> Spark 是基于内存计算的大数据并行计算框架，可用于构建大型的、低延迟的数据分析应用程序。Spark 是一个通用的计算引擎，专门为大规模数据处理而设计，与MapReduce类似，不同的是，MapReduce把中间结果写入 HDFS，而Spark直接写入内存，这使得它能够实现实时计算，速度比MapReduce快100倍。

Spark 基础

Spark 安装

安装Spark

将Spark上传至新建的目录 `/opt/software/` 目录下，并解压至 `/usr/local/src/` 目录下。

配置环境变量

全局生效： `/etc/profile.d/Spark.sh`

只针对root账户生效： `/root/.bash_profile`

```
export SPARK_HOME=/usr/local/src/hive
export PATH=$PATH:$SPARK_HOME/bin:$SPARK_HOME/sbin
```

配置完成，重启系统或使用source命令重新加载文件使其生效：

```
source /etc/profile
或
source /root/.bash_profile
```

Spark 配置

修改配置文件

Spark 配置目录：

`$SPARK_HOME/conf/`

1. 修改环境变量

复制 `spark-env.sh.template` 并改名为 `spark-env.sh`

```
cp spark-env.sh.template spark-env.sh
```

配置 `spark-env.sh` 环境变量：

```
export HADOOP_CONF_DIR=/opt/hadoop/etc/hadoop
export SPARK_CONF_DIR=/usr/local/src/spark/conf
export JAVA_HOME=[JDK安装目录]
export SPARK_MASTER_HOST=[主机名(master)]
export SPARK_MASTER_PORT=7077
```

2. 配置集群

复制 `slaves.template` 并改名为 `slaves`

```
cp slaves.template slaves
```

修改 `slaves` 添加集群：

```
master
slave1
slave2
```

拷贝Mysql驱动包

将JDBC驱动包拷贝至 `$SPARK_HOME/jars/` 下。

拷贝Hive配置至Spark

将Hive的配置文件 `hive-site.xml` 拷贝至 Spark的 `/conf` 目录下。

拷贝至其他集群

将安装好的Spark与环境变量拷贝至其他的集群。

```
scp -r /usr/local/src/spark slave1:/usr/local/src
scp -r /usr/local/src/spark slave2:/usr/local/src
scp -r /root/.bash_profile slave1:/root/.bash_profile
scp -r /root/.bash_profile slave2:/root/.bash_profile
```

并在其他集群使用 `source` 重新加载集群的环境变量 `.bash_profile` 。

启动 Spark

（为了避免与 Hadoop 的全局 start-all.sh 冲突）

进入到 `spark/sbin` 目录下，输入 `./start-all.sh` 启动 Spark。

启动成功后，在master主机上会显示进程

- Master
- Worker

在 slave 主机上会显示进程

- Worker

进入WebUI

安装配置完毕后，在实体机的浏览器地址栏输入 `虚拟机IP:8080` 即可进入 Spark 的WebUI。

Spark 任务提交

运行Spark内置案例

运行Spark内置案例 **PI** 进行测试。

使用 **Spark集群** 方式运行spark内置案例（确保Spark集群是启动的）：

```
spark-submit --class org.apache.spark.examples.SparkPi --master spark://master:7077 --executor-memory 1g --total-executor-cores 1 /usr/local/src/spark/examples/jars/spark-examples_2.11-2.1.1.jar 100
```

使用 **Spark-on-yarn** 方式运行spark内置案例（确保Hadoop集群是启动的）：

```
spark-submit --class org.apache.spark.examples.SparkPi --master yarn --executor-memory 1g --total-executor-cores 1 /usr/local/src/spark/examples/jars/spark-examples_2.11-2.1.1.jar 100
```

运行jar打包程序

使用 **Spark集群** 方式运行spark内置案例（确保Spark集群是启动的）：

```
spark-submit --class [类名] --master spark://master:7077 --executor-memory 1g --total-executor-cores 1 [jar所在目录] [main方法参数（没有可不填）]
```

使用 **Spark-on-yarn** 方式运行spark内置案例（确保Hadoop集群是启动的）：

```
spark-submit --class [类名] --master yarn --executor-memory 1g --total-executor-cores 1 [jar所在目录] [main方法参数（没有可不填）]
```

