# Spark

# spark架构

# spark部署

## 安装Spark

1.下载spark-2.1.0-bin-hadoop2.7.tgz

2.tar开

3.环境变量

SPARK\_HOME=/soft/spark

PATH=$PATH:$SPARK\_HOME/bin:$SPARK\_HOME/sbin

4.执行spark-shell命令进入

$>spark/bin/spark-shell

5.scala提示符

$scala>

6.体验

//sc对Spark当前环境的封装。

$scala>sc.textFile("/home/centos/1.txt")

.flatMap(\_.split(" "))

.foreach(println)

//使用spark实现wordcount

val rdd1 = sc.textFile("/home/centos/1.txt")

val rdd2 = rdd1.flatMap(\_.split(" "))

val rdd3 = rdd2.map((\_,1))

val rdd4 = rdd.redudeByKey(\_ + \_)

rdd4.collect().forearch(t => println(t.\_1 + " = " + t.\_2))

//一句话实现

sc.textFile("/home/centos/1.txt")

.flatMap(\_.split(" "))

.map((\_,1))

.redudeByKey(\_ + \_)

.collect().forearch(println)

7.联系

统计所有含有字母l的单词的个数。

sc.textFile("/home/centos/1.txt").flatMap(\_.split(" ")).filter(\_.contains("l")).map((\_,1)).reduceByKey(\_ + \_).collect().foreach(println)

//1

sc.textFile("/home/centos/1.txt").flatMap(\_.split(" ")).map((\_,1)).filter(t=>(t \_1).contains("l")).reduceByKey(\_ + \_).collect().foreach(println)

//2.

sc.textFile("/home/centos/1.txt").flatMap(\_.split(" ")).map((\_,1)).reduceByKey(\_ + \_).filter(\_.\_1.contains("l")).collect().foreach(println)

//

sc.textFile("/home/centos/1.txt").flatMap(\_.split(" ")).map((\_,1)).reduceByKey(\_ + \_).collect().foreach(t=>{

if (t.\_1.contains("l")) println(t.\_1 + "=" + t.\_2)

})

## Local模式

## Standalone模式

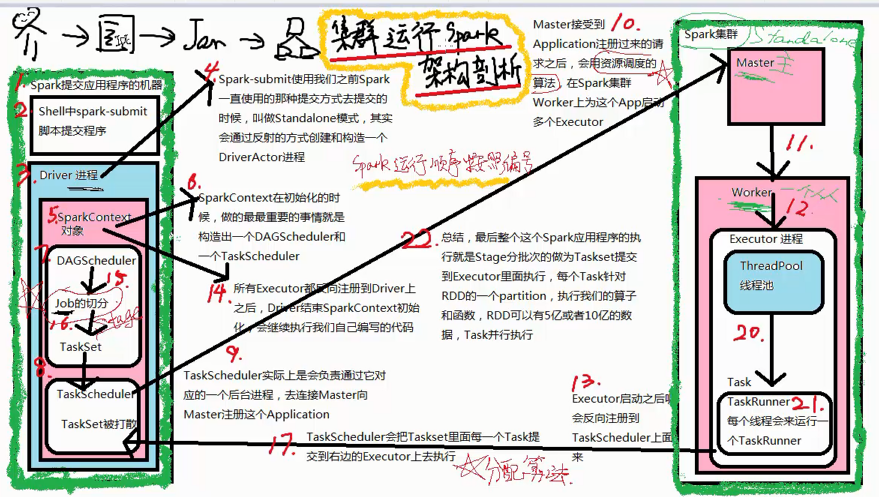
## YARN模式

## MESOS模式

# spark-core

## RDD

## 作业调度流程



# spark-sql

## spark-sql简介

Spark SQL是Spark中的一个模块，主要用于进行**结构化数据的处理**。它提供的最核心的编程抽象，就是DataFrame。同时Spark SQL还可以作为分布式的SQL查询引擎。Spark SQL最重要的功能之一，就是从Hive中查询数据。

Spark 1.0版本开始，推出了Spark SQL。其实最早使用的，都是Hadoop自己的Hive查询引擎；但是后来Spark提供了Shark；再后来Shark被淘汰（严重依赖hive），推出了Spark SQL。Shark的性能比Hive就要高出一个数量级，而Spark SQL的性能又比Shark高出一个数量级。

最早来说，Hive的诞生，主要是因为要让那些不熟悉Java，无法深入进行MapReduce编程的数据分析师，能够使用他们熟悉的关系型数据库的SQL模型，来操作HDFS上的数据。因此推出了Hive。Hive底层基于MapReduce实现SQL功能，能够让数据分析人员，以及数据开发人员，方便的使用Hive进行数据仓库的建模和建设，然后使用SQL模型针对数据仓库中的数据进行统计和分析。但是Hive有个致命的缺陷，就是它的底层基于MapReduce，而MapReduce的shuffle又是基于磁盘的，因此导致Hive的性能异常低下。进场出现复杂的SQL ETL，要运行数个小时，甚至数十个小时的情况。

后来，Spark推出了Shark，Shark与Hive实际上还是紧密关联的，Shark底层很多东西还是依赖于Hive，但是修改了内存管理、物理计划、执行三个模块，底层使用Spark的基于内存的计算模型，从而让性能比Hive提升了数倍到上百倍。

然而，Shark还是它的问题所在，Shark底层依赖了Hive的语法解析器、查询优化器等组件，因此对于其性能的提升还是造成了制约。所以后来Spark团队决定，完全抛弃Shark，推出了全新的Spark SQL项目。Spark SQL就不只是针对Hive中的数据了，而且可以支持其他很多数据源的查询。

### Spark SQL的特点

1、支持多种数据源：Hive、RDD、Parquet、JSON、JDBC等。

2、多种性能优化技术：in-memory columnar storage、byte-code generation、cost model动态评估等。

3、组件扩展性：对于SQL的语法解析器、分析器以及优化器，用户都可以自己重新开发，并且动态扩展。

### Spark SQL的性能优化技术简介

**1、内存列存储（in-memory columnar storage）**

内存列存储意味着，Spark SQL的数据，不是使用Java对象的方式来进行存储，而是使用面向列的内存存储的方式来进行存储。也就是说，每一列，作为一个数据存储的单位。从而大大优化了内存使用的效率。采用了内存列存储之后，减少了对内存的消耗，也就避免了gc大量数据的性能开销。

**2、字节码生成技术（byte-code generation）**

Spark SQL在其catalyst模块的expressions中增加了codegen模块，对于SQL语句中的计算表达式，比如select num + num from t这种的sql，就可以使用动态字节码生成技术来优化其性能。

3、Scala代码编写的优化

对于Scala代码编写中，可能会造成较大性能开销的地方，自己重写，使用更加复杂的方式，来获取更好的性能。比如Option样例类、for循环、map/filter/foreach等高阶函数，以及不可变对象，都改成了用null、while循环等来实现，并且重用可变的对象。

## DataFrame

DataFrame，可以理解为是，**以列的形式组织的，分布式的数据集合**。它其实和关系型数据库中的表非常类似，但是底层做了很多的优化。DataFrame可以通过很多来源进行构建，包括：结构化的数据文件，Hive中的表，外部的关系型数据库，以及RDD。

### SQLContext

要使用Spark SQL，首先就得创建一个SQLContext对象，或者是它的子类的对象，比如HiveContext的对象。

Java版本：

JavaSparkContext sc = ...;

SQLContext sqlContext = new SQLContext(sc);

Scala版本：

val sc: SparkContext = ...

val sqlContext = new SQLContext(sc)

import sqlContext.implicits.\_

### HiveContext

除了基本的SQLContext以外，还可以使用它的子类——HiveContext。HiveContext的功能除了包含SQLContext提供的所有功能之外，还包括了额外的专门针对Hive的一些功能。这些额外功能包括：**使用HiveQL语法来编写和执行SQL，使用Hive中的UDF函数**，从Hive表中读取数据。

要使用HiveContext，就必须预先安装好Hive，SQLContext支持的数据源，HiveContext也同样支持——而不只是支持Hive。对于Spark 1.3.x以上的版本，都推荐使用HiveContext，因为其功能更加丰富和完善。

Spark SQL还支持用spark.sql.dialect参数设置SQL的方言。使用SQLContext的setConf()即可进行设置。对于SQLContext，它只支持“sql”一种方言。对于HiveContext，它默认的方言是“hiveql”。

### 创建DataFrame

使用SQLContext，可以从RDD、Hive表或者其他数据源，来创建一个DataFrame。以下是一个使用JSON文件创建DataFrame的例子：

Java版本：

JavaSparkContext sc = ...;

SQLContext sqlContext = new SQLContext(sc);

DataFrame df = sqlContext.read().json("hdfs://spark1:9000/students.json");

df.show();

Scala版本：

val sc: SparkContext = ...

val sqlContext = new SQLContext(sc)

val df = sqlContext.read.json("hdfs://spark1:9000/students.json")

df.show()

### RDD转换为DataFrame

为什么要将RDD转换为DataFrame？因为这样的话，我们就可以直接针对HDFS等任何可以构建为RDD的数据，使用Spark SQL进行SQL查询了。这个功能是无比强大的。想象一下，针对HDFS中的数据，直接就可以使用SQL进行查询。

Spark SQL支持两种方式来将RDD转换为DataFrame。

第一种方式，是使用反射来推断包含了特定数据类型的RDD的元数据。这种基于反射的方式，代码比较简洁，当你已经知道你的RDD的元数据时，是一种非常不错的方式。

第二种方式，是通过编程接口来创建DataFrame，你可以在程序运行时动态构建一份元数据，然后将其应用到已经存在的RDD上。这种方式的代码比较冗长，但是如果在编写程序时，还不知道RDD的元数据，只有在程序运行时，才能动态得知其元数据，那么只能通过这种动态构建元数据的方式。

#### 使用反射方式推断元数据

Java版本：Spark SQL是**支持将包含了JavaBean的RDD转换为DataFrame**的。JavaBean的信息，就定义了元数据。Spark SQL现在是不支持将包含了嵌套JavaBean或者List等复杂数据的JavaBean，作为元数据的。只支持一个包含简单数据类型的field的JavaBean。

Scala版本：而Scala由于其具有隐式转换的特性，所以Spark SQL的Scala接口，是支持自动将包含了case class的RDD转换为DataFrame的。case class就定义了元数据。Spark SQL会通过反射读取传递给case class的参数的名称，然后将其作为列名。与Java不同的是，Spark SQL是支持将包含了嵌套数据结构的case class作为元数据的，比如包含了Array等。

// 创建普通的RDD

JavaRDD<String> lines = sc.textFile("C://Users//Administrator//Desktop//students.txt");

JavaRDD<Student> students = lines.map(new Function<String, Student>() {

private static final long serialVersionUID = 1L;

@Override

public Student call(String line) throws Exception {

String[] lineSplited = line.split(",");

Student stu = new Student();

stu.setId(Integer.valueOf(lineSplited[0].trim()));

stu.setName(lineSplited[1]);

stu.setAge(Integer.valueOf(lineSplited[2].trim()));

return stu;

}

});

// 使用反射方式，将RDD转换为DataFrame

// 将Student.class传入进去，其实就是用反射的方式来创建DataFrame

// 因为Student.class本身就是反射的一个应用

// 然后底层还得通过对Student Class进行反射，来获取其中的field

// 这里要求，JavaBean必须实现Serializable接口，是可序列化的

DataFrame studentDF = sqlContext.createDataFrame(students, Student.class);

// 拿到了一个DataFrame之后，就可以将其注册为一个临时表，然后针对其中的数据执行SQL语句

studentDF.registerTempTable("students");

// 针对students临时表执行SQL语句，查询年龄小于等于18岁的学生，就是teenageer

DataFrame teenagerDF = sqlContext.sql("select \* from students where age<= 18");

// 将查询出来的DataFrame，再次转换为RDD

JavaRDD<Row> teenagerRDD = teenagerDF.javaRDD();

/\*\*

\* 如果要用scala开发spark程序

\* 然后在其中，还要实现基于反射的RDD到DataFrame的转换，就必须得用object extends App的方式

\* 不能用def main()方法的方式，来运行程序，否则就会报no typetag for ...class的错误

\* @author Administrator

\*/

object RDD2DataFrameReflection **extends App** {

val conf = new SparkConf()

.setMaster("local")

.setAppName("RDD2DataFrameReflection")

val sc = new SparkContext(conf)

val sqlContext = new SQLContext(sc)

// 在Scala中使用反射方式，进行RDD到DataFrame的转换，需要手动导入一个隐式转换

import sqlContext.implicits.\_

case class Student(id: Int, name: String, age: Int)

// 这里其实就是一个普通的，元素为case class的RDD

// 直接对它使用toDF()方法，即可转换为DataFrame

val studentDF = sc.textFile("C://Users//Administrator//Desktop//students.txt", 1)

.map { line => line.split(",") }

.map { arr => Student(arr(0).trim().toInt, arr(1), arr(2).trim().toInt) }

.toDF()

studentDF.registerTempTable("students")

val teenagerDF = sqlContext.sql("select \* from students where age<=18")

val teenagerRDD = teenagerDF.rdd

// 在scala中，row中的数据的顺序，反而是按照我们期望的来排列的，这个跟java是不一样的哦

teenagerRDD.map { row => Student(row(0).toString().toInt, row(1).toString(), row(2).toString().toInt) }

.collect()

.foreach { stu => println(stu.id + ":" + stu.name + ":" + stu.age) }

// 在scala中，对row的使用，比java中的row的使用，更加丰富

// 在scala中，可以用row的getAs()方法，获取指定列名的列

teenagerRDD.map { row => Student(row.getAs[Int]("id"), row.getAs[String]("name"), row.getAs[Int]("age")) }

.collect()

.foreach { stu => println(stu.id + ":" + stu.name + ":" + stu.age) }

// 还可以通过row的getValuesMap()方法，获取指定几列的值，返回的是个map

val studentRDD = teenagerRDD.map { row => {

val map = row.getValuesMap[Any](Array("id", "name", "age"));

Student(map("id").toString().toInt, map("name").toString(), map("age").toString().toInt)

}

}

studentRDD.collect().foreach { stu => println(stu.id + ":" + stu.name + ":" + stu.age) }

}

#### 使用编程方式指定元数据

Java版本：当JavaBean无法预先定义和知道的时候，比如要动态从一个文件中读取数据结构，那么就只能用编程方式动态指定元数据了。首先要从原始RDD创建一个元素为Row的RDD；其次要创建一个StructType，来代表Row；最后将动态定义的元数据应用到RDD<Row>上。

// 第一步，创建一个普通的RDD，但是，必须将其转换为RDD<Row>的这种格式

JavaRDD<String> lines = sc.textFile("C://Users//Administrator//Desktop//students.txt");

JavaRDD<Row> studentRDD = lines.map(new Function<String, Row>() {

private static final long serialVersionUID = 1L;

@Override

public Row call(String line) throws Exception {

String[] lineSplited = line.split(",");

return RowFactory.create(

**Integer.valueOf(lineSplited[0]),**

**lineSplited[1],**

**Integer.valueOf(lineSplited[2]));**

}

});

// 第二步，动态构造元数据

// 比如说，id、name等，field的名称和类型，可能都是在程序运行过程中，动态从mysql db里

// 或者是配置文件中，加载出来的，是不固定的

// 所以特别适合用这种编程的方式，来构造元数据

**List<StructField> structFields = new ArrayList<StructField>();**

**structFields.add(DataTypes.createStructField("id", DataTypes.IntegerType, true));**

**structFields.add(DataTypes.createStructField("name", DataTypes.StringType, true));**

**structFields.add(DataTypes.createStructField("age", DataTypes.IntegerType, true));**

**StructType structType = DataTypes.createStructType(structFields);**

// 第三步，使用动态构造的元数据，将RDD转换为DataFrame

**DataFrame studentDF = sqlContext.createDataFrame(studentRDD, structType);**

// 后面，就可以使用DataFrame了

studentDF.registerTempTable("students");

DataFrame teenagerDF = sqlContext.sql("select \* from students where age<=18");

List<Row> rows = teenagerDF.javaRDD().collect();

for(Row row : rows) {

System.out.println(row);

}

}

Scala版本：Scala的实现方式，与Java是基本一样的。

object RDD2DataFrameProgrammatically extends App {

// 第一步，构造出元素为Row的普通RDD

val studentRDD = sc.textFile("C://Users//Administrator//Desktop//students.txt", 1)

.map { line => Row(line.split(",")(0).toInt, line.split(",")(1), line.split(",")(2).toInt) }

// 第二步，编程方式动态构造元数据

**val structType = StructType(Array(**

**StructField("id", IntegerType, true),**

**StructField("name", StringType, true),**

**StructField("age", IntegerType, true)))**

// 第三步，进行RDD到DataFrame的转换

**val studentDF = sqlContext.createDataFrame(studentRDD, structType)**

// 继续正常使用

studentDF.registerTempTable("students")

val teenagerDF = sqlContext.sql("select \* from students where age<=18")

val teenagerRDD = teenagerDF.rdd.collect().foreach { row => println(row) }

}

#### load和save操作

##### 通用的load和save操作

对于Spark SQL的DataFrame来说，无论是从什么数据源创建出来的DataFrame，都有一些共同的load和save操作。load操作主要用于加载数据，创建出DataFrame；save操作，主要用于将DataFrame中的数据保存到文件中。

Java版本

DataFrame df = sqlContext.read().load("users.parquet");

df.select("name", "favorite\_color").write().save("namesAndFavColors.parquet");

Scala版本

val df = sqlContext.read.load("users.parquet")

df.select("name", "favorite\_color").write.save("namesAndFavColors.parquet")

