的一个分区所使用就是窄依赖(map,filter,union,etc)
- 宽依赖：表现为存在一个父RDD的一个分区对应一个子RDD的多个分区，不符合窄依赖的就是宽依赖(groupByKey,sortByKey,etc)

对于join分为两种情况

1. 对输入进行协同划分属于窄依赖。协同划分指多个父RDD的某一分区的所有键，落在子RDD的一个分区内，不会产生同一个父RDD的某一分区，落在子RDD的两个分区的情况
2. 对输入做非协同划分属于宽依赖。

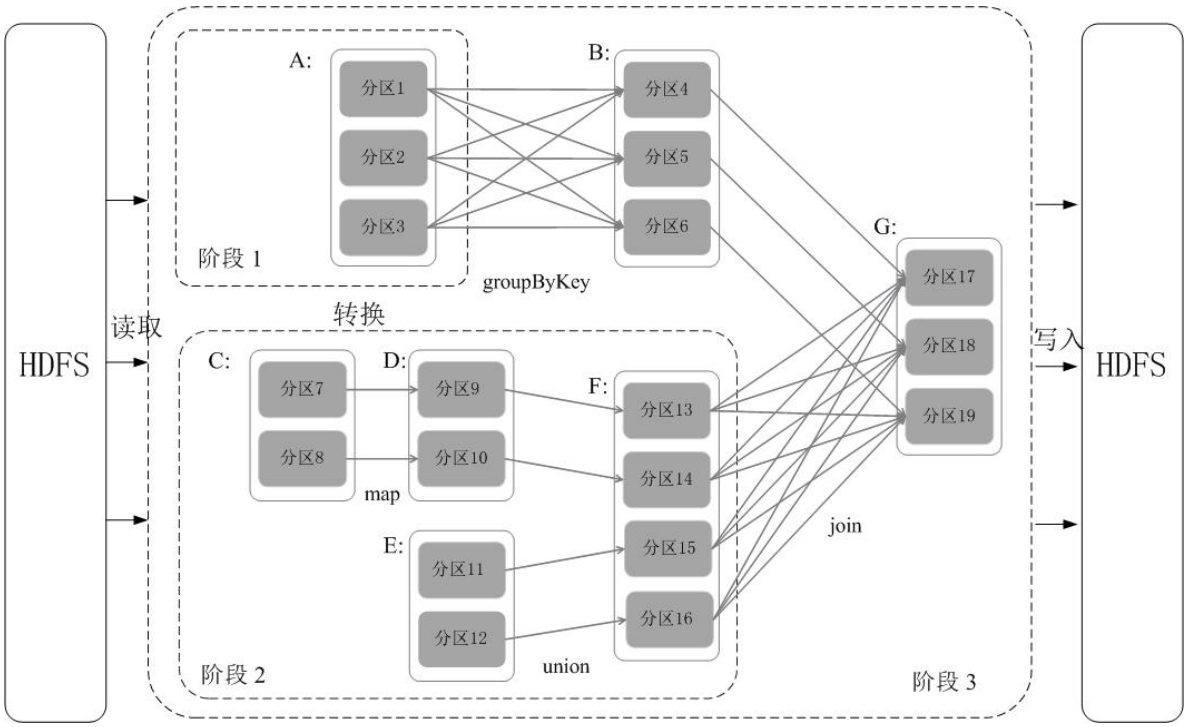
对于窄依赖的RDD，可以以流水线的方式计算所有父分区，不会造成网络之间的数据混合，对于宽依赖的RDD，通常进行shuffle操作，即首先需要计算好所有父分区数据然后在节点之间进行shuffle