指定yarn运行模式为cluster模式（默认为client模式）

```
--deploy-mode cluster
或者
--master yarn-cluster
```

推荐参数

```
--num-executors 3
--executor-cores 3
--executor-memory 6g
```

Spark 报错案例

若在执行 jar 时，出现 `SessionHiveMetaStoreClient` 关键词错误，需要关闭 Hive 的版本检测。

在 Hive 的配置文件 hive-site.xml 中，找到以下配置，并将值设置为 `false` 。

```
<name>hive.metastore.schema.verification</name>
<value>true</value> (需要更改为false)
```

修改完成后，要记得将 hive-site 重新覆盖到 Spark/conf 下以及其他集群的 Spark/conf 下。

Spark core

RDD 编程模型

创建RDD

准备前置环境：

```
// 创建SparkConf和SparkContext对象
val conf = new SparkConf.setAppName("类名").setMaster("local")
val sc = new SparkContext(conf)

// 业务处理代码

// 关闭Spark连接，释放资源
sc.stop()
```

从集合（内存）中创建RDD：

```
val rdd = sc.parallelize(Array(1,2,3,4,5,6))
```

从外部文件创建RDD：

```
val rdd = sc.textFile(文件目录)
```

方法 —— 算子

RDD的方法（算子）分为转换算子与行动算子。

只有在调用行动算子时，转换算子才会执行。

- 转换算子：功能的补充和封装，将旧的RDD包装成新的RDD
- 行动算子：触发任务的调度和作业的执行

转换算子

`map(func(x))`

将待处理的数据**逐条**进行处理，并生成返回一个新的RDD。

`filter(func(x))`

将数据进行过滤，结果为true的保留，结果为false的进行过滤。

`gruopBy(func(x))`

gruopBy会将数据源中的每一个数据根据函数的返回值作为key进行分组判断，相同的返回值的数据会被分为一组。

`gruopByKey()`

gruopBykey会将数据源中的数据，相同Key的数据分在一个组中，形成并返回一个对偶元组。（适用于Key-Value类型的数据）

元组中的第一个元素就是Key，

元组中的第二个元素就是相同Key的value的**集合**。

`reduceByKey(fanc(a,b))`

相同的Key的数据进行value数据的聚合操作。

slaca的聚合操作原理是两两聚合，因此该函数内参数应该是两个。

行动算子

`collect()`

collect可以将RDD转换为Scala中的数组。

`count()`

获取数据源中数据的个数

累加器和广播变量

累加器

实现原理

累加器（Accumulator）用来把 Executor 端变量信息聚合到 Driver 端。在 Driver 程序中定义的变量，在 Executor 端的每个 Task 都会得到这个变量的一份新的副本，每个 task 更新这些副本的值后，传回 Driver 端进行合并。

创建累加器

Spark 提供三种系统累加器，分别是 Long 类型、Double 类型、Collecttion 类型（底层是 Java.Unit.List）。

```
val sumAcc = sc.longAccumulator("acc1");
```

累加器的注意事项

少加：转换算子中调用累加器，如果没有行动算子的话，那么不会执行

多加：转换算子中调用累加器，如果多次调用行动算子，会执行多次

统计条数使用方法：

```
// 在filter内，对累加器进行add(1)
// 在最后，将累加器重置
counter1.reset()
// 然后对目标df运行一遍count方法
df2.count()
// 再使用变量接收value
val count2 = counter1.value
```

累加器的方法

累加器增加值

`acc.add(1)`

获取累加器的值

`acc.value`

广播变量

实现原理

广播变量用来高效分发较大的对象。向所有工作节点发送一个较大的只读值，以供一个或多个Spark操作使用。比如，如果你的应用需要向所有节点发送一个较大的只读查询表广播变量用起来都很顺手。在多个并行操作中使用同一个变量，但是Spark会为每个任务分别发送。