##### 手动指定数据源类型

也可以手动指定用来操作的数据源类型。数据源通常需要使用其全限定名来指定，比如parquet是org.apache.spark.sql.parquet。但是Spark SQL内置了一些数据源类型，比如json，parquet，jdbc等等。实际上，通过这个功能，就可以在不同类型的数据源之间进行转换了。比如将json文件中的数据保存到parquet文件中。默认情况下，如果不指定数据源类型，那么就是parquet。

Java版本

DataFrame df = sqlContext.read().format("json").load("people.json");

df.select("name", "age").write().format("parquet").save("namesAndAges.parquet");

Scala版本

val df = sqlContext.read.format("json").load("people.json")

df.select("name", "age").write.format("parquet").save("namesAndAges.parquet")

##### Save Mode

Spark SQL对于save操作，提供了不同的save mode。主要用来处理，当目标位置，已经有数据时，应该如何处理。而且save操作并不会执行锁操作，并且不是原子的，因此是有一定风险出现脏数据的。

|  |  |
| --- | --- |
| **Save Mode** | **意义** |
| SaveMode.ErrorIfExists (默认) | 如果目标位置已经存在数据，那么抛出一个异常 |
| SaveMode.Append | 如果目标位置已经存在数据，那么将数据追加进去 |
| SaveMode.Overwrite | 如果目标位置已经存在数据，那么就将已经存在的数据删除，用新数据进行覆盖 |
| SaveMode.Ignore | 如果目标位置已经存在数据，那么就忽略，不做任何操作。 |

#### registerTempTable

1万条数据注册成临时表和1亿条数据注册成临时表时，效率上是否会有很大的差距？

临时表的生命周期和创建该表的SqlContext的生命周期相关，是随着SqlContext的消亡而消亡的。

**def** registerTempTable(tableName: String): Unit = **{** sqlContext.registerDataFrameAsTable(**this**, tableName)  
**}**

/\*\*  
 \* Registers the given [[DataFrame]] as a temporary table in the catalog. Temporary tables exist  
 \* only during the lifetime of this instance of SQLContext.  
 \*/  
**private**[sql] **def** registerDataFrameAsTable(df: DataFrame, tableName: String): Unit = **{  
 *catalog***.registerTable(TableIdentifier(tableName), df.***logicalPlan***)  
**}**

## 数据源类型

### 数据源Parquet

#### 使用编程方式加载数据

Parquet是面向分析型业务的列式存储格式，由Twitter和Cloudera合作开发，2015年5月从Apache的孵化器里毕业成为Apache顶级项目，最新的版本是1.8.0。

列式存储和行式存储相比有哪些优势呢？

1、可以跳过不符合条件的数据，只读取需要的数据，降低IO数据量。

2、压缩编码可以降低磁盘存储空间。由于同一列的数据类型是一样的，可以使用更高效的压缩编码（例如Run Length Encoding和Delta Encoding）进一步节约存储空间。

3、只读取需要的列，支持向量运算，能够获取更好的扫描性能。

这里讲解Parquet数据源的第一个知识点，使用编程的方式加载Parquet文件中的数据。

案例：查询用户数据中的用户姓名。

def main(args: Array[String]): Unit = {

val conf = new SparkConf()

.setAppName("ParquetLoadData")

val sc = new SparkContext(conf)

val sqlContext = new SQLContext(sc)

// 读取Parquet文件中的数据，创建一个DataFrame

val usersDF = sqlContext.read.parquet("hdfs://spark1:9000/spark-study/users.parquet")

usersDF.registerTempTable("users")

val userNamesDF = sqlContext.sql("select name from users")

userNamesDF.rdd.map { row => "Name: " + row(0) }.collect()

.foreach { userName => println(userName) }

}

#### 自动分区推断

表分区是一种常见的优化方式，比如Hive中就提供了表分区的特性。在一个分区表中，不同分区的数据通常存储在不同的目录中，分区列的值通常就包含在了分区目录的目录名中。Spark SQL中的Parquet数据源，支持自动根据目录名推断出分区信息。例如，如果将人口数据存储在分区表中，并且使用性别和国家作为分区列。那么目录结构可能如下所示：

tableName

|- gender=male

|- country=US

...

...

...

|- country=CN

...

|- gender=female

|- country=US

...

|- country=CH

如果将/tableName传入SQLContext.read.parquet()或者SQLContext.read.load()方法，那么Spark SQL就会自动根据目录结构，推断出分区信息，是gender和country。即使数据文件中只包含了两列值，name和age，但是Spark SQL返回的DataFrame，调用printSchema()方法时，会打印出四个列的值：name，age，country，gender。这就是自动分区推断的功能。

此外，分区列的数据类型，也是自动被推断出来的。目前，Spark SQL仅支持自动推断出数字类型和字符串类型。有时，用户也许不希望Spark SQL自动推断分区列的数据类型。此时只要设置一个配置即可， spark.sql.sources.partitionColumnTypeInference.enabled，默认为true，即自动推断分区列的类型，设置为false，即不会自动推断类型。禁止自动推断分区列的类型时，所有分区列的类型，就统一默认都是String。

案例：自动推断用户数据的性别和国家

public static void main(String[] args) {

SparkConf conf = new SparkConf()

.setAppName("ParquetPartitionDiscovery");

JavaSparkContext sc = new JavaSparkContext(conf);

SQLContext sqlContext = new SQLContext(sc);

DataFrame usersDF = sqlContext.read().parquet(

"hdfs://spark1:9000/spark-study/users/gender=male/country=US/users.parquet");

usersDF.printSchema();

usersDF.show();

}

#### 合并元数据

如同ProtocolBuffer，Avro，Thrift一样，Parquet也是支持元数据合并的。用户可以在一开始就定义一个简单的元数据，然后随着业务需要，逐渐往元数据中添加更多的列。在这种情况下，用户可能会创建多个Parquet文件，有着多个不同的但是却互相兼容的元数据。Parquet数据源支持自动推断出这种情况，并且进行多个Parquet文件的元数据的合并。

因为元数据合并是一种相对耗时的操作，而且在大多数情况下不是一种必要的特性，从Spark 1.5.0版本开始，默认是关闭Parquet文件的自动合并元数据的特性的。可以通过以下两种方式开启Parquet数据源的自动合并元数据的特性：

1、读取Parquet文件时，将数据源的选项，mergeSchema，设置为true

2、使用SQLContext.setConf()方法，将spark.sql.parquet.mergeSchema参数设置为true

案例：合并学生的基本信息，和成绩信息的元数据

object ParquetMergeSchema {

def main(args: Array[String]): Unit = {

val conf = new SparkConf()

.setAppName("ParquetMergeSchema")

val sc = new SparkContext(conf)

val sqlContext = new SQLContext(sc)

import sqlContext.implicits.\_

// 创建一个DataFrame，作为学生的基本信息，并写入一个parquet文件中

val studentsWithNameAge = Array(("leo", 23), ("jack", 25)).toSeq

val studentsWithNameAgeDF = sc.parallelize(studentsWithNameAge, 2).toDF("name", "age")

studentsWithNameAgeDF.save("hdfs://spark1:9000/spark-study/students", "parquet", SaveMode.Append)

// 创建第二个DataFrame，作为学生的成绩信息，并写入一个parquet文件中

val studentsWithNameGrade = Array(("marry", "A"), ("tom", "B")).toSeq

val studentsWithNameGradeDF = sc.parallelize(studentsWithNameGrade, 2).toDF("name", "grade")

studentsWithNameGradeDF.save("hdfs://spark1:9000/spark-study/students", "parquet", SaveMode.Append)

// 首先，第一个DataFrame和第二个DataFrame的元数据肯定是不一样的吧

// 一个是包含了name和age两个列，一个是包含了name和grade两个列

// 所以， 这里期望的是，读取出来的表数据，自动合并两个文件的元数据，出现三个列，name、age、grade

// 用mergeSchema的方式，读取students表中的数据，进行元数据的合并

**val students = sqlContext.read.option("mergeSchema", "true")**

**.parquet("hdfs://spark1:9000/spark-study/students")**

students.printSchema()

students.show()

}

}

### json数据源

Spark SQL可以自动推断JSON文件的元数据，并且加载其数据，创建一个DataFrame。可以使用SQLContext.read.json()方法，针对一个元素类型为String的RDD，或者是一个JSON文件。

但是要注意的是，这里使用的JSON文件与传统意义上的JSON文件是不一样的。每行都必须，也只能包含一个，单独的，自包含的，有效的JSON对象。不能让一个JSON对象分散在多行。否则会报错。

### Hive数据源

Spark SQL支持对Hive中存储的数据进行读写。操作Hive中的数据时，必须创建HiveContext，而不是SQLContext。HiveContext继承自SQLContext，但是增加了在Hive元数据库中查找表，以及用HiveQL语法编写SQL的功能。除了sql()方法，HiveContext还提供了hql()方法，从而用Hive语法来编译sql。

使用HiveContext，可以执行Hive的大部分功能，包括创建表、往表里导入数据以及用SQL语句查询表中的数据。查询出来的数据是一个Row数组。

将hive-site.xml拷贝到spark/conf目录下，将mysql connector拷贝到spark/lib目录下

HiveContext sqlContext = new HiveContext(sc);

sqlContext.sql("CREATE TABLE IF NOT EXISTS students (name STRING, age INT)");

sqlContext.sql("LOAD DATA LOCAL INPATH '/usr/local/spark-study/resources/students.txt' INTO TABLE students");

Row[] teenagers = sqlContext.sql("SELECT name, age FROM students WHERE age<=18").collect();

Spark SQL还允许将数据保存到Hive表中。调用DataFrame的saveAsTable命令，即可将DataFrame中的数据保存到Hive表中。与registerTempTable不同，saveAsTable是会将DataFrame中的数据物化到Hive表中的，而且还会在Hive元数据库中创建表的元数据。

默认情况下，saveAsTable会创建一张Hive Managed Table，也就是说，数据的位置都是由元数据库中的信息控制的。当Managed Table被删除时，表中的数据也会一并被物理删除。

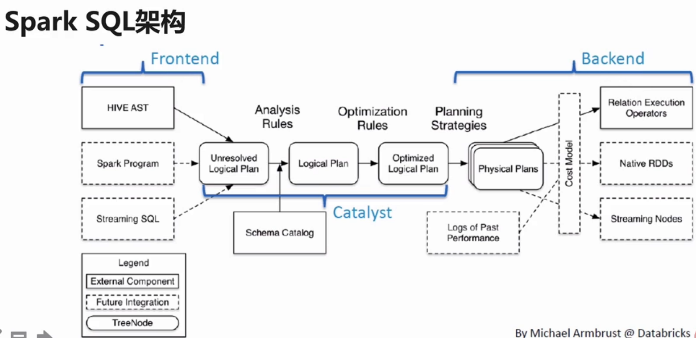
registerTempTable只是注册一个临时的表，只要Spark Application重启或者停止了，那么表就没了。而saveAsTable创建的是物化的表，无论Spark Application重启或者停止，表都会一直存在。

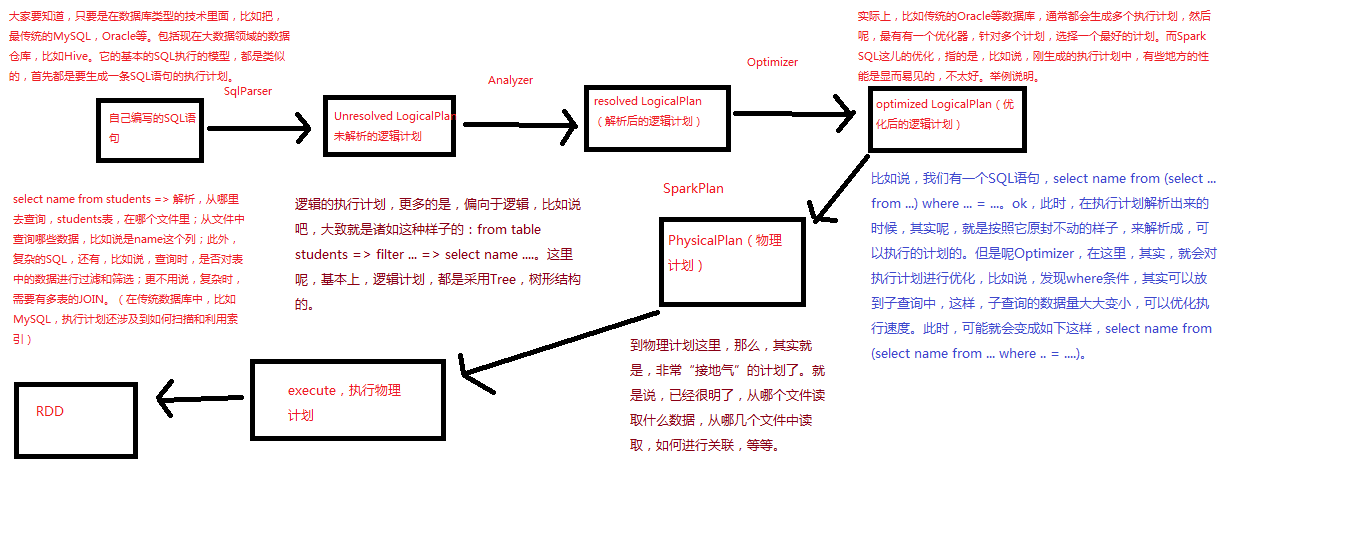
调用HiveContext.table()方法，还可以直接针对Hive中的表，创建一个DataFrame。

案例：查询分数大于80分的学生的完整信息

### JDBC数据源

## Spark sql 架构





## 案例

### 每日UV

// 这里着重说明一下！！！

// 要使用Spark SQL的内置函数，就必须在这里导入SQLContext下的隐式转换

import sqlContext.implicits.\_

// 构造用户访问日志数据，并创建DataFrame

// 模拟用户访问日志，日志用逗号隔开，第一列是日期，第二列是用户id

val userAccessLog = Array(

"2015-10-01,1122",

"2015-10-01,1122",

"2015-10-01,1123",

"2015-10-01,1124",

"2015-10-01,1124",

"2015-10-02,1122",

"2015-10-02,1121",

"2015-10-02,1123",

"2015-10-02,1123");

val userAccessLogRDD = sc.parallelize(userAccessLog, 5)

// 将模拟出来的用户访问日志RDD，转换为DataFrame

// 首先，将普通的RDD，转换为元素为Row的RDD

val userAccessLogRowRDD = userAccessLogRDD

.map { log => Row(log.split(",")(0), log.split(",")(1).toInt) }

// 构造DataFrame的元数据

val structType = StructType(Array(

StructField("date", StringType, true),

StructField("userid", IntegerType, true)))

// 使用SQLContext创建DataFrame

val userAccessLogRowDF = sqlContext.createDataFrame(userAccessLogRowRDD, structType)

// 这里讲解一下uv的基本含义和业务

// 每天都有很多用户来访问，但是每个用户可能每天都会访问很多次

// 所以，uv，指的是，对用户进行去重以后的访问总数

**// 这里，正式开始使用Spark 1.5.x版本提供的最新特性，内置函数，countDistinct**

**// 讲解一下聚合函数的用法**

**// 首先，对DataFrame调用groupBy()方法，对某一列进行分组**

**// 然后，调用agg()方法 ，第一个参数，必须，必须，传入之前在groupBy()方法中出现的字段**

**// 第二个参数，传入countDistinct、sum、first等，Spark提供的内置函数**

**// 内置函数中，传入的参数，也是用单引号作为前缀的，其他的字段**

userAccessLogRowDF.groupBy("date")

.agg('date, countDistinct('userid))

.map { row => Row(row(1), row(2)) }

.collect()

.foreach(println)

}

}

### UDAF（count）自定义分区函数

class StringCount extends UserDefinedAggregateFunction {

// inputSchema，指的是，输入数据的类型

def inputSchema: StructType = {

StructType(Array(StructField("str", StringType, true)))

}

// bufferSchema，指的是，中间进行聚合时，所处理的数据的类型

def bufferSchema: StructType = {

StructType(Array(StructField("count", IntegerType, true)))

}

// dataType，指的是，函数返回值的类型

def dataType: DataType = {

IntegerType

}

def deterministic: Boolean = {

true

}

// 为每个分组的数据执行初始化操作

def initialize(buffer: MutableAggregationBuffer): Unit = {

buffer(0) = 0

}

// 指的是，每个分组，有新的值进来的时候，如何进行分组对应的聚合值的计算

def update(buffer: MutableAggregationBuffer, input: Row): Unit = {

buffer(0) = buffer.getAs[Int](0) + 1

}

// 由于Spark是分布式的，所以一个分组的数据，可能会在不同的节点上进行局部聚合，就是update

// 但是，最后一个分组，在各个节点上的聚合值，要进行merge，也就是合并

def merge(buffer1: MutableAggregationBuffer, buffer2: Row): Unit = {

buffer1(0) = buffer1.getAs[Int](0) + buffer2.getAs[Int](0)

}

// 最后，指的是，一个分组的聚合值，如何通过中间的缓存聚合值，最后返回一个最终的聚合值

def evaluate(buffer: Row): Any = {

buffer.getAs[Int](0)

}

}

def main(args: Array[String]): Unit = {

val conf = new SparkConf()

.setMaster("local")

.setAppName("UDAF")

val sc = new SparkContext(conf)

val sqlContext = new SQLContext(sc)

// 构造模拟数据

val names = Array("Leo", "Marry", "Jack", "Tom", "Tom", "Tom", "Leo")

val namesRDD = sc.parallelize(names, 5)

val namesRowRDD = namesRDD.map { name => Row(name) }

val structType = StructType(Array(StructField("name", StringType, true)))

val namesDF = sqlContext.createDataFrame(namesRowRDD, structType)

// 注册一张names表

namesDF.registerTempTable("names")