阶段的划分

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

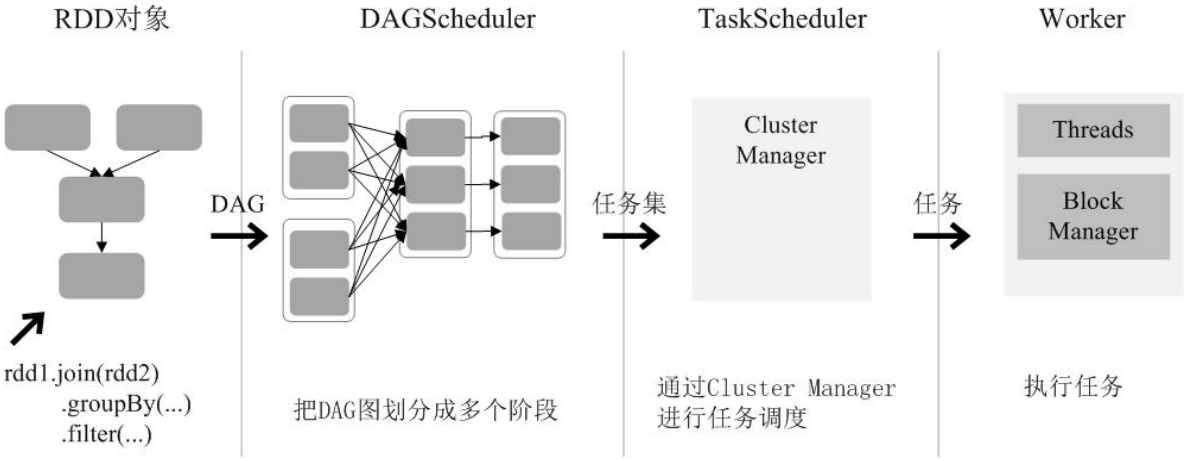
具体划分方法是：在DAG中进行反向解析，遇到宽依赖就断开，遇到窄依赖就把当前的RDD加入到当前的阶段中；将窄依赖尽量划分在同一个阶段中，可以实现流水线计算



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。

RDD运行过程

1. 创建RDD对象
2. sparkcontext负责计算RDD之间的依赖关系，构建DAG
3. DAG Scheduler负责把DAG图分解成多个阶段，每个阶段中包含了多个任务，每个任务会被任务调度器分发给各个工作节点去执行



spark的部署模式

Spark应用程序在集群上部署运行时，可以由不同的组件为其提供资源管理调度服务（资源包括CPU、内存等）。比如，可以使用自带的独立集群管理器（standalone），或者使用YARN，也可以使用Mesos。因此，Spark包括三种不同类型的集群部署方式，包括standalone、Spark on Mesos和Spark on YARN。

standalone模式

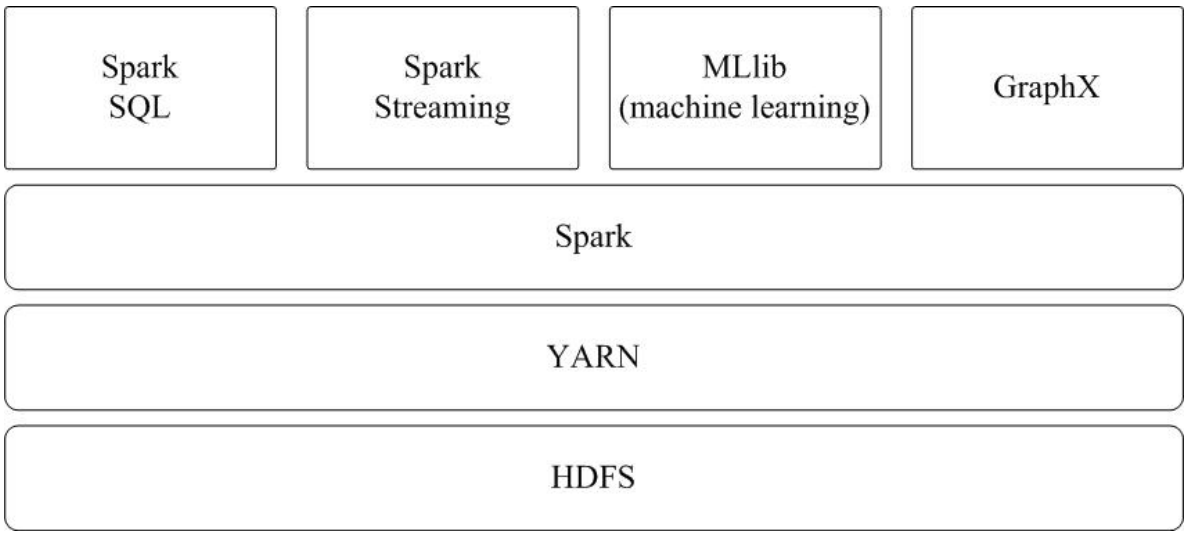
Spark框架本身也自带了完整的资源调度管理服务，可以独立部署到一个集群中，而不需要依赖其他系统来为其提供资源管理调度服务。在架构的设计上，Spark与MapReduce1.0完全一致，都是由一个Master和若干个Slave构成，并且以槽（slot）作为资源分配单位。不同的是，Spark中的槽不再像MapReduce1.0那样分为Map 槽和Reduce槽，而是只设计了统一的一种槽提供给各种任务来使用。