阶段二三

数据抽取

全量抽取

代码：

```
// 构建SparkSession对象
val sparkSession =
SparkSession.builder().appName("Test").master("local").getOrCreate()

// 构建数据库访问属性配置对象
val url = "jdbc:mysql://localhost:3306/sthd_student?useSSL=false"
val prop = new Properties()
prop.setProperty("user","root")
prop.setProperty("password",123456)

// 查询数据得到DataFrame对象
val df1 = sparkSession.read.jdbc(url,"tb_class",prop)

//如果需要条件查询，使用时需要用将语句用()装起来，并且用 as 给表设置别名
val df2 = sparkSession.read.jdbc(url,"(select * from 表名 where 字段 = 值) as tb1",prop)

// 查看DataFrame对象中的数据，默认显示20条
df.show()

// 关闭spark对象，释放资源
sparkSession.stop()
```

增量抽取

增量即从mysql中排除掉hive的ods对应表中已存在的数据，方法如下：

- 如果字段是按照顺序排序，直接从hive表中查看最大id值或最大日期，从mysql中获取数据的时候直接过滤掉（使用where）
- 查询出hive中对应表的数据，然后在mysql抽取的数据中使用filter过滤掉这些数据

1. 获取Hive的指定字段数据集合，用于进行对比

```
// 将dataFrame转换为rdd，再使用map()生成一个新的rdd，再使用collect()将rdd转换为数组
val id_arr = hiveDF.rdd.map((x) => {
    x.getInt(0)
}).collect()
```

2. 使用 `filter(func)` 对mysqlD进行过滤，获得经过过滤的新dataFrame

```
val dataFrame2 = mysqlDF.filter((x) => {
    val id = x.getInt(0)
    if (id_arr.contains(id)){
        false
    }
    else{
        true
    }
})
```

数据插入

插入静态分区

1. sparkSession对象开启对Hive的支持：

   在创建SparkSession对象时，在 .master() 方法后面添加 `.enableHiveSupport().`

```
val sparkSession =
SparkSession.builder().appName("test").master("local").enableHiveSupport().getOrCreate
()
```

2. 注册临时表

   在DataFrame对象获取到数据后，调用 `.createOrReplaceTempView("临时表名")` 将DateFrame注册为一张临时表。

3. 执行sql语句，选择到Hive中对应的数据库

```
// 在程序中执行sql语句禁止在末尾加分号
sparkSesseion.sql("use ods")
```

4. 执行sql语句，插入临时表中的数据至Hive中对应的表中，并且添加静态分区值

```
sparkSesseion.sql("insert into [表名] partition(分区字段名 = '静态分区值') select * from 临
时表名")
```

插入动态分区

在插入数据时，将DataFrame的分区列（最后一列）同步插入到hive表的动态分区中。

也就是在 `partition()` 的括号中只需要填写hive中的分区列名，所选插入表单中的最后一列将会自动插入到hive的动态分区列中。

在动态分区插入数据前，需要先执行hive中的set命令用于开启动态分区：

```
sparkSession.sql("set hive.exec.dynamic.partition=true")
sparkSession.sql("set hive.exec.dynamic.partition.mode=nonstrict")
```

插入数据：

```
sparkSession.sql("insert into tb_student partition(动态分区列名) select * from
mystudent")
```

打包运行

1. 右键项目--》open module settings-》左侧Artifacts-》点击+号-》Jar-From module with dependencies-》删除右侧依赖只保留output
2. idea菜单中：Build-》Builde Artifacts-》Build然后在项目下产生一个out目录
3. 将jar包上传至Linux中，使用Spark执行。

其他操作

从文件加载数据

```
// 加载为RDD
val rdd = saprkContext.textFile("path")

// 加载为dataFrame
val dataFrame = sparkSession.read.csv("path")

// 加载为dataFrame并使用.option("key","value")添加参数
val dataFrame = sparkSession.read.option("header",true).csv("path")
// option("header",true) 将使用第一行数据作为列名
// option("delimiter","分隔符") 自定义数据分隔符，如不设置默认为逗号
```

RDD转换为dataFrame

使用toDF之前，需要导入隐式转换包：

```
import sparkSession.implicits._
```

```
val df = rdd.toDF("字段名1","字段名2")
```