// 定义和注册自定义函数

// 定义函数：自己写匿名函数

// 注册函数：SQLContext.udf.register()

sqlContext.udf.register("strCount", new StringCount)

// 使用自定义函数

sqlContext.sql("select name,strCount(name) from names group by name")

.collect()

.foreach(println)

}

# spark-streaming

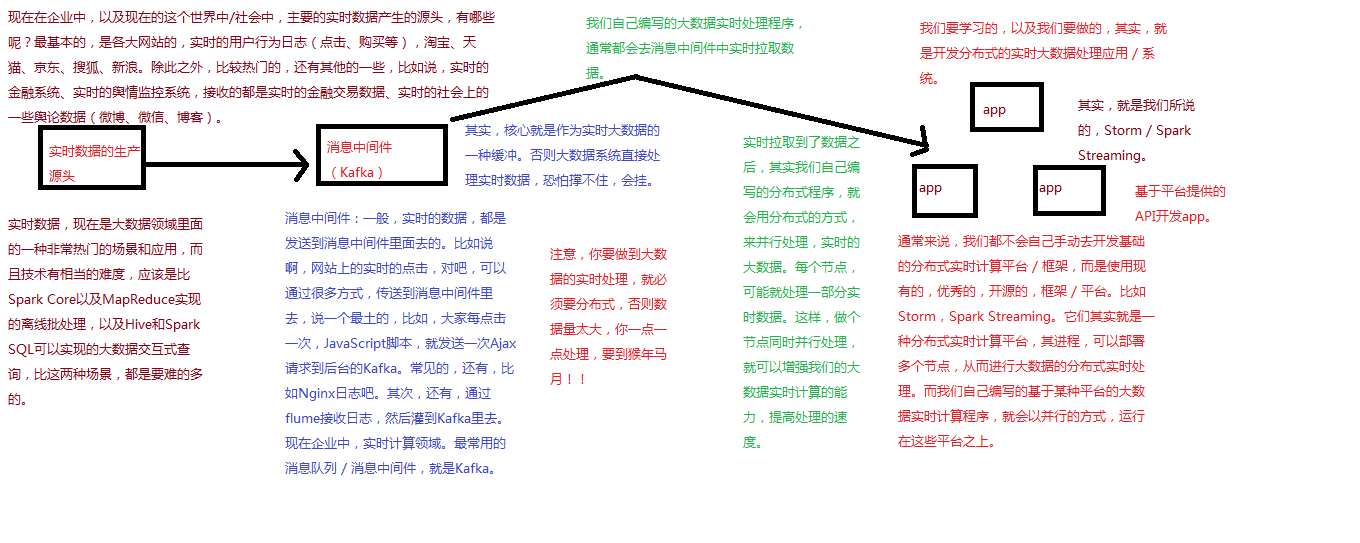
## spark-streaming简介

对core API扩展，针对实时流计算、可伸缩、容错、高吞吐量。数据从各种源头中抓取出来。kafka、socket、flume。计算结果可以推送到filesystem,database，dashboards。

在流计算中也可以应用机器学习和图计算。

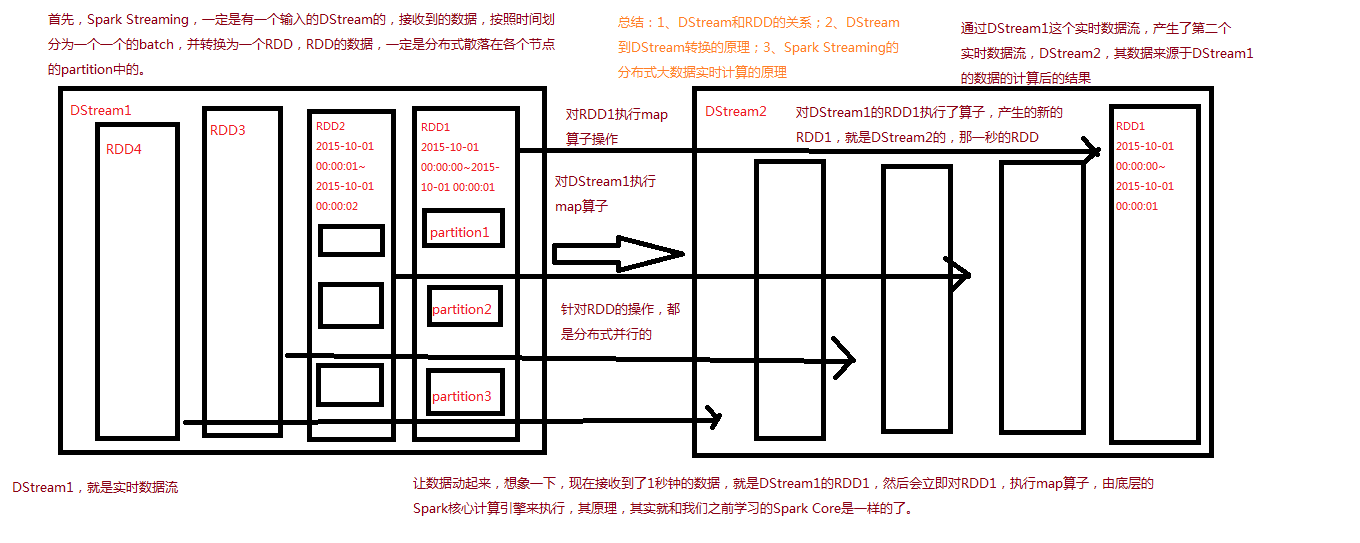
在内部接受实时数据，分割成小批次。通过spark引擎计算，对每个批次产生结果。





## DStream

离散流,连续数据流，DStream可通过数据源创建，也可通过DStream变换得到。



DStream的内部，其实一系列持续不断产生的RDD。RDD是Spark Core的核心抽象，即，不可变的，分布式的数据集。DStream中的每个RDD都包含了一个时间段内的数据。



对DStream应用的算子，比如map，其实在底层会被翻译为对DStream中每个RDD的操作。比如对一个DStream执行一个map操作，会产生一个新的DStream。但是，在底层，其实其原理为，对输入DStream中每个时间段的RDD，都应用一遍map操作，然后生成的新的RDD，即作为新的DStream中的那个时间段的一个RDD。底层的RDD的transformation操作，其实，还是由Spark Core的计算引擎来实现的。Spark Streaming对Spark Core进行了一层封装，隐藏了细节，然后对开发人员提供了方便易用的高层次的API。

### DStream的transformation操作

|  |  |
| --- | --- |
| **Transformation** | **Meaning** |
| map | 对传入的每个元素，返回一个新的元素 |
| flatMap | 对传入的每个元素，返回一个或多个元素 |
| filter | 对传入的元素返回true或false，返回的false的元素被过滤掉 |
| union | 将两个DStream进行合并 |
| count | 返回元素的个数 |
| reduce | 对所有values进行聚合 |
| countByValue | 对元素按照值进行分组，对每个组进行计数，最后返回<K, V>的格式 |
| reduceByKey | 对key对应的values进行聚合 |
| cogroup | 对两个DStream进行连接操作，一个key连接起来的两个RDD的数据，都会以Iterable<V>的形式，出现在一个Tuple中。 |
| join | 对两个DStream进行join操作，每个连接起来的pair，作为新DStream的RDD的一个元素 |
| transform | 对数据进行转换操作 |
| updateStateByKey | 为每个key维护一份state，并进行更新（这个，我认为，是在普通的实时计算中，最有用的一种操作） |
| window | 对滑动窗口数据执行操作（实时计算中最有特色的一种操作） |

### DStream的output操作

|  |  |
| --- | --- |
| **Output** | **Meaning** |
| print | 打印每个batch中的前10个元素，主要用于测试，或者是不需要执行什么output操作时，用于简单触发一下job。 |
| saveAsTextFile(prefix, [suffix]) | 将每个batch的数据保存到文件中。每个batch的文件的命名格式为：prefix-TIME\_IN\_MS[.suffix] |
| saveAsObjectFile | 同上，但是将每个batch的数据以序列化对象的方式，保存到SequenceFile中。 |
| saveAsHadoopFile | 同上，将数据保存到Hadoop文件中 |
| foreachRDD | 最常用的output操作，遍历DStream中的每个产生的RDD，进行处理。可以将每个RDD中的数据写入外部存储，比如文件、数据库、缓存等。通常在其中，是针对RDD执行action操作的，比如foreach。 |

DStream中的所有计算，都是由output操作触发的，比如print()。如果没有任何output操作，那么，压根儿就不会执行定义的计算逻辑。

此外，即使你使用了foreachRDD output操作，也必须在里面对RDD执行action操作，才能触发对每一个batch的计算逻辑。否则，光有foreachRDD output操作，在里面没有对RDD执行action操作，也不会触发任何逻辑。

## StreamingContext

一个StreamingContext定义之后，必须做以下几件事情：

1、通过创建输入DStream来创建输入数据源。

2、通过对DStream定义transformation和output算子操作，来定义实时计算逻辑。

3、调用StreamingContext的start()方法，来开始实时处理数据。

4、调用StreamingContext的awaitTermination()方法，来等待应用程序的终止。可以使用CTRL+C手动停止，或者就是让它持续不断的运行进行计算。

5、也可以通过调用StreamingContext的stop()方法，来停止应用程序。

需要注意的要点：

1、只要一个StreamingContext启动之后，就不能再往其中添加任何计算逻辑了。比如执行start()方法之后，还给某个DStream执行一个算子。

2、一个StreamingContext停止之后，是肯定不能够重启的。调用stop()之后，不能再调用start()

3、一个JVM同时只能有一个StreamingContext启动。在你的应用程序中，不能创建两个StreamingContext。

4、调用stop()方法时，会同时停止内部的SparkContext，如果不希望如此，还希望后面继续使用SparkContext创建其他类型的Context，比如SQLContext，那么就用stop(false)。

5、一个SparkContext可以创建多个StreamingContext，只要上一个先用stop(false)停止，再创建下一个即可。

## 缓存与持久化机制

与RDD类似，Spark Streaming也可以让开发人员手动控制，将数据流中的数据持久化到内存中。对DStream调用persist()方法，就可以让Spark Streaming自动将该数据流中的所有产生的RDD，都持久化到内存中。如果要对一个DStream多次执行操作，那么，对DStream持久化是非常有用的。因为多次操作，可以共享使用内存中的一份缓存数据。

对于基于窗口的操作，比如reduceByWindow、reduceByKeyAndWindow，以及基于状态的操作，比如updateStateByKey，默认就隐式开启了持久化机制。即Spark Streaming默认就会将上述操作产生的Dstream中的数据，缓存到内存中，不需要开发人员手动调用persist()方法。

对于通过网络接收数据的输入流，比如socket、Kafka、Flume等，默认的持久化级别，是将数据复制一份，以便于容错。相当于是，用的是类似MEMORY\_ONLY\_SER\_2。

与RDD不同的是，默认的持久化级别，统一都是要序列化的。

## 输入DStream和Receiver

每个输入DStream（文件流除外）都与Receiver对象相关联，该对象从源接收数据并将其存储在Spark的内存中进行处理。

Spark Streaming提供了两种内置的数据源支持；  
1、基础数据源：StreamingContext API中直接提供了对这些数据源的支持，比如文件、socket、Akka Actor等。

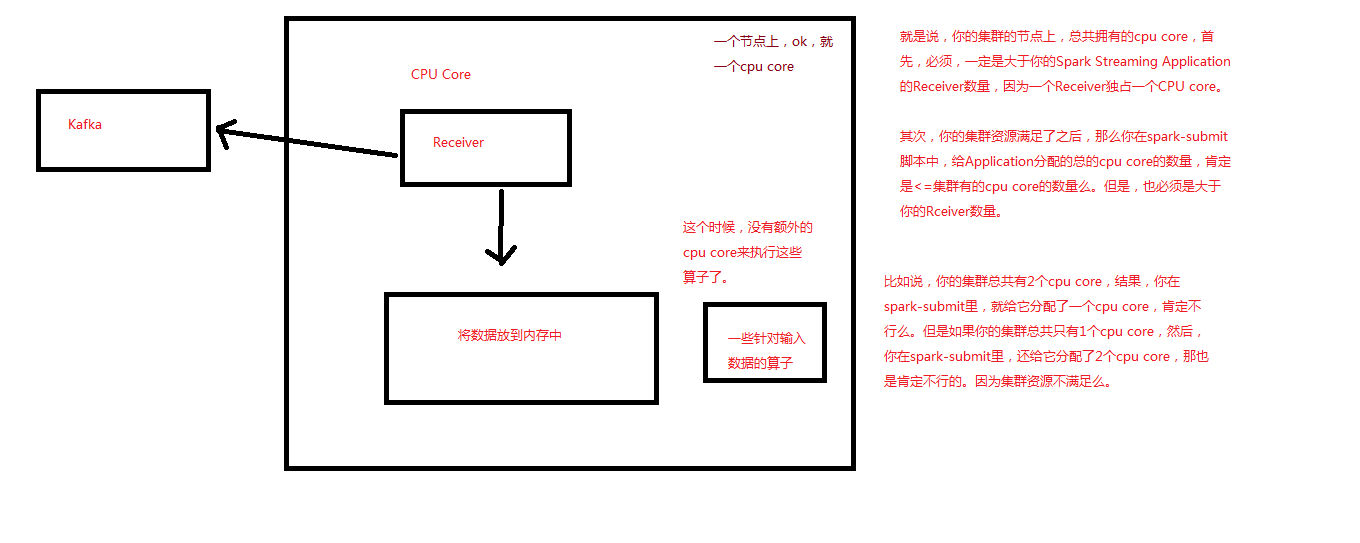
2、高级数据源：诸如Kafka、Flume、Kinesis、Twitter等数据源，通过第三方工具类提供支持。这些数据源的使用，需要引用其依赖。

3、自定义数据源：我们可以自己定义数据源，来决定如何接受和存储数据。

要注意的是，如果你想要在实时计算应用中并行接收多条数据流，可以创建多个输入DStream。这样就会创建多个Receiver，从而并行地接收多个数据流。但是要注意的是，一个Spark Streaming Application的Executor，是一个长时间运行的任务，因此，它会独占分配给Spark Streaming Application的cpu core。从而只要Spark Streaming运行起来以后，这个节点上的cpu core，就没法给其他应用使用了。

使用本地模式，运行程序时，绝对不能用local或者local[1]，因为那样的话，只会给执行输入DStream的executor分配一个线程。而Spark Streaming底层的原理是，至少要有两条线程，一条线程用来分配给Receiver接收数据，一条线程用来处理接收到的数据。因此必须使用local[n]，n>=2的模式。

如果不设置Master，也就是直接将Spark Streaming应用提交到集群上运行，那么首先，必须要求集群节点上，有>1个cpu core，其次，给Spark Streaming的每个executor分配的core，必须>1，这样，才能保证分配到executor上运行的输入DStream，两条线程并行，一条运行Receiver，接收数据；一条处理数据。否则的话，只会接收数据，不会处理数据。



## checkpoint检查点设置

### Checkpoint机制概述

每一个Spark Streaming应用，正常来说，都是要7 \* 24小时运转的，这就是实时计算程序的特点。因为要持续不断的对数据进行计算。因此，对实时计算应用的要求，应该是必须要能够对与应用程序逻辑无关的失败，进行容错。

如果要实现这个目标，Spark Streaming程序就必须将足够的信息checkpoint到容错的存储系统上，从而让它能够从失败中进行恢复。有两种数据需要被进行checkpoint：

1、元数据checkpoint——将定义了流式计算逻辑的信息，保存到容错的存储系统上，比如HDFS。当运行Spark Streaming应用程序的Driver进程所在节点失败时，该信息可以用于进行恢复。元数据信息包括了：

1.1 配置信息——创建Spark Streaming应用程序的配置信息，比如SparkConf中的信息。

1.2 DStream的操作信息——定义了Spark Stream应用程序的计算逻辑的DStream操作信息。

1.3 未处理的batch信息——那些job正在排队，还没处理的batch信息。

2、数据checkpoint——将实时计算过程中产生的RDD的数据保存到可靠的存储系统中。

对于一些将多个batch的数据进行聚合的，有状态的transformation操作，这是非常有用的。在这种transformation操作中，生成的RDD是依赖于之前的batch的RDD的，这会导致随着时间的推移，RDD的依赖链条变得越来越长。

要避免由于依赖链条越来越长，导致的一起变得越来越长的失败恢复时间，有状态的transformation操作执行过程中间产生的RDD，会定期地被checkpoint到可靠的存储系统上，比如HDFS。从而削减RDD的依赖链条，进而缩短失败恢复时，RDD的恢复时间。

一句话概括，元数据checkpoint主要是为了从driver失败中进行恢复；而RDD checkpoint主要是为了，使用到有状态的transformation操作时，能够在其生产出的数据丢失时，进行快速的失败恢复。

### 何时启用Checkpoint机制？

1、使用了有状态的transformation操作——比如updateStateByKey，或者reduceByKeyAndWindow操作，被使用了，那么checkpoint目录要求是必须提供的，也就是必须开启checkpoint机制，从而进行周期性的RDD checkpoint。

2、要保证可以从Driver失败中进行恢复——元数据checkpoint需要启用，来进行这种情况的恢复。

要注意的是，并不是说，所有的Spark Streaming应用程序，都要启用checkpoint机制，如果即不强制要求从Driver失败中自动进行恢复，又没使用有状态的transformation操作，那么就不需要启用checkpoint。事实上，这么做反而是有助于提升性能的。

### 如何启用Checkpoint机制？

1、对于有状态的transformation操作，启用checkpoint机制，定期将其生产的RDD数据checkpoint，是比较简单的。

可以通过配置一个容错的、可靠的文件系统（比如HDFS）的目录，来启用checkpoint机制，checkpoint数据就会写入该目录。使用StreamingContext的checkpoint()方法即可。然后，你就可以放心使用有状态的transformation操作了。