Spark on Mesos模式

Mesos是一种资源调度管理框架，可以为运行在它上面的Spark提供服务。Spark on Mesos模式中，Spark程序所需要的各种资源，都由Mesos负责调度。由于Mesos和Spark存在一定的血缘关系，因此，Spark这个框架在进行设计开发的时候，就充分考虑到了对Mesos的充分支持，因此，相对而言，Spark运行在Mesos上，要比运行在YARN上更加灵活、自然。目前，Spark官方推荐采用这种模式，所以，许多公司在实际应用中也采用该模式。

Spark on YARN模式

Spark可运行于YARN之上，与Hadoop进行统一部署，即“Spark on YARN”，其架构如图所示，资源管理和调度依赖YARN，分布式存储则依赖HDFS。



在jupyter notebook上实现wordCount

```
# 共100个词，随机生成空格和换行符
file_path = work_path + "word_count.txt"
with open(file_path, 'w') as f:
    for i in range(100):
        word_len = random.randint(3, 10)
        tword = ""
        for j in range(word_len):
            tword += "a"
        f.write(tword)
        if random.randint(0, 10) >=5:
            f.write(" ")
        else:
            f.write("\n")
```

```
import os
from pyspark import SparkContext
from pyspark.sql import SparkSession

os.environ["PYSPARK_PYTHON"]="python3"
os.environ["PYSPARK_DRIVER_PYTHON"]="python3"
os.environ['SPARK_HOME'] = "/home/mayfly/workstation/env/spark"
```

```
file_path = "file://" + file_path
```

```
# 创建sparkcontext对象
sc = SparkContext(appName="word_count", master="local")
# 加载文件，文件按行读取所以直接统计出有多少行
mrdd = sc.textFile(file_path)
print("file line num:{}".format(mrdd.count()))
```

```
file line num:54
```

```
# 执行flatMap将每一行切分的结果混合在一起
all_word = mrdd.flatMap(lambda x:x.split(" "))
print("word num:{}".format(all_word.count()))
```

```
word num:100
```

```
# 以word为key进行map，然后将出现的次数进行累加
word_count = all_word.map(lambda x:(x, 1)).reduceByKey(lambda x,y:x+y)
print(word_count.collect())
```

```
[('aaaaaaaa', 10), ('aaa', 18), ('aaaaa', 21), ('aaaaaa', 10), ('aaaaaaa', 11), ('aaaaaaaaa', 9), ('aaaa', 13), ('aaaaaaaaaa', 8)]
```

问题回答

spark 和 mapreduce有哪些区别，请用具体的例子说明？

1. Spark可以基于内存处理数据,Job中间输出结果可以保存在内存中，从而不再需要读写HDFS。但是mapreduce的每一步操作都会存储到内存中，所以IO较大
2. Spark基于DAG的任务调度执行机制，要优于MapReduce的迭代执行机制。DAG带来的好处巨大，DAG划分阶段后对于窄依赖问题可以依赖于流水线加速运算。
3. Spark存储数据可以指定副本个数，MR默认3个。

4. Spark中提供了各种场景 的算子，MR中只有map ,reduce 相当于Spark中的map和reduceByKey 两个算子。
5. Spark 是粗粒度资源申请，Application执行快。
6. Spark 中shuffle map端自动聚合功能，MR手动设置。
7. MapReduce采用了多进程模型，而Spark采用了多线程模型

rdd的本质是什么？

一个**RDD**就是**一个分布式对象集合**，**本质上是一个只读的分区记录集合**，每个**RDD可以分成多个分区**，每个**分区就是一个数据集片段**，并且**一个RDD的不同分区可以被保存到集群中不同的节点上**，从而可以在集群中的不同节点上进行并行计算。