写入到数据库

```
dataFrame.write.jdbc("url","table",properties)
```

数据清洗

1. 对ods从层表数据进行清洗和处理，将处理后的数据存入dwd层中。

常见要求：

- 去重
- 异常值处理
- 缺失值的处理
- 格式的规范
- 计算新增额外列
- ……

2. 针对dwd层数据进行指标计算（RDD计算、spark-sql计算）

去重

完全匹配去重

当多条数据的内容完全一致才会进行去重，保留一条数据。

`rdd.distinct()` 对rdd进行去重，返回一个新的rdd。

`dataFrame.distinct()` 对dataFrame进行去重，返回一个新的dataFrame。

根据字段的内容去重

当多条数据的某一列数据相同时会进行去重，保留一条数据。

`dataFrame.dropDuplicates("列名1,列名2")` 返回一个新的dataFrame。

获取去重的条数

使用去重前的rdd的count减去去重后的count，可以得到去重的条数。

数据过滤

数据过滤

`.filter(func(x))`

将数据进行过滤，结果为true的保留，结果为false的进行过滤。返回一个新的DF。

获取过滤的条数

生成计数器：

```
// 创建sparkConText对象
val sc = sparkSession.sparkContext
// 通过sparkConText生成计数器
val counter1 = sc.longAccumulator("counter1")
```

计数器数值增加：

```
counter1.add(1)
```

获取计数器的值：

```
counter1.value
```

缺失值填充处理

RDD方式处理：

- **要求：将第一列为空的填充为“匿名”**

```
val rdd = sc.textFile("data/student.txt")
val rdd2 = rdd.map((x) => {
    val arr = x.split(",")
    if (arr(0).equals("")){
        "匿名"+","+arr(1)+","+arr(2)
    }
    else {
        arr(0)+","+arr(1)+","+arr(2)
    }
})
```

- **要求：将所有缺失年龄的数据填充为年龄均值（使用collect方式）**

步骤1：计算出年龄平均值。

```
// 将没有年龄的数据进行过滤，并算出平均值
val age_arr = rdd.filter((x) => {
    val arr = x.split(",")

    // 当最后一个逗号后面没有内容时，空字符串将不会作为数组元素被切割，因此这里使用
arr(3).equels("")会造成数组越界，所以表达式改为使用 arr.length == 3
    if(arr.length == 3){
        false
    }
    else {
        true
    }
    // 提取出所有Age汇总为数组
}).map((x) => {
    val arr = x.split(",")
    // 将String类型转换为Int类型以便于计算
    arr(3).toInt
}).collect()

// 使用Age数组算出平均值，保存至变量中
val age_avg = age_arr.sum / age_arr.length
```

步骤2：填充年龄平均值，

```
val rdd2 = rdd.map((x) => {
    val arr = x.split(",")
    if (arr.length == 3) {
        // 如果最后一列没有数据，将平均值填充至最后一列
        arr(0)+","+arr(1)+","+arr(2)+","+age_avg
    }
    else {
        // 如果最后一列有数据，数据原封不动返回
        arr(0)+","+arr(1)+","+arr(2)+","+arr(3)
    }
})
```

- **要求：将所有缺失年龄的数据按照性别进行分组，根据性别填充为对应的年龄均值（使用groupByKey方式）**

步骤1：计算出男/女各自的年龄平均值。

```
// 首先对数据进行过滤，年龄为空的数据不参与计算
val age_avg_map = rdd.filter((x) => {
    val arr = x.split(",")
    if (arr.length == 3) {
        false
    }
    else {
        true
    }
    // 将每条数据中的性别和年龄提取出来，每条数据形成对偶元组
}).map((x) => {
    val arr = x.split(",")
    val sex = arr(2)
    val age = arr(3)
    (sex, age.toInt)
    // 将对偶元组使用groupByKey()进行汇总
}).groupByKey().map((x) => {
    // 汇总完成后将每组数据进行平均值计算
    val sex = x._1
    val age_avg = x._2.sum / x._2.size
    (sex, age_avg)
    // 将得到的二元组数据转换为Map，以便接下来使用
}).collectAsMap()
```