2、如果为了要从Driver失败中进行恢复，那么启用checkpoint机制，是比较复杂的。需要改写Spark Streaming应用程序。

当应用程序第一次启动的时候，需要创建一个新的StreamingContext，并且调用其start()方法，进行启动。当Driver从失败中恢复过来时，需要从checkpoint目录中记录的元数据中，恢复出来一个StreamingContext。

def functionToCreateContext(): StreamingContext = {

val ssc = new StreamingContext(...)

val lines = ssc.socketTextStream(...)

ssc.checkpoint(checkpointDirectory)

ssc

}

val context = StreamingContext.getOrCreate(checkpointDirectory, functionToCreateContext \_)

context.start()

context.awaitTermination()

### 配置spark-submit提交参数

按照上述方法，进行Spark Streaming应用程序的重写后，当第一次运行程序时，如果发现checkpoint目录不存在，那么就使用定义的函数来第一次创建一个StreamingContext，并将其元数据写入checkpoint目录；当从Driver失败中恢复过来时，发现checkpoint目录已经存在了，那么会使用该目录中的元数据创建一个StreamingContext。

但是上面的重写应用程序的过程，只是实现Driver失败自动恢复的第一步。第二步是，必须确保Driver可以在失败时，自动被重启。

要能够自动从Driver失败中恢复过来，运行Spark Streaming应用程序的集群，就必须监控Driver运行的过程，并且在它失败时将它重启。对于Spark自身的standalone模式，需要进行一些配置去supervise driver，在它失败时将其重启。

首先，要在spark-submit中，添加--deploy-mode参数，默认其值为client，即在提交应用的机器上启动Driver；但是，要能够自动重启Driver，就必须将其值设置为cluster；此外，需要添加--supervise参数。

使用上述第二步骤提交应用之后，就可以让driver在失败时自动被重启，并且通过checkpoint目录的元数据恢复StreamingContext。

将RDD checkpoint到可靠的存储系统上，会耗费很多性能。当RDD被checkpoint时，会导致这些batch的处理时间增加。因此，checkpoint的间隔，需要谨慎的设置。对于那些间隔很多的batch，比如1秒，如果还要执行checkpoint操作，则会大幅度削减吞吐量。而另外一方面，如果checkpoint操作执行的太不频繁，那就会导致RDD的lineage变长，又会有失败恢复时间过长的风险。

对于那些要求checkpoint的有状态的transformation操作，默认的checkpoint间隔通常是batch间隔的数倍，至少是10秒。使用DStream的checkpoint()方法，可以设置这个DStream的checkpoint的间隔时长。通常来说，将checkpoint间隔设置为窗口操作的滑动间隔的5~10倍，是个不错的选择。

## Spark Streaming 集成kafka

### 基于Receiver的方式

这种方式使用Receiver来获取数据。Receiver是使用Kafka的高层次Consumer API来实现的。receiver从Kafka中获取的数据都是存储在Spark Executor的内存中的，然后Spark Streaming启动的job会去处理那些数据。

然而，在默认的配置下，这种方式可能会因为底层的失败而丢失数据。如果要启用高可靠机制，让数据零丢失，就必须启用Spark Streaming的预写日志机制（Write Ahead Log，WAL）。该机制会同步地将接收到的Kafka数据写入分布式文件系统（比如HDFS）上的预写日志中。所以，即使底层节点出现了失败，也可以使用预写日志中的数据进行恢复。

需要注意的要点

1、Kafka中的topic的partition，与Spark中的RDD的partition是没有关系的。所以，在KafkaUtils.createStream()中，提高partition的数量，只会增加一个Receiver中，读取partition的线程的数量。不会增加Spark处理数据的并行度。

2、可以创建多个Kafka输入DStream，使用不同的consumer group和topic，来通过多个receiver并行接收数据。

3、如果基于容错的文件系统，比如HDFS，启用了预写日志机制，接收到的数据都会被复制一份到预写日志中。因此，在KafkaUtils.createStream()中，设置的持久化级别是StorageLevel.MEMORY\_AND\_DISK\_SER。

### 基于Direct的方式

这种新的不基于Receiver的直接方式，是在Spark 1.3中引入的，从而能够确保更加健壮的机制。替代掉使用Receiver来接收数据后，这种方式会周期性地查询Kafka，来获得每个topic+partition的最新的offset，从而定义每个batch的offset的范围。当处理数据的job启动时，就会使用Kafka的简单consumer api来获取Kafka指定offset范围的数据。

这种方式有如下优点：

1、简化并行读取：如果要读取多个partition，不需要创建多个输入DStream然后对它们进行union操作。Spark会创建跟Kafka partition一样多的RDD partition，并且会并行从Kafka中读取数据。所以在Kafka partition和RDD partition之间，有一个一对一的映射关系。

1. 高性能：如果要保证零数据丢失，在基于receiver的方式中，需要开启WAL机制。这种方式其实效率低下，因为数据实际上被复制了两份，Kafka自己本身就有高可靠的机制，会对数据复制一份，而这里又会复制一份到WAL中。而基于direct的方式，不依赖Receiver，不需要开启WAL机制，只要Kafka中作了数据的复制，那么就可以通过Kafka的副本进行恢复。

3、一次且仅一次的事务机制：

基于receiver的方式，是使用Kafka的高阶API来在ZooKeeper中保存消费过的offset的。这是消费Kafka数据的传统方式。这种方式配合着WAL机制可以保证数据零丢失的高可靠性，但是却无法保证数据被处理一次且仅一次，可能会处理两次。因为Spark和ZooKeeper之间可能是不同步的。

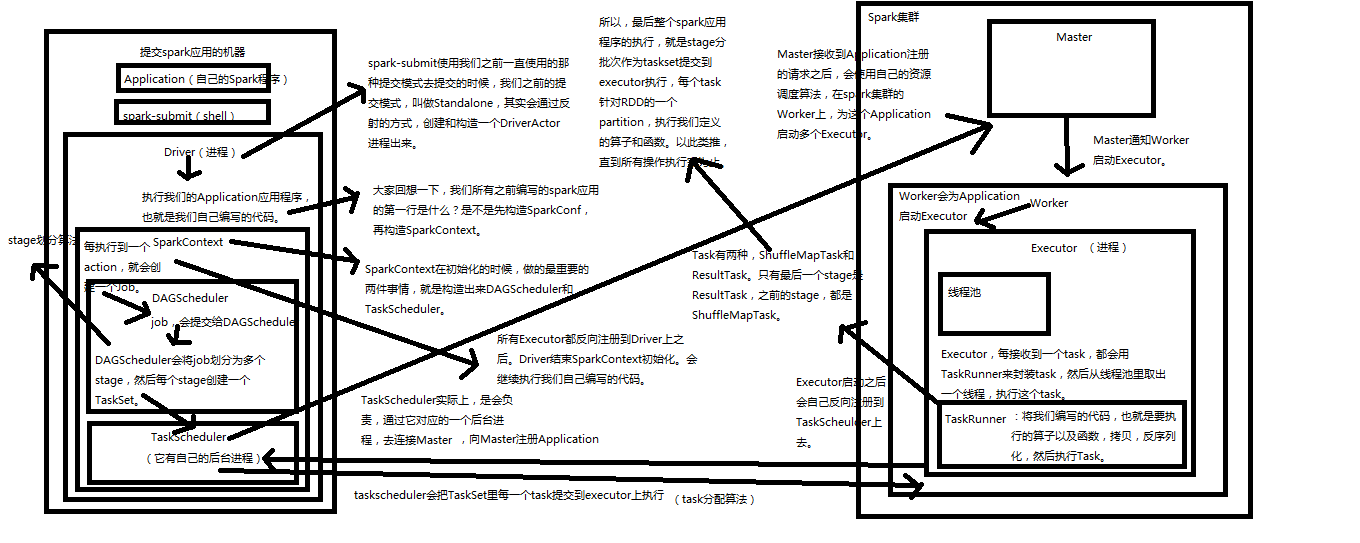
基于direct的方式，使用kafka的简单api，Spark Streaming自己就负责追踪消费的offset，并保存在checkpoint中。Spark自己一定是同步的，因此可以保证数据是消费一次且仅消费一次。

# spark-mllib

# Spark源码剖析

## spark架构原理

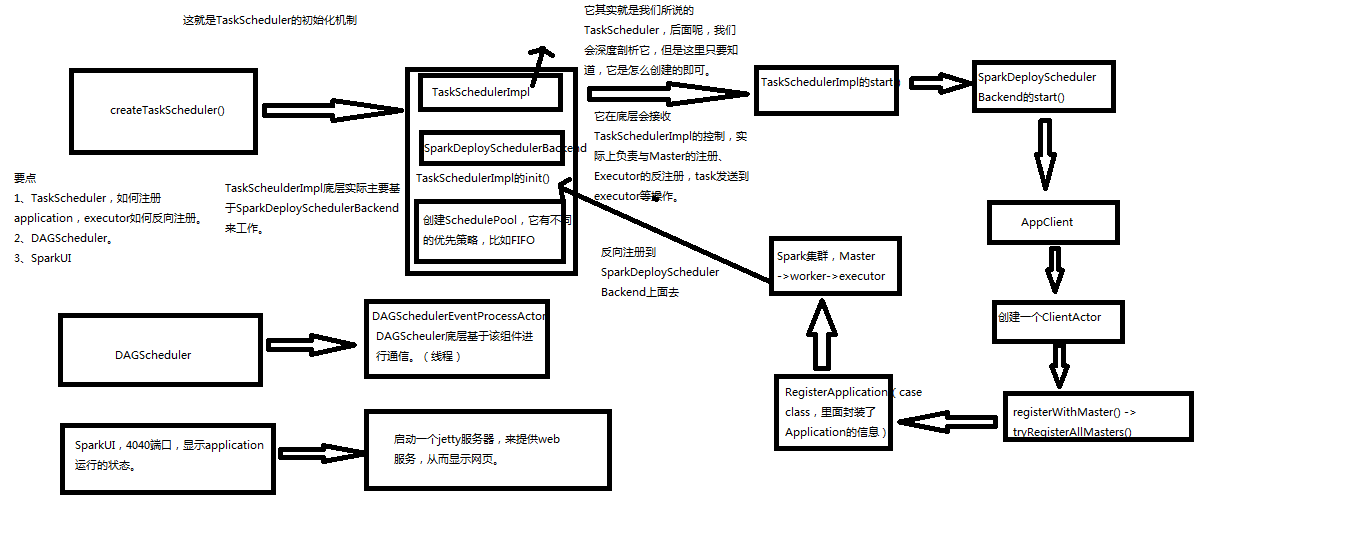
### Spark程序执行流程



1. Spark-submit提交程序
2. Driver进程启动，创建sparkconf与sparkcontext
3. Sparkcontext初始化开始，构造DAGscheduler与taskscheduler
4. Taskscheduler连接master，向master注册application。
5. Master接收application注册请求使用资源调度算法，启动多个worker
6. Work为application启动executer
7. Executer启动后反向注册到driver（taskscheduler）
8. 所有execute都注册到driver后，sparkcontext初始化结束继续执行代码。
9. 每执行到一个action操作都会创建一个job，job会提交给DAGscheduler
10. DAGscheduler会将job划分为多个stage，每个stage创建一个taskset
11. Taskset会由taskscheduler把它其中每一个task提交给executer执行
12. Executer每接收一个task都会用taskRunner来封装task，然后从线程池取出一个线程执行task
13. taskRunner将要执行的算子及函数，拷贝，反序列化，然后执行。
14. Task有两种，shuffleMaptask和ResultTask，只有最后一个stage是ResultTask，之前的stage都是shuffleMaptask

总结：整个spark程序的执行，就是stage分批次的提交到executer执行。每个task针对RDD的一个partition，执行我们的算子和函数。以此类推，直到所有操作执行完。

### SparkContext原理剖析与源码分析



1. 创建 taskscheduler，创建sparkdeployschedulerBachend,创建schedulePool
2. taskschedulerImlp.Start -> sparkdeployschedulerBachend.Start - > AppClient -> ClientActior -> registerWithMaster 。向master发送RegisterApplication（case class），里面封装了Application信息。
3. Master->worker->executor。Executor反向注册到sparkdeployschedulerBachend

#### TaskSchedulerImpl

1. 底层通过SchedulerBackend，针对不同种类的cluster（standalone，yarn，mesos），调度task
2. 它可以通过使用LocalSchedulerBackend，并且将isLocal参数设为true，来在本地模式下工作
3. 它负责处理一些通用逻辑，比如说job调度顺序，启动推测执行等
4. 客户端应该首先调用initialize（）和start（）然后调用 runTasks 方法

#### sparkdeployschedulerBachend

负责接收taskschedulerImlp的控制，master注册，executer的反注册，task发送到executer等操作

#### DAGScheduler

面向stage的调度机制的高层次调度。它会为每一个job计算一个stage的DAG（有向无环图），追踪RDD和stage的输出是否写入磁盘或内存等地方，并且寻找一个最小消耗调度机制来运行job。

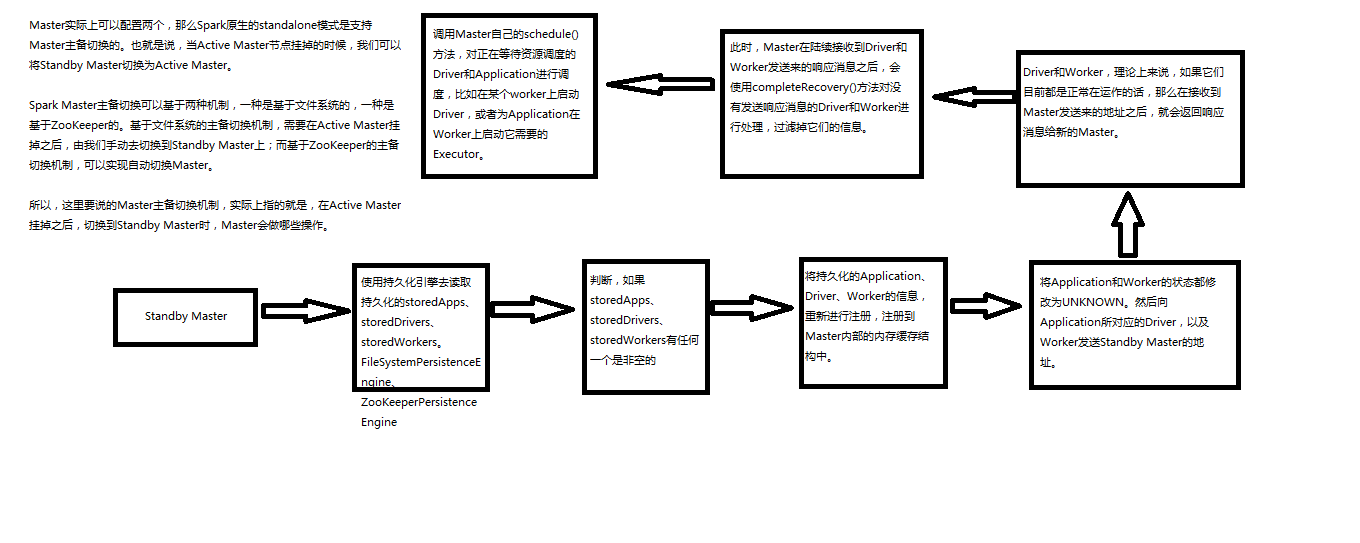
它会将stage作为tasksets提交到底层的taskschedulerImpl上。

它还负责决定运行每一个task的最佳位置，基于当前的缓存状态，将这些最佳位置提交给底层的taskschedulerImpl。此外，它还会处理由于shuffle输出的文件丢失导致的失败，这种情况下，旧的stage可能会被重新提交。一个stage内部失败，如果不是由于shuffle文件丢失导致的，会被taskscheduler处理，它会多次重试每一个task，直到最后，实在不行，才会取消stage。

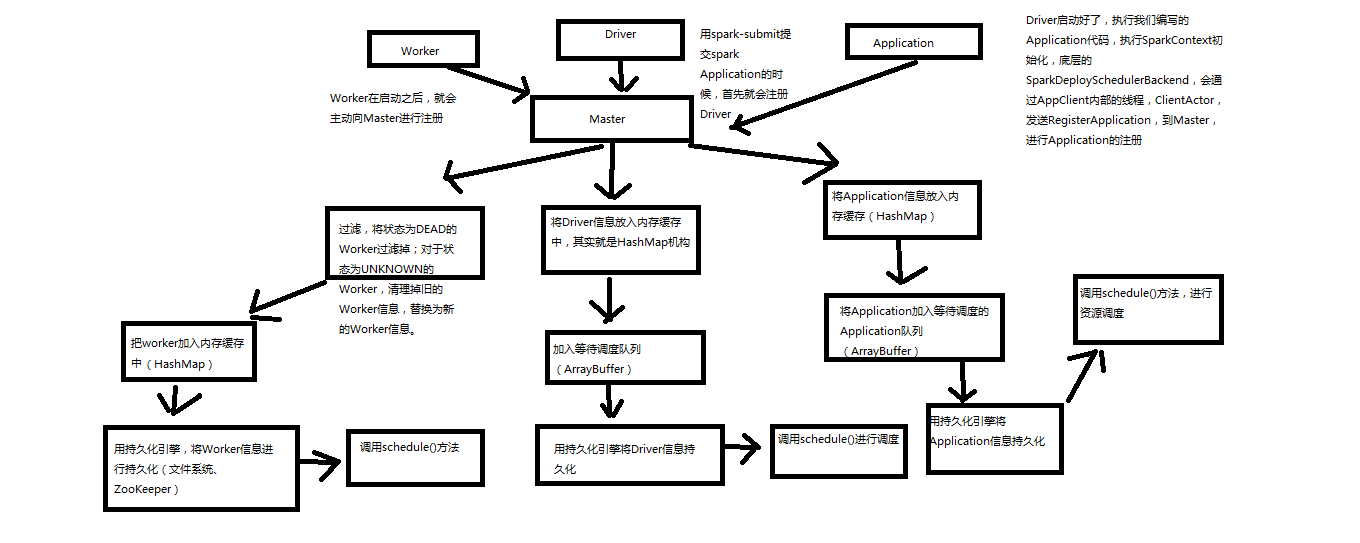
#### SparkUI

启动一个jetty服务器，提供web服务。

### Master主备切换机制原理剖析与源码分析



### Master注册机制原理剖析与源码分析



##### Work向master注册：

1. 过滤：将状态为dead，unknown的worker清理掉
2. 把work加入内存缓存（hashmap）
3. 用持久化引擎 将work的增量信息持久化
4. 调用schedule方法，进行资源调度