步骤2：根据性别分组填充年龄平均值。

```
val rdd2 = rdd.map((x) => {
    val arr = x.split(",")
    if (arr.length == 3){
        val sex = arr(2)
        // 如果最后一列没有数据，使用avg_map中对应的key取出value
        arr(0) + "," + arr(1) + "," + arr(2) + "," + age_avg_map(sex)
    }
    else {
        // 如果最后一列有数据，数据原封不动返回
        arr(0) + "," + arr(1) + "," + arr(2) + "," + arr(3)
    }
})
```

DataFrame方式处理：

- **要求：将第一列为NULL的填充为“匿名”**

注意：在进行空值判断的时候，检查数据源中的空值是为Null还是空字符串，如果是空字符串则使用.euqels("")进行判断。

```
df = df.withColumn("name",
    // 如果字段为Null，则填充为“匿名” 如果字段不为Null，则仍然填充原来数据
    when(col("name").isNull,"匿名").otherwise(col("name")))
```

- **要求：将所有缺失年龄的数据填充为年龄均值**

实现方式1：把df注册为一张临时表，使用SQL语句得到平均值。

```
df.createOrReplaceTempView("student")
val df2 = sparkSession.sql("SELECT AVG(age) FROM `student`")
val age_avg = df2.collect()(0)(0)
```

实现方式2：使用avg(列名)函数获取一列的平均值（NULL自动不参与计算）。

```
df.withColumn("age",when(col("age").isNull,avg("age").over(Window.partitionBy("lit("aa
a")"))).otherwise(col("age")))
// 由于avg()函数必须在分组情况下使用，所以使用over()开窗进行分组,由于没有"aaa"字段所以所有数据都会被归纳为一组
```

- **要求：将所有缺失年龄的数据按照性别进行分组，根据性别填充为对应的年龄均值**

实现方式1：把df注册为一张临时表，使用SQL语句得到每个性别对应的平均值，再使用map进行保存。

```
df.createOrReplaceTempView("student")
val df2 = sparkSession.sql("SELECT sex,FLOOR(AVG(age)) FROM `student` GROUP BY sex")
// 将df转换为rdd，经过处理后再转换为map
val age_avg_map = df2.rdd.map((x) => {
    (x(0),x(1))
}).collectAsMap()

// 使用嵌套when将平均值填充至对应的行
df.withColumn("age",when(col("age").isNull,when(col("sex").equalTo("男"),age_avg_map("
男")).when(col("sex").equalTo("女"),age_avg_map("女"))).otherwise(col("age")))
```

实现方式2：使用avg(列名)函数获取一列的平均值，按照性别进行分组求平均值。

```
df.withColumn("age",when(col("age").isNull,avg("age").over(Window.partitionBy("lit("se
x")"))).otherwise(col("age")))
// 由于avg()函数必须在分组情况下使用，所以使用over()进行分组，分组的条件为sex字段
```

格式规范处理

RDD方式处理：

- **要求：对表的日期进行格式化处理**

步骤1：对字符串类型的日期进行格式转换得到新字符串。

```
// 创建 SimpleDateFormat对象1，用于将原格式字符串解析为Date对象
val sdf = new SimpleDateFormat("yyyy-MM-dd")
// 创建 SimpleDateFormat对象2，用于将Date对象格式化为新格式
val sdf2 = new SimpleDateFormat("yyyy-MM-dd HH:mm:ss")

val old_date = "arr(1)"
// 使用sdf1，将该格式解析为Date对象
val date = sdf.parse(str)
// 使用sdf2，将旧格式的Date对象转换为新格式并返回字符串
val new_date = sdf2.format(date)
```

步骤2：将新字符串在数据中替换。

- **要求：根据指定日期计算出星期几，并作为新增字段保存**

步骤1：通过调用Java中的Calendar类，获取到对应的星期。

```
// 指定日期
val str = "2022-08-19"
// 创建星期数组，从星期日开始
val weekDays = Array("星期日", "星期一", "星期二", "星期三", "星期四", "星期五", "星期六")
// 创建SimpleDateFormat对象，用于将字符串格式的日期转换为Date
val sdf = new SimpleDateFormat("yyyy-MM-dd")
// 创建Calendar(日历类)对象
val cal = Calendar.getInstance()
// 将cal对象的日期设置为指定的日期
cal.setTime(sdf.parse(str))
// 使用Calendar.DAY_OF_WEEK获取到该日期的星期，并且-1得到与数组对应的星期
val w = cal.get(Calendar.DAY_OF_WEEK) -1
// 取出数组元素并保存为变量
val weekDay = weekDays(w)
```