##### Driver向master注册：

1. 将driver信息放入缓存，hashmap
2. 加入等待调度队列（ArrayBuffer）
3. 用持久化引擎 将driver的增量信息持久化
4. 调用schedule方法，进行资源调度

##### Application向master注册：

执行sparkcontext初始化，底层的sparkdeployschedulerBachend，会通过AppClient内部线程，ClientActor，发送到RegisterApplication，到master，进行Application的注册。

1. 将Application信息放入缓存，hashmap
2. 加入等待调度队列
3. 用持久化引擎 将Application的增量信息持久化
4. 调用schedule方法，进行资源调度

### Master状态改变处理机制原理剖析与源码分析

### Master资源调度算法原理剖析与源码分析

private def schedule(): Unit = {

//首先判断master状态是否为alive，也就是说，standbymaster不参与资源调度

if (state != RecoveryState.ALIVE) {

return

}

// Random.shuffle 就是对传入的元素进行随机打乱（不断随机交换元素位置）

// 取出来wokers中所有之前注册上来的worker，进行过滤，过滤出状态为alive的work

// 对alive状态的worker进行随机打乱

val shuffledAliveWorkers =

Random.shuffle(workers.toSeq.filter(\_.state == WorkerState.ALIVE))

val numWorkersAlive = shuffledAliveWorkers.size

var curPos = 0

//首先调度driver，除了yarn-cluster模式，其他模式都是在本地启动driver，而不会来

//注册driver，也就不需要调度driver了。

for (driver <- waitingDrivers.toList) { // iterate over a copy of waitingDrivers

var launched = false

var numWorkersVisited = 0

// 只要还有活着的worker没有遍历到就继续遍历，或者是当前driver没有被启动

while (numWorkersVisited < numWorkersAlive && !launched) {

val worker = shuffledAliveWorkers(curPos)

numWorkersVisited += 1

//如果当前这个worker的空闲内存大于driver的需要内存

//并且worker的空闲cpu数量大于driver的需要cpu数量

if (worker.memoryFree >= driver.desc.mem && worker.coresFree >= driver.desc.cores) {

//启动driver

launchDriver(worker, driver)

//从缓存队列中移除

waitingDrivers -= driver

**launched = true**

}

//将指针指向下一个worker

curPos = (curPos + 1) % numWorkersAlive

}

}

startExecutorsOnWorkers()

}

private def launchDriver(worker: WorkerInfo, driver: DriverInfo) {

logInfo("Launching driver " + driver.id + " on worker " + worker.id)

//将driver加入worker内存的缓存结构

// 将worker使用的内存和cpu数量加上driver需要的内存和cpu数量

worker.addDriver(driver)

//同时把work也加入到driver的缓存结构

driver.worker = Some(worker)

// 调用worker的rpc，发送请求，让worker来启动driver

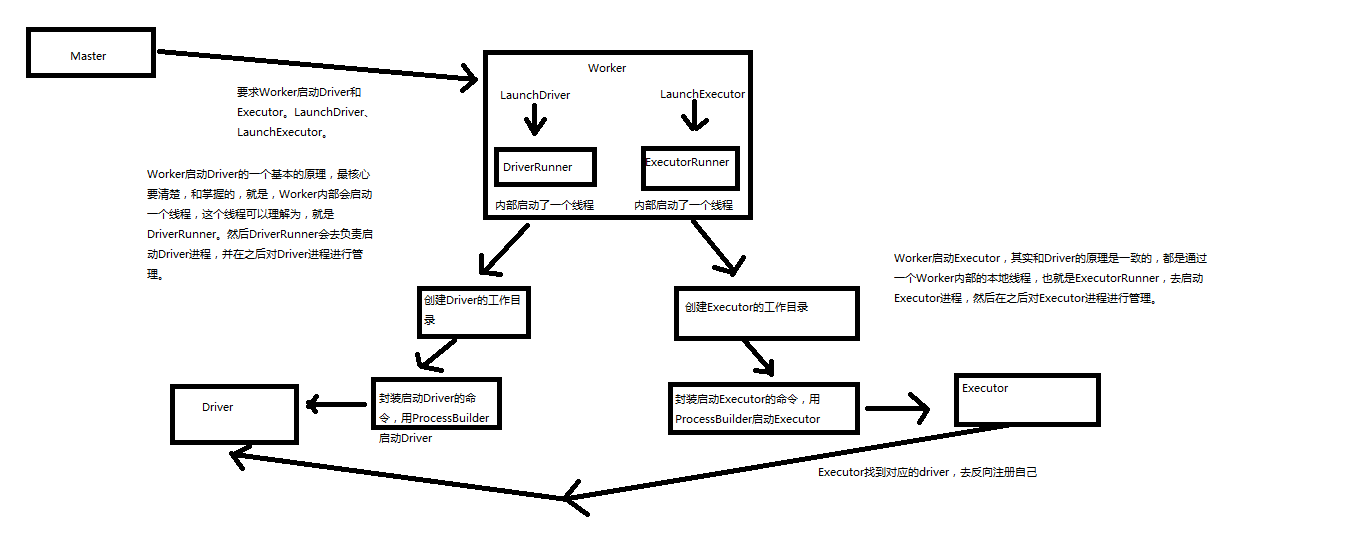
worker.endpoint.send(LaunchDriver(driver.id, driver.desc))

//改变driver状态

driver.state = DriverState.RUNNING

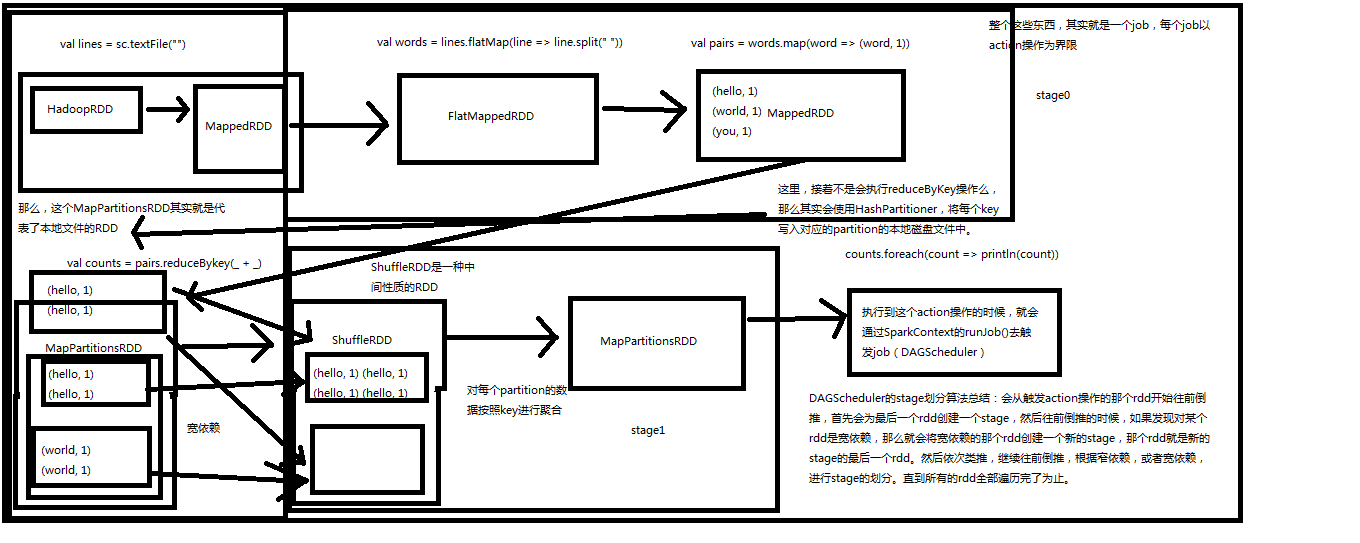
}

### Worker原理剖析与源码分析



### job触发流程原理剖析与源码分析

### DAGScheduler原理剖析与源码分析



#### Stage划分算法

会从触发action操作的那个rdd开始往前倒推，首先会为最后一个rdd创建一个stage，然后如果发现某一个rdd是宽依赖，那么就将宽依赖的那个rdd重新创建一个stage。

#### Task最佳位置计算算法

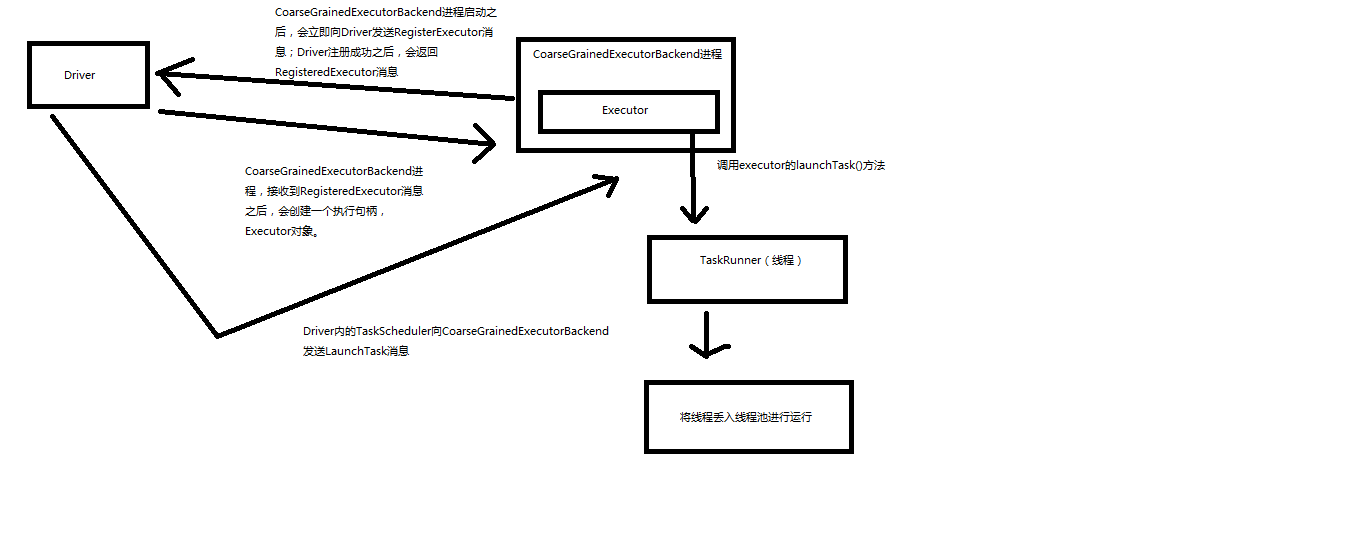
从stage的最后一个rdd开始，去找被cache或者checkpoint的rdd partition

Task的最佳位置就是cache的或者checkpoint的位置。

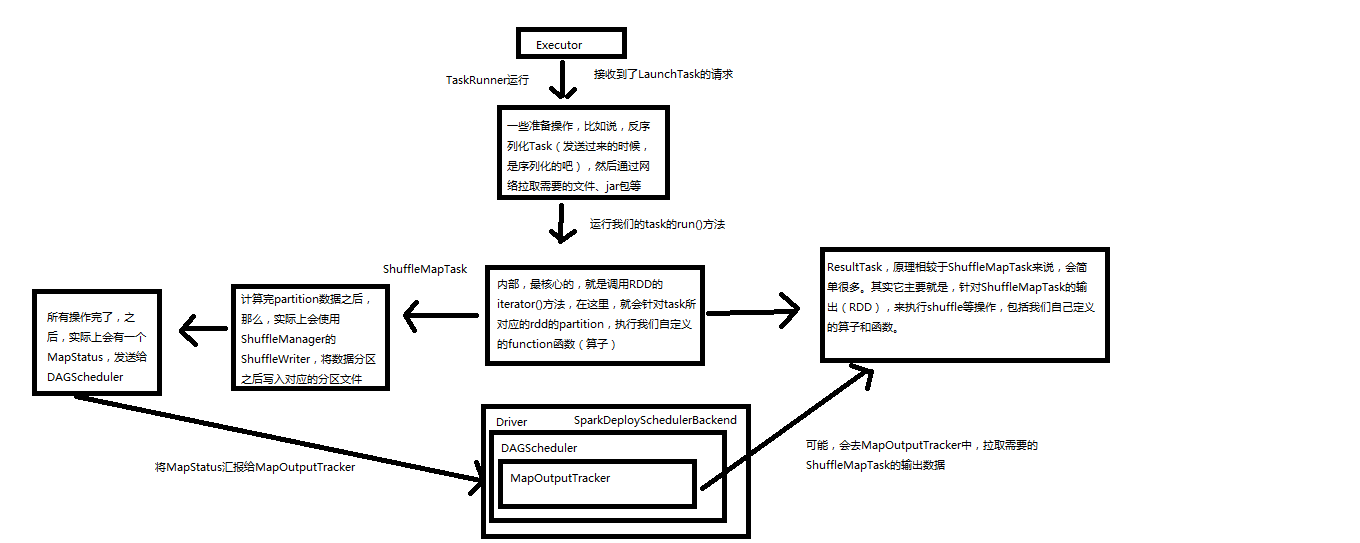
因为在那个节点上，不需要计算之前的rdd了。

### TaskScheduler原理剖析与源码分析

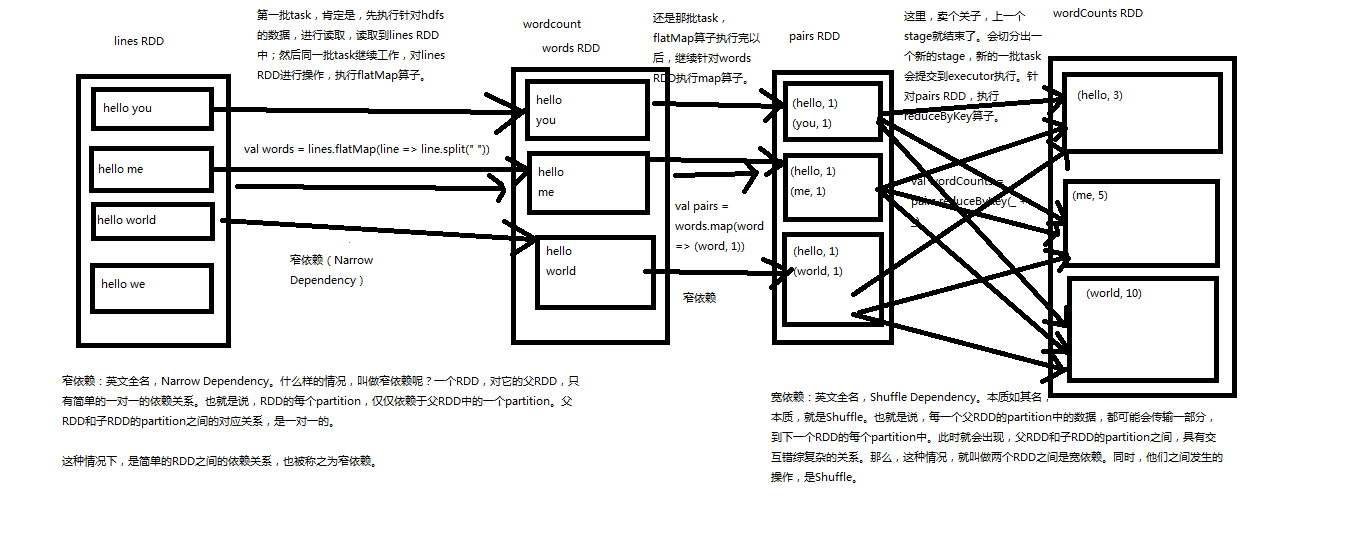
### Executor原理剖析与源码分析



### Task原理剖析与源码分析



## 宽依赖与窄依赖深度剖析

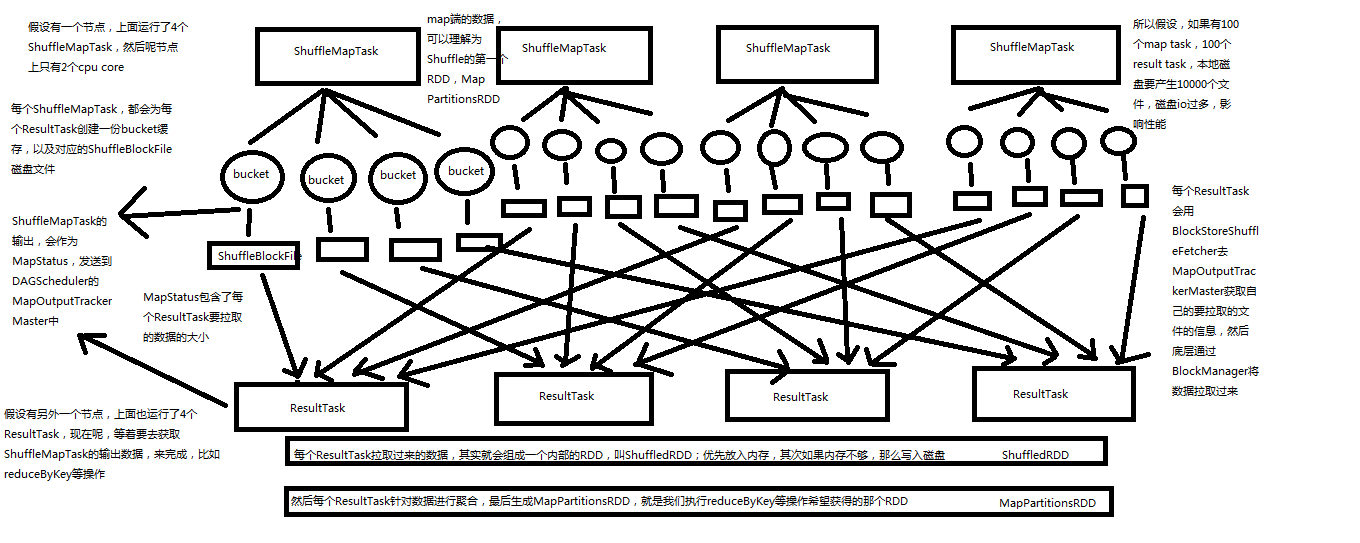


宽依赖：Shuffle Dependency 父RDD的一个partition可能会传输到子RDD每一个partition（可能和每一个partition都有关系）。具有宽依赖的父RDD到子RDD的转化操作，被称为shuffle。