步骤2：将取出的星期作为新的字段返回。

DataFrame方式处理：

- **要求：对表的日期进行格式化处理**

使用 data_format() 函数进行格式化。

```
df = df.withColumn("birthday",date_format(col("birthday"),"yyyy-MM-dd HH:mm:ss"))
```

或者使用 .cast 将date类型的字段转换为timestamp类型。

```
.withColumn("reg_date",col("reg_date").cast("timestamp"))
```

- **要求：根据指定日期计算出星期几，并作为新增字段保存**

使用 data_format() 函数进行格式化。格式为 `u` ，获取到的为对应星期。

```
df = df.withColumn("week",date_format(col("birthday"),"星期u"))
```

指标计算

Dwd层是清洗处理后的规范化数据表，联合该层数据表对业务功能进行分析计算得到结果存入Mysql或Hive表中，常见计算方法有：

- 将查询的DataFrame转RDD进行一系列map、groupby、reduce、sortByi计算结果
- 直接使用Hive-SQL或者Spark-SQL针对DataFrame进行统计得到结果

Hive-SQL方式处理：

- **要求：统计各个学校下的班级最多的前三名：（单表操作）**

```
select shcool,count(*) as class_sum from dim_class
group by school
order by class_sum desc
limit 3
```

- **要求：统计各个学校考试的总次数，平均分，并且按照平均分进行排名（三表操作）**

Hive SQL代码：

```
select school,count(*) as score_sum,avg(score) as avg_score,row_number over(order by
avg_score desc)
from dim_class as c inner join dim_student as s on c.cid = s.scid inner join dim_score
as d on d.sid = s.sid
group by school
```

Scala代码：

由于在程序中执行SQL代码会直接将 `row_nomber() over()` 判断为字段名，因此需要将原数据使用括号包起来并且设置别名，以把这些数据当作一个表作为数据来源，再额外使用row_number()作为字段名。

```
select school,score_sum,avg_score,row_number over(order by avg_score desc) from
(select school,count(*) as score_sum,avg(score) as avg_score
from dim_class as c inner join dim_student as s on c.cid = s.scid inner join dim_score
as d on d.sid = s.sid
group by school) as t1
```

- **要求：统计各个学校下男女人数以及占比，各个学校一行记录的格式**

```
select school,
sum(case when sex=''then 1 else 0 end) as nan,
sum(case when sex=''then 1 else 0 end)/count(*) as nan rate,
sum(case when sex=''then 1 else 0 end) as nv,
sum(case when sex=''then 1 else 0 end)/count(*)as nv rate
from dim_class as c inner join dim_student as s on c.cid=s.scid
group by school
```

- **要求：统计各个学校、各个班级考试的总次数、平均分，并且按照学校内部添加平均分排名（三表操作）**

```
select school,name,pjf,row number() over(partition by school order by pjf desc) as
seq
from (select school,c.name,count(*),avg(score) as pjf
from dim_class as c inner join dim_student as s on c.cid=s.scid inner join dim_score
as ts on ts.sid=s.sid
group by school,c.name) as t1
```

RDD方式处理：

- **要求：统计各个学校下的班级最多的前三名：（单表操作）**

```
val rdd = sparkSession.sql("select * from dim_class").rdd
val top3_arr = rdd.map((x) => {
    val school = x(3)
    // 返回对偶元组，方便以学校名进行分组，而后面的value则是为了方便统计数量
    (school,1)
    // 对每组的数据进行统计，并且进行排序（参数false代表降序），最后截取前面3条数据
}).reduceByKey(_+_).sortBy(_._2,false).take(3)

// 将数组转化为rdd
val rdd2 = sparkSession.sparkContext.parallelize(top3_arr)
```

- **要求：统计各个学校考试的总次数，平均分，并且按照平均分进行排名（三表操作）**

详见`zipWithIndex()`方法。

使用该方法可以生成排名。