窄依赖：Narrow Dependency 一对一的对应关系

## Shuffle原理剖析与源码分析

### 普通的Shuffle操作原理剖析



### Spark Shuffle操作的两个特点

第一个特点，

在Spark早期版本中，那个bucket缓存是非常非常重要的，因为需要将一个ShuffleMapTask所有的数据都写入内存缓存之后，才会刷新到磁盘。但是这就有一个问题，如果map side数据过多，那么很容易造成内存溢出。所以spark在新版本中，优化了，默认那个内存缓存是100kb，然后呢，写入一点数据达到了刷新到磁盘的阈值之后，就会将数据一点一点地刷新到磁盘。

这种操作的优点，是不容易发生内存溢出。缺点在于，如果内存缓存过小的话，那么可能发生过多的磁盘写io操作。所以，这里的内存缓存大小，是可以根据实际的业务情况进行优化的。

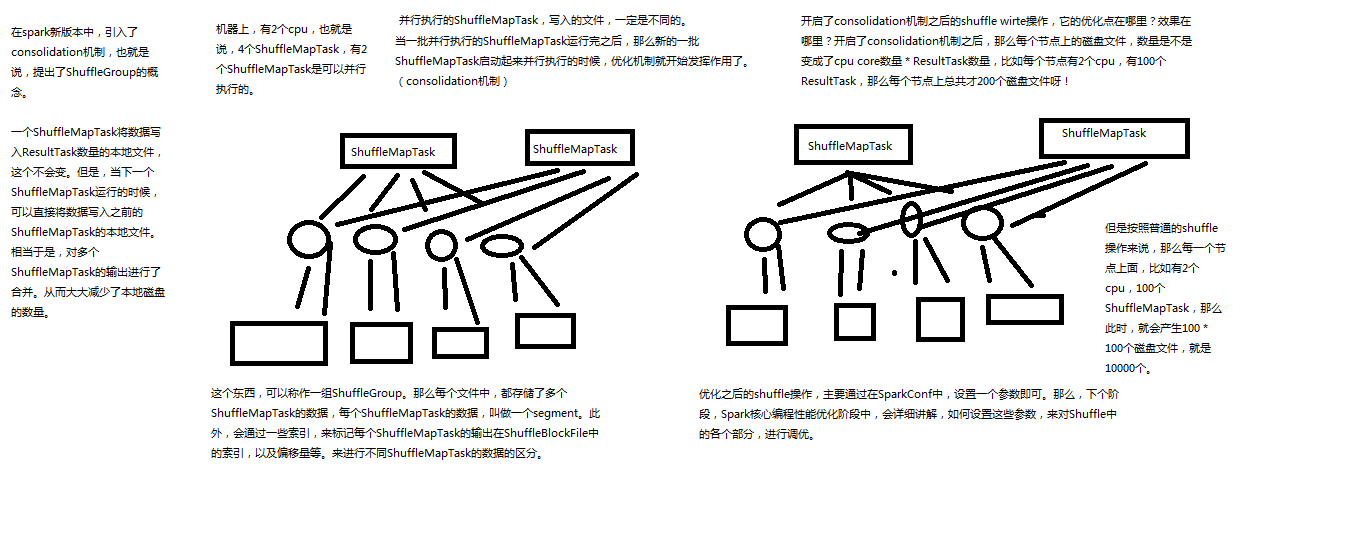
第二个特点，

与MapReduce完全不一样的是，MapReduce它必须将所有的数据都写入本地磁盘文件以后，才能启动reduce操作，来拉取数据。为什么？因为mapreduce要实现默认的根据key的排序！所以要排序，肯定得写完所有数据，才能排序，然后reduce来拉取。

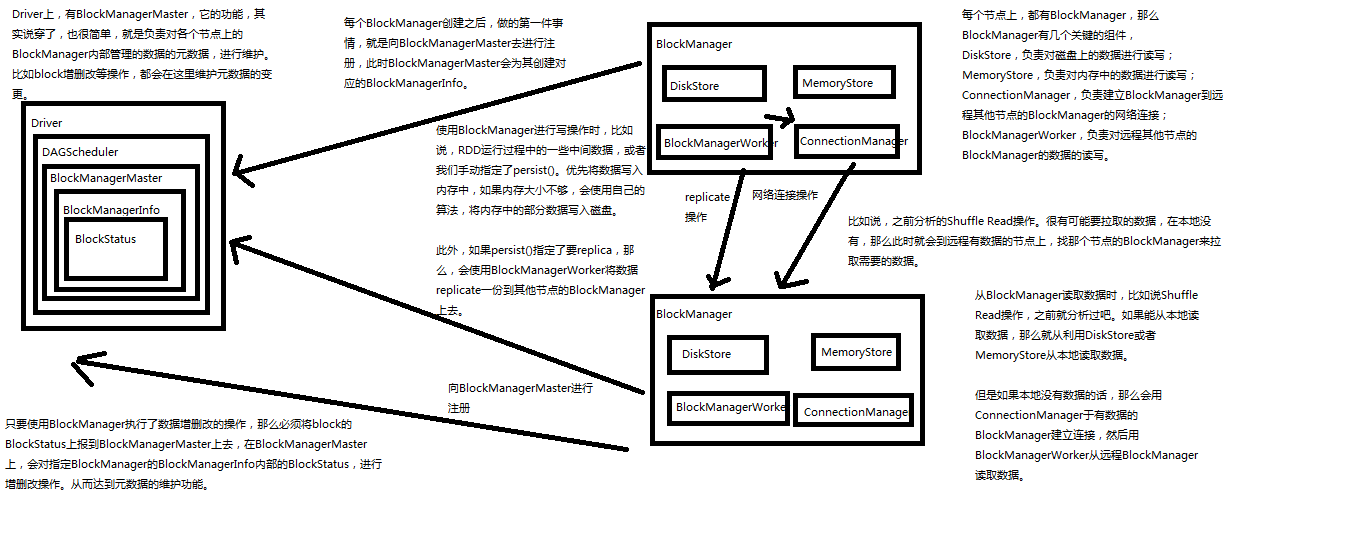
但是Spark不需要**，spark默认情况下，是不会对数据进行排序的**。因此ShuffleMapTask每写入一点数据，ResultTask就可以拉取一点数据，然后在本地执行我们定义的聚合函数和算子，进行计算。

spark这种机制的好处在于，速度比mapreduce快多了。但是也有一个问题，mapreduce提供的reduce，是可以处理每个key对应的value上的，很方便。但是spark中，由于这种实时拉取的机制，因此提供不了，直接处理key对应的values的算子，只能通过groupByKey，先shuffle，有一个MapPartitionsRDD，然后用map算子，来处理每个key对应的values。就没有mapreduce的计算模型那么方便。

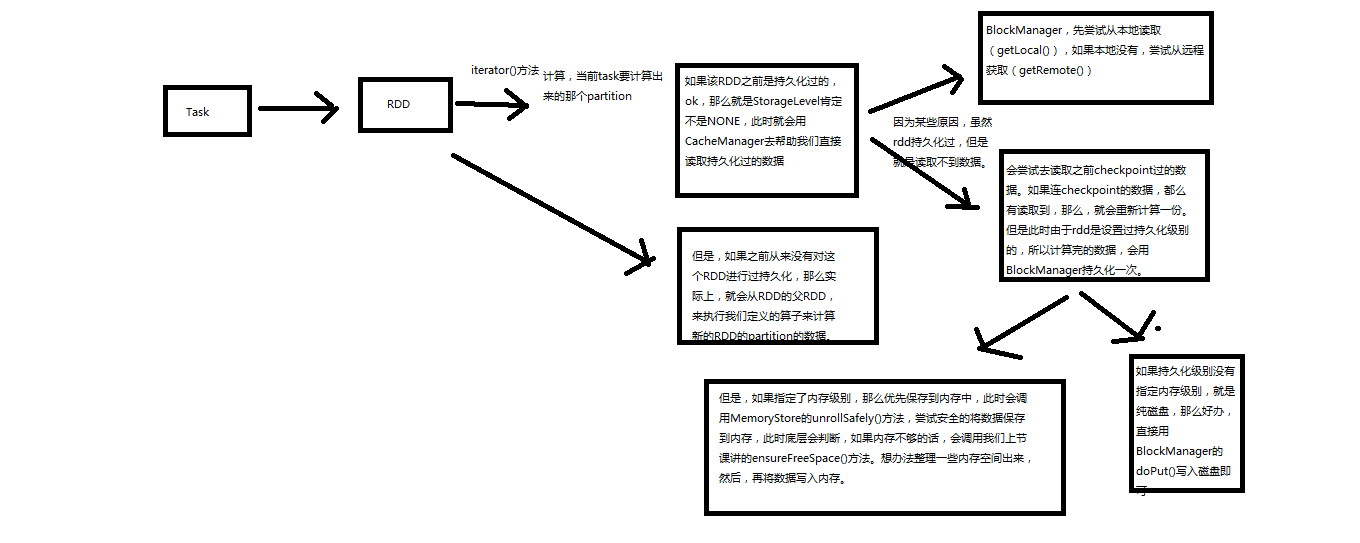
### 优化后的Shuffle原理剖析



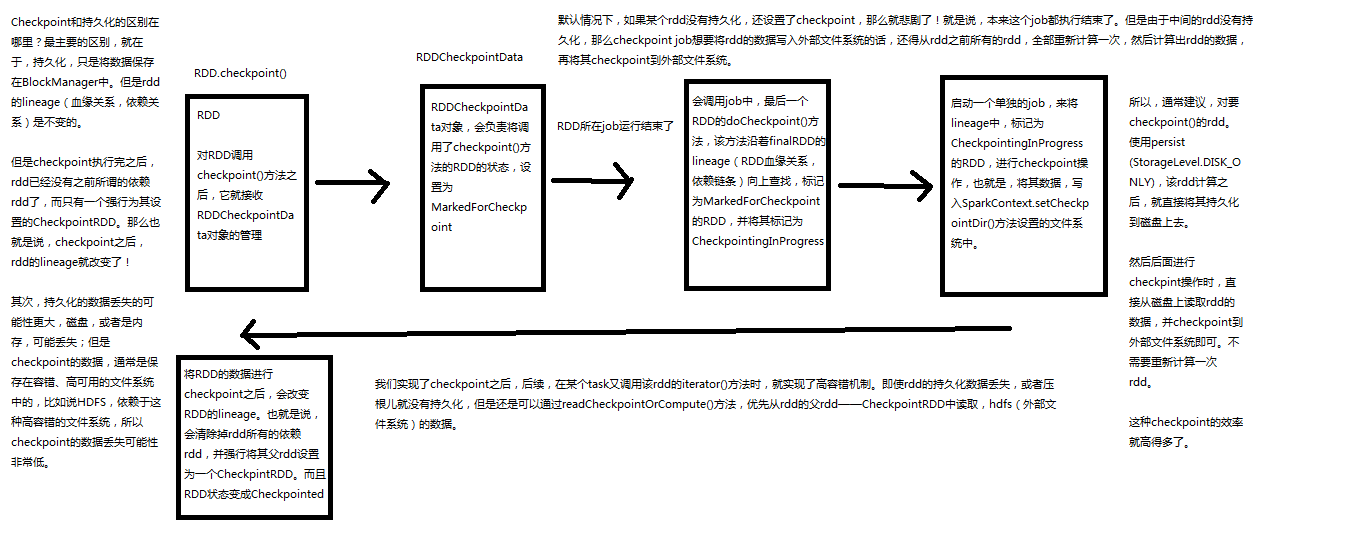
## BlockManager原理剖析与源码分析



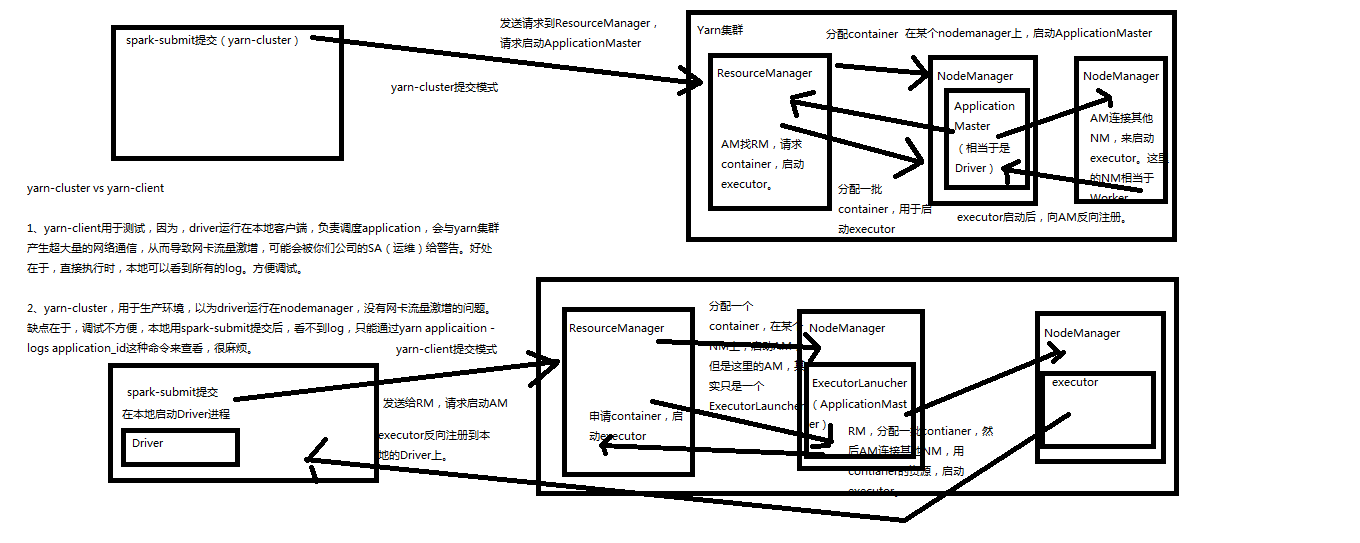
## CacheManager原理剖析与源码分析



## Checkpoint原理剖析



## 基于Yarn的两种提交模式深度剖析



1. Spark-submit提交程序
2. 发送请求到ResourceManager,请求启动ApplicationMaster
3. ResourceManager接收请求，分配container
4. 在某个nodemanager上启动ApplicationMaster（yarn-cluster），ApplicationMaster相当于driver。如果是yarn-client模式，driver在本地，在NM启动的为ExecutorLanucher，相当于ApplicationMaster。
5. AM找RM请求container，启动executer
6. RM分配一批container，AM连接其他NM来启动executer。NM相当于worker。
7. Executer启动后向AM反向注册。

Yarn-client用于测试，因为driver运行在本地，负责调度Application，会与yarn集群产生超大量的网络通信，从而导致网卡流量激增！

# Spark性能优化

## 使用高性能序列化类库

### 内存都花费在哪里了？

1、每个Java对象，都有一个对象头，会占用16个字节，主要是包括了一些对象的元信息，比如指向它的类的指针。如果一个对象本身很小，比如就包括了一个int类型的field，那么它的对象头实际上比对象自己还要大。

2、Java的String对象，会比它内部的原始数据，要多出40个字节。因为它内部使用char数组来保存内部的字符序列的，并且还得保存诸如数组长度之类的信息。而且因为String使用的是UTF-16编码，所以每个字符会占用2个字节。比如，包含10个字符的String，会占用60个字节。

3、Java中的集合类型，比如HashMap和LinkedList，内部使用的是链表数据结构，所以对链表中的每一个数据，都使用了Entry对象来包装。Entry对象不光有对象头，还有指向下一个Entry的指针，通常占用8个字节。

1. 元素类型为原始数据类型（比如int）的集合，内部通常会使用原始数据类型的包装类型，比如Integer，来存储元素。

### 如何判断你的程序消耗了多少内存？

这里有一个非常简单的办法来判断，你的spark程序消耗了多少内存。

1、首先，自己设置RDD的并行度，有两种方式：要不然，在parallelize()、textFile()等方法中，传入第二个参数，设置RDD的task / partition的数量；要不然，用SparkConf.set()方法，设置一个参数，spark.default.parallelism，可以统一设置这个application所有RDD的partition数量。

2、其次，在程序中将RDD cache到内存中，调用RDD.cache()方法即可。

3、最后，观察Driver的log，你会发现类似于：“INFO BlockManagerMasterActor: Added rdd\_0\_1 in memory on mbk.local:50311 (size: 717.5 KB, free: 332.3 MB)”的日志信息。这就显示了每个partition占用了多少内存。

1. 将这个内存信息乘以partition数量，即可得出RDD的内存占用量。

### 数据序列化概述

在任何分布式系统中，序列化都是扮演着一个重要的角色的。如果使用的序列化技术，在执行序列化操作的时候很慢，或者是序列化后的数据还是很大，那么会让分布式应用程序的性能下降很多。所以，进行Spark性能优化的第一步，就是进行序列化的性能优化。

Spark自身默认就会在一些地方对数据进行序列化，比如Shuffle。还有就是，如果我们的算子函数使用到了外部的数据（比如Java内置类型，或者自定义类型），那么也需要让其可序列化。

而Spark自身对于序列化的便捷性和性能进行了一个取舍和权衡。默认，Spark倾向于序列化的便捷性，使用了Java自身提供的序列化机制——基于ObjectInputStream和ObjectOutputStream的序列化机制。因为这种方式是Java原生提供的，很方便使用。

但是问题是，Java序列化机制的性能并不高。序列化的速度相对较慢，而且序列化以后的数据，还是相对来说比较大，还是比较占用内存空间。因此，如果你的Spark应用程序对内存很敏感，那么，实际上默认的Java序列化机制并不是最好的选择。

### Spark提供的两种序列化机制

Spark实际上提供了两种序列化机制，它只是默认使用了第一种：

1、Java序列化机制：默认情况下，Spark使用Java自身的ObjectInputStream和ObjectOutputStream机制进行对象的序列化。只要你的类实现了Serializable接口，那么都是可以序列化的。而且Java序列化机制是提供了自定义序列化支持的，只要你实现Externalizable接口即可实现自己的更高性能的序列化算法。Java序列化机制的速度比较慢，而且序列化后的数据占用的内存空间比较大。

1. Kryo序列化机制：Spark也支持使用Kryo类库来进行序列化。Kryo序列化机制比Java序列化机制更快，而且序列化后的数据占用的空间更小，通常比Java序列化的数据占用的空间要小10倍。Kryo序列化机制之所以不是默认序列化机制的原因是，有些类型虽然实现了Seriralizable接口，但是它也不一定能够进行序列化；此外，如果你要得到最佳的性能，Kryo还要求你在Spark应用程序中，对所有你需要序列化的类型都进行注册。

#### 如何使用Kryo序列化机制（一）

如果要使用Kryo序列化机制，首先要用SparkConf设置一个参数，使用**new SparkConf().set("spark.serializer", "org.apache.spark.serializer.KryoSerializer")**即可，即将Spark的序列化器设置为KryoSerializer。这样，Spark在内部的一些操作，比如Shuffle，进行序列化时，就会使用Kryo类库进行高性能、快速、更低内存占用量的序列化了。

使用Kryo时，它要求是需要序列化的类，是要预先进行注册的，以获得最佳性能——如果不注册的话，那么Kryo必须时刻保存类型的全限定名，反而占用不少内存。Spark默认是对Scala中常用的类型自动注册了Kryo的，都在AllScalaRegistry类中。

但是，比如自己的算子中，使用了外部的自定义类型的对象，那么还是需要将其进行注册。

（实际上，下面的写法是错误的，因为counter不是共享的，所以累加的功能是无法实现的）

val counter = new Counter();

val numbers = sc.parallelize(Array(1, 2, 3, 4, 5))

numbers.foreach(num => counter.add(num));

#### 如何使用Kryo序列化机制（二）

如果要注册自定义的类型，那么就使用如下的代码，即可：

Scala版本：

val conf = new SparkConf().setMaster(...).setAppName(...)

conf.registerKryoClasses(Array(classOf[Counter] ))

val sc = new SparkContext(conf)

Java版本：

SparkConf conf = new SparkConf().setMaster(...).setAppName(...)

conf.registerKryoClasses(Counter.class)

JavaSparkContext sc = new JavaSparkContext(conf)

#### 优化Kryo类库的使用

1、优化缓存大小

如果注册的要序列化的自定义的类型，本身特别大，比如包含了超过100个field。那么就会导致要序列化的对象过大。此时就需要对Kryo本身进行优化。因为Kryo内部的缓存可能不够存放那么大的class对象。此时就需要调用SparkConf.set()方法，设置**spark.kryoserializer.buffer.mb**参数的值，将其调大。

默认情况下它的值是2，就是说最大能缓存2M的对象，然后进行序列化。可以在必要时将其调大。比如设置为10。

2、预先注册自定义类型

虽然不注册自定义类型，Kryo类库也能正常工作，但是那样的话，对于它要序列化的每个对象，都会保存一份它的全限定类名。此时反而会耗费大量内存。因此通常都建议预先注册号要序列化的自定义的类。

#### 在什么场景下使用Kryo序列化类库？

首先，这里讨论的都是Spark的一些普通的场景，一些特殊的场景，比如RDD的持久化，在后面会讲解。这里先不说。

那么，这里针对的Kryo序列化类库的使用场景，就是算子函数使用到了外部的大数据的情况。比如说吧，我们在外部定义了一个封装了应用所有配置的对象，比如自定义了一个MyConfiguration对象，里面包含了100m的数据。然后，在算子函数里面，使用到了这个外部的大对象。

此时呢，如果默认情况下，让Spark用java序列化机制来序列化这种外部的大对象，那么就会导致，序列化速度缓慢，并且序列化以后的数据还是比较大，比较占用内存空间。

因此，在这种情况下，比较适合，切换到Kryo序列化类库，来对外部的大对象进行序列化操作。一是，序列化速度会变快；二是，会减少序列化后的数据占用的内存空间。

### 优化数据结构

要减少内存的消耗，除了使用高效的序列化类库以外，还有一个很重要的事情，就是优化数据结构。从而避免Java语法特性中所导致的额外内存的开销，比如基于指针的Java数据结构，以及包装类型。

有一个关键的问题，就是优化什么数据结构？其实主要就是优化你的算子函数，内部使用到的局部数据，或者是算子函数外部的数据。都可以进行数据结构的优化。优化之后，都会减少其对内存的消耗和占用。

#### 如何优化数据结构

1、优先使用数组以及字符串，而不是集合类。也就是说，优先用array，而不是ArrayList、LinkedList、HashMap等集合。

比如，有个List<Integer> list = new ArrayList<Integer>()，将其替换为int[] arr = new int[]。这样的话，array既比List少了额外信息的存储开销，还能使用原始数据类型（int）来存储数据，比List中用Integer这种包装类型存储数据，要节省内存的多。

还比如，通常企业级应用中的做法是，对于HashMap、List这种数据，统一用String拼接成特殊格式的字符串，比如Map<Integer, Person> persons = new HashMap<Integer, Person>()。可以优化为，特殊的字符串格式：id:name,address|id:name,address...。

2、避免使用多层嵌套的对象结构。比如说，public class Teacher { private List<Student> students = new ArrayList<Student>() }。就是非常不好的例子。因为Teacher类的内部又嵌套了大量的小Student对象。

比如说，对于上述例子，也完全可以使用特殊的字符串来进行数据的存储。比如，用json字符串来存储数据，就是一个很好的选择。

{"teacherId": 1, "teacherName": "leo", students:[{"studentId": 1, "studentName": "tom"},{"studentId":2, "studentName":"marry"}]}

3、对于有些能够避免的场景，尽量使用int替代String。因为String虽然比ArrayList、HashMap等数据结构高效多了，占用内存量少多了，但是之前分析过，还是有额外信息的消耗。比如之前用String表示id，那么现在完全可以用数字类型的int，来进行替代。

这里提醒，在spark应用中，id就不要用常用的uuid了，因为无法转成int，就用自增的int类型的id即可。（sdfsdfdf-234242342-sdfsfsfdfd）

## 对多次使用的RDD进行持久化 / Checkpoint

如果程序中，对某一个RDD，基于它进行了多次transformation或者action操作。那么就非常有必要对其进行持久化操作，以避免对一个RDD反复进行计算。

此外，如果要保证在RDD的持久化数据可能丢失的情况下，还要保证高性能，那么可以对RDD进行Checkpoint操作。



## 使用序列化的持久化级别

除了对多次使用的RDD进行持久化操作之外，还可以进一步优化其性能。因为很有可能，RDD的数据是持久化到内存，或者磁盘中的。那么，此时，如果内存大小不是特别充足，完全可以使用序列化的持久化级别，比如MEMORY\_ONLY\_SER、MEMORY\_AND\_DISK\_SER等。使用RDD.persist(StorageLevel.MEMORY\_ONLY\_SER)这样的语法即可。

这样的话，将数据序列化之后，再持久化，可以大大减小对内存的消耗。此外，数据量小了之后，如果要写入磁盘，那么磁盘io性能消耗也比较小。

对RDD持久化序列化后，RDD的每个partition的数据，都是序列化为一个巨大的字节数组。这样，对于内存的消耗就小的多了。但是唯一的缺点就是，获取RDD数据时，需要对其进行反序列化，会增大其性能开销。

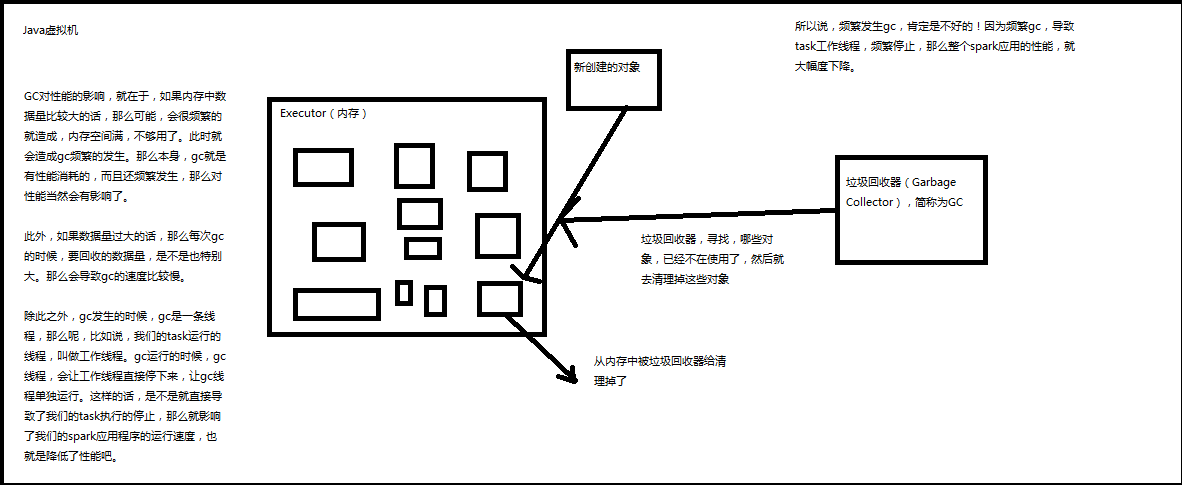
因此，对于序列化的持久化级别，还可以进一步优化，也就是说，使用Kryo序列化类库，这样，可以获得更快的序列化速度，并且占用更小的内存空间。但是要记住，如果RDD的元素（RDD<T>的泛型类型），是自定义类型的话，在Kryo中提前注册自定义类型。

## JVM调优

### JVM调优的背景

如果在持久化RDD的时候，持久化了大量的数据，那么Java虚拟机的垃圾回收就可能成为一个性能瓶颈。因为Java虚拟机会定期进行垃圾回收，此时就会追踪所有的java对象，并且在垃圾回收时，找到那些已经不在使用的对象，然后清理旧的对象，来给新的对象腾出内存空间。

垃圾回收的性能开销，是跟内存中的对象的数量，成正比的。所以，对于垃圾回收的性能问题，首先要做的就是，使用更高效的数据结构，比如array和string；其次就是在持久化rdd时，使用序列化的持久化级别，而且用Kryo序列化类库，这样，每个partition就只是一个对象——一个字节数组。



### 监测垃圾回收

我们可以对垃圾回收进行监测，包括多久进行一次垃圾回收，以及每次垃圾回收耗费的时间。只要在spark-submit脚本中，增加一个配置即可，--conf "spark.executor.extraJavaOptions=-verbose:gc -XX:+PrintGCDetails -XX:+PrintGCTimeStamps"。

但是要记住，这里虽然会打印出Java虚拟机的垃圾回收的相关信息，但是是输出到了worker上的日志中，而不是driver的日志中。

但是这种方式也只是一种，其实也完全可以通过SparkUI（4040端口）来观察每个stage的垃圾回收的情况。

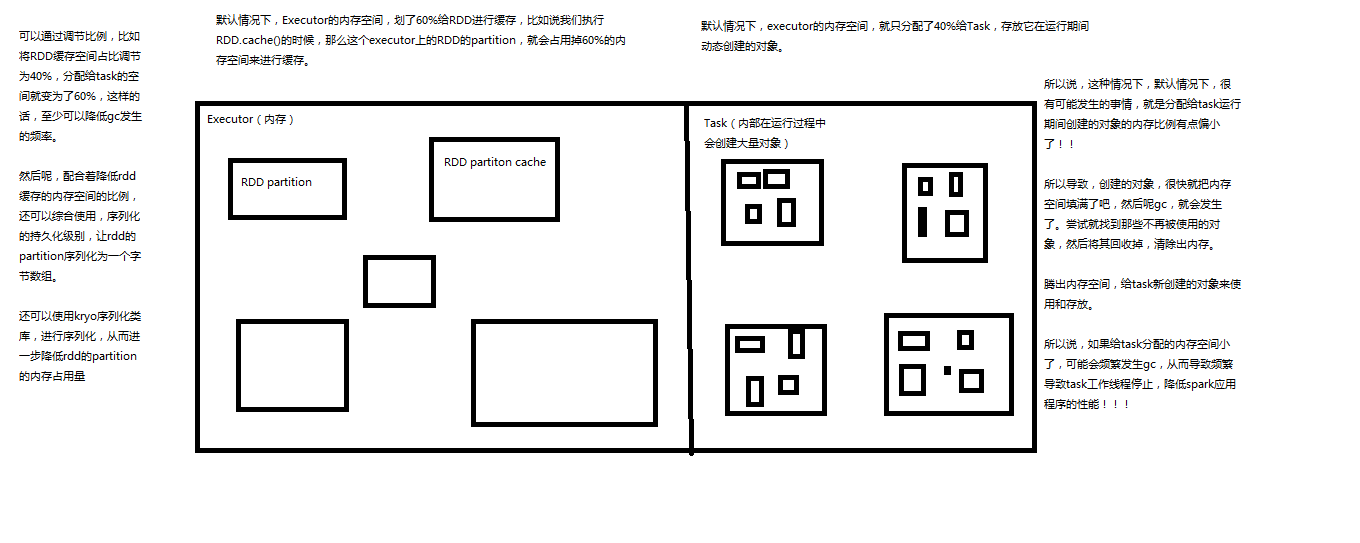
### 优化executor内存比例

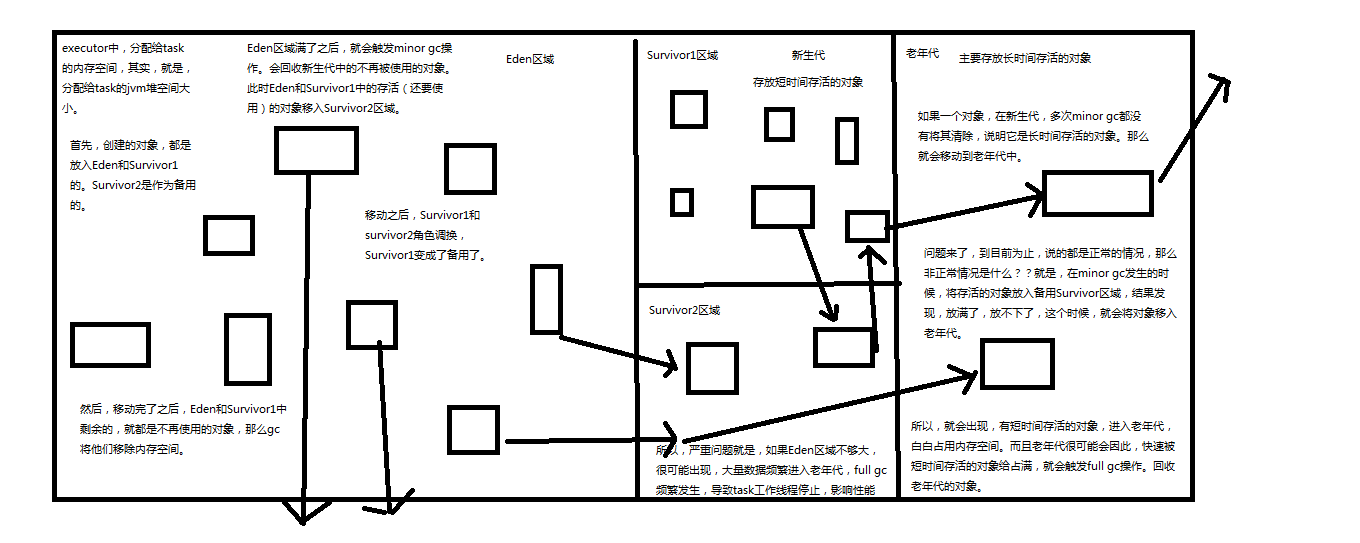
对于垃圾回收来说，最重要的就是调节RDD缓存占用的内存空间，与算子执行时创建的对象占用的内存空间的比例。默认情况下，Spark使用每个executor 60%的内存空间来缓存RDD，那么在task执行期间创建的对象，只有40%的内存空间来存放。

在这种情况下，很有可能因为你的内存空间的不足，task创建的对象过大，那么一旦发现40%的内存空间不够用了，就会触发Java虚拟机的垃圾回收操作。因此在极端情况下，垃圾回收操作可能会被频繁触发。

在上述情况下，如果发现垃圾回收频繁发生。那么就需要对那个比例进行调优，使用new SparkConf().set("spark.storage.memoryFraction", "0.5")即可，可以将RDD缓存占用空间的比例降低，从而给更多的空间让task创建的对象进行使用。

因此，对于RDD持久化，完全可以使用Kryo序列化，加上降低其executor内存占比的方式，来减少其内存消耗。给task提供更多的内存，从而避免task的执行频繁触发垃圾回收。





### 高级垃圾回收调优

Java堆空间被划分成了两块空间，一个是年轻代，一个是老年代。年轻代放的是短时间存活的对象，老年代放的是长时间存活的对象。年轻代又被划分了三块空间，Eden、Survivor1、Survivor2。

首先，Eden区域和Survivor1区域用于存放对象，Survivor2区域备用。创建的对象，首先放入Eden区域和Survivor1区域，如果Eden区域满了，那么就会触发一次Minor GC，进行年轻代的垃圾回收。Eden和Survivor1区域中存活的对象，会被移动到Survivor2区域中。然后Survivor1和Survivor2的角色调换，Survivor1变成了备用。

如果一个对象，在年轻代中，撑过了多次垃圾回收，都没有被回收掉，那么会被认为是长时间存活的，此时就会被移入老年代。此外，如果在将Eden和Survivor1中的存活对象，尝试放入Survivor2中时，发现Survivor2放满了，那么会直接放入老年代。此时就出现了，短时间存活的对象，进入老年代的问题。

如果老年代的空间满了，那么就会触发Full GC，进行老年代的垃圾回收操作。

Spark中，垃圾回收调优的目标就是，只有真正长时间存活的对象，才能进入老年代，短时间存活的对象，只能呆在年轻代。不能因为某个Survivor区域空间不够，在Minor GC时，就进入了老年代。从而造成短时间存活的对象，长期呆在老年代中占据了空间，而且Full GC时要回收大量的短时间存活的对象，导致Full GC速度缓慢。

如果发现，在task执行期间，大量full gc发生了，那么说明，年轻代的Eden区域，给的空间不够大。此时可以执行一些操作来优化垃圾回收行为：

1、包括降低spark.storage.memoryFraction的比例，给年轻代更多的空间，来存放短时间存活的对象；

2、给Eden区域分配更大的空间，使用-Xmn即可，通常建议给Eden区域，预计大小的4/3；

3、如果使用的是HDFS文件，那么很好估计Eden区域大小，如果每个executor有4个task，然后每个hdfs压缩块解压缩后大小是3倍，此外每个hdfs块的大小是64M，那么Eden区域的预计大小就是：4 \* 3 \* 64MB，然后呢，再通过-Xmn参数，将Eden区域大小设置为4 \* 3 \* 64 \* 4/3。

### 小结

对于垃圾回收的调优，尽量就是说，调节executor内存的比例就可以了。因为jvm的调优是非常复杂和敏感的。除非是，真的到了万不得已的地方，然后呢，自己本身又对jvm相关的技术很了解，那么此时进行eden区域的调节，调优，是可以的。

一些高级的参数

-XX:SurvivorRatio=4：如果值为4，那么就是两个Survivor跟Eden的比例是2:4，也就是说每个Survivor占据的年轻代的比例是1/6，所以，你其实也可以尝试调大Survivor区域的大小。

-XX:NewRatio=4：调节新生代和老年代的比例。

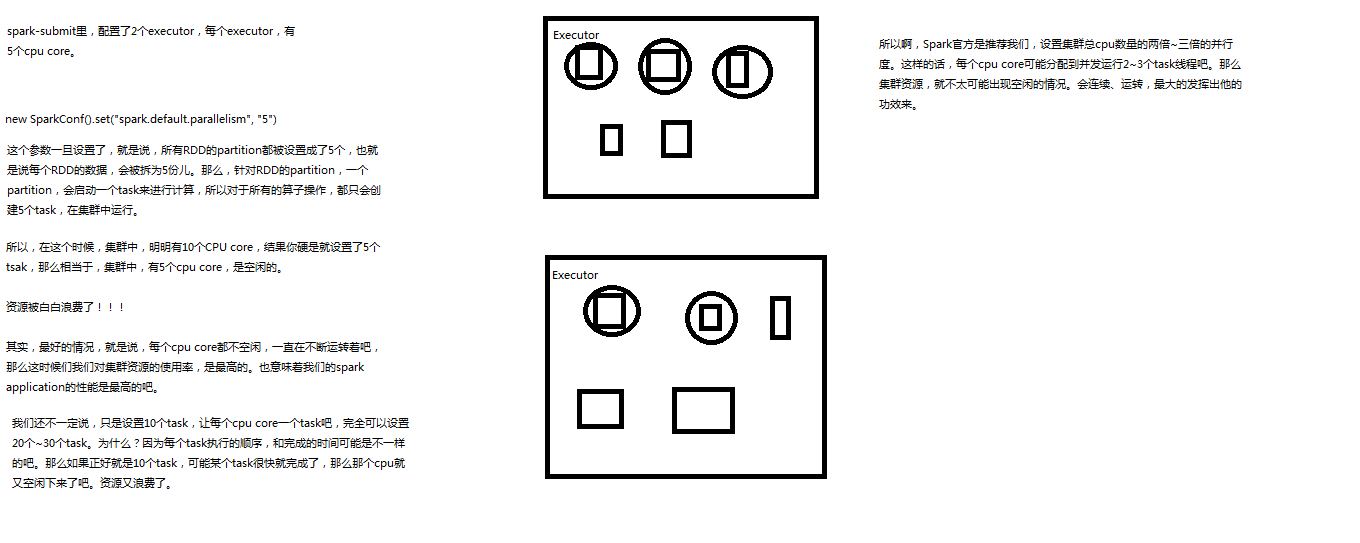
## 提高并行度

实际上Spark集群的资源并不一定会被充分利用到，所以要尽量设置合理的并行度，来充分地利用集群的资源。才能充分提高Spark应用程序的性能。

Spark会自动设置以文件作为输入源的RDD的并行度，依据其大小，比如HDFS，就会给每一个block创建一个partition，也依据这个设置并行度。对于reduceByKey等会发生shuffle的操作，就使用并行度最大的父RDD的并行度即可。

可以手动使用textFile()、parallelize()等方法的第二个参数来设置并行度；也可以使用spark.default.parallelism参数，来设置统一的并行度。Spark官方的推荐是，给集群中的每个cpu core设置2~3个task。

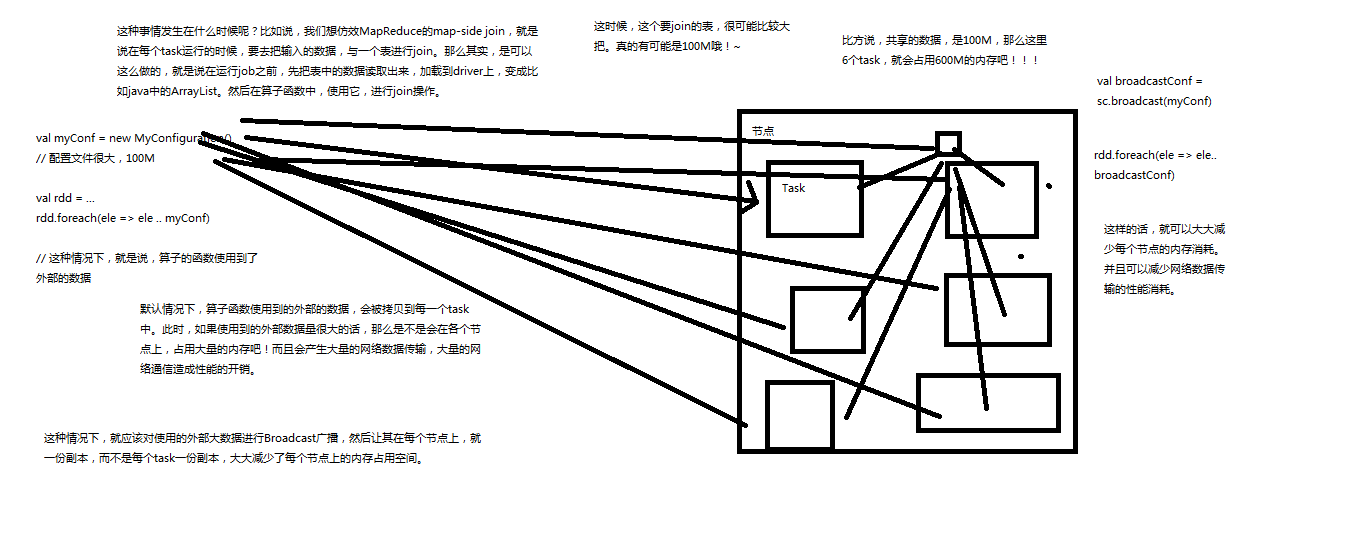
比如说，spark-submit设置了executor数量是10个，每个executor要求分配2个core，那么application总共会有20个core。此时可以设置new SparkConf().set("spark.default.parallelism", "60")来设置合理的并行度，从而充分利用资源。



## 广播共享数据

如果你的算子函数中，使用到了特别大的数据，那么，这个时候，推荐将该数据进行广播。这样的话，就不至于将一个大数据拷贝到每一个task上去。而是给每个节点拷贝一份，然后节点上的task共享该数据。

这样的话，就可以减少大数据在节点上的内存消耗。并且可以减少数据到节点的网络传输消耗。



## 数据本地化

数据本地化对于Spark Job性能有着巨大的影响。如果数据以及要计算它的代码是在一起的，那么性能当然会非常高。但是，如果数据和计算它的代码是分开的，那么其中之一必须到另外一方的机器上。通常来说，移动代码到其他节点，会比移动数据到代码所在的节点上去，速度要快得多，因为代码比较小。Spark也正是基于这个数据本地化的原则来构建task调度算法的。

数据本地化，指的是，数据离计算它的代码有多近。基于数据距离代码的距离，有几种数据本地化级别：

1、PROCESS\_LOCAL：数据和计算它的代码在同一个JVM进程中。

2、NODE\_LOCAL：数据和计算它的代码在一个节点上，但是不在一个进程中，比如在不同的executor进程中，或者是数据在HDFS文件的block中。

3、NO\_PREF：数据从哪里过来，性能都是一样的。

4、RACK\_LOCAL：数据和计算它的代码在一个机架上。

5、ANY：数据可能在任意地方，比如其他网络环境内，或者其他机架上。

Spark倾向于使用最好的本地化级别来调度task，但是这是不可能的。如果没有任何未处理的数据在空闲的executor上，那么Spark就会放低本地化级别。这时有两个选择：第一，等待，直到executor上的cpu释放出来，那么就分配task过去；第二，立即在任意一个executor上启动一个task。

## reduceByKey和groupByKey的合理使用

Spark默认会等待一会儿，来期望task要处理的数据所在的节点上的executor空闲出一个cpu，从而将task分配过去。只要超过了时间，那么Spark就会将task分配到其他任意一个空闲的executor上。

可以设置参数，spark.locality系列参数，来调节Spark等待task可以进行数据本地化的时间。spark.locality.wait（3000毫秒）、spark.locality.wait.node、spark.locality.wait.process、spark.locality.wait.rack。

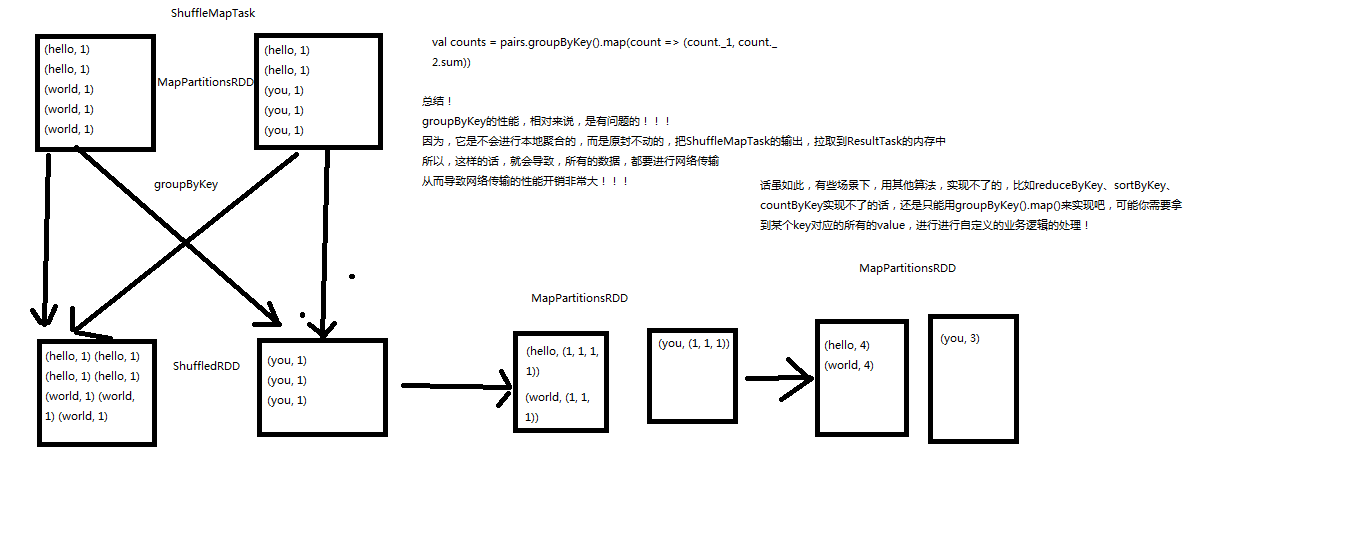
### reduceByKey和groupBykey

val counts = pairs.reduceByKey(\_ + \_)

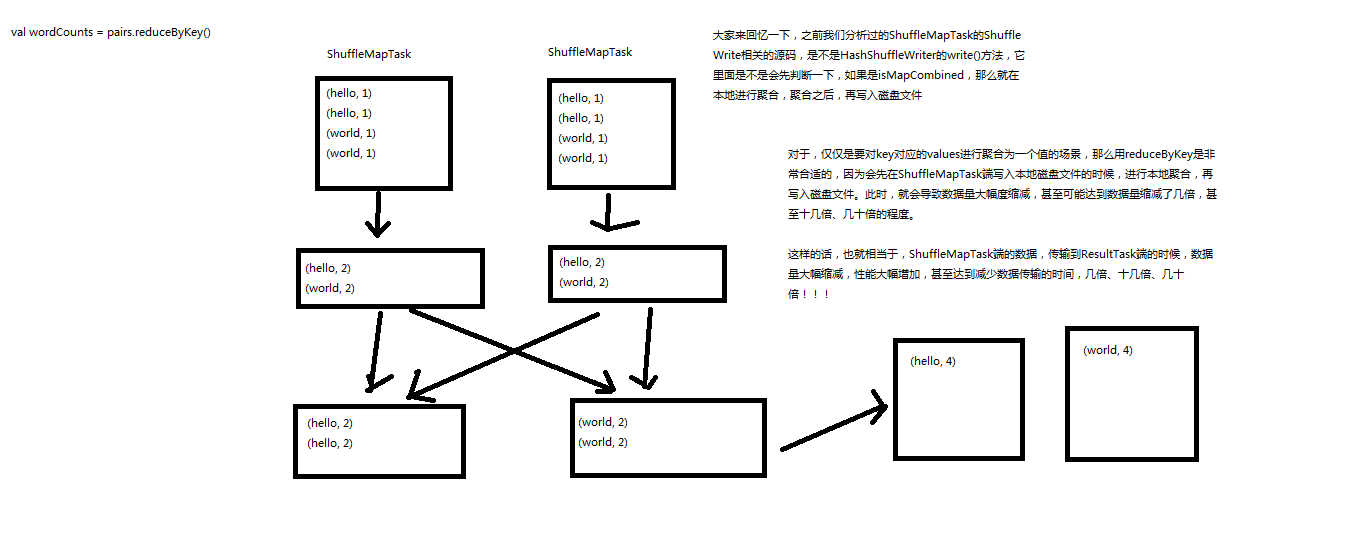
val counts = pairs.groupByKey().map(wordCounts => (wordCounts.\_1, wordCounts.\_2.sum))

如果能用reduceByKey，那就用reduceByKey，因为它会在map端，先进行本地combine，可以大大减少要传输到reduce端的数据量，减小网络传输的开销。

只有在reduceByKey处理不了时，才用groupByKey().map()来替代。



groupByKey原理



reduceByKey原理

## Shuffle调优（核心中的核心，重中之重）

1. 在Spark中，什么情况下，会发生shuffle？reduceByKey、groupByKey、sortByKey、countByKey、join、cogroup等操作。

第一个特点，

在Spark早期版本中，那个bucket缓存是非常非常重要的，因为需要将一个ShuffleMapTask所有的数据都写入内存缓存之后，才会刷新到磁盘。但是这就有一个问题，如果map side数据过多，那么很容易造成内存溢出。所以spark在新版本中，优化了，默认那个内存缓存是100kb，然后呢，写入一点数据达到了刷新到磁盘的阈值之后，就会将数据一点一点地刷新到磁盘。

这种操作的优点，是不容易发生内存溢出。缺点在于，如果内存缓存过小的话，那么可能发生过多的磁盘写io操作。所以，这里的内存缓存大小，是可以根据实际的业务情况进行优化的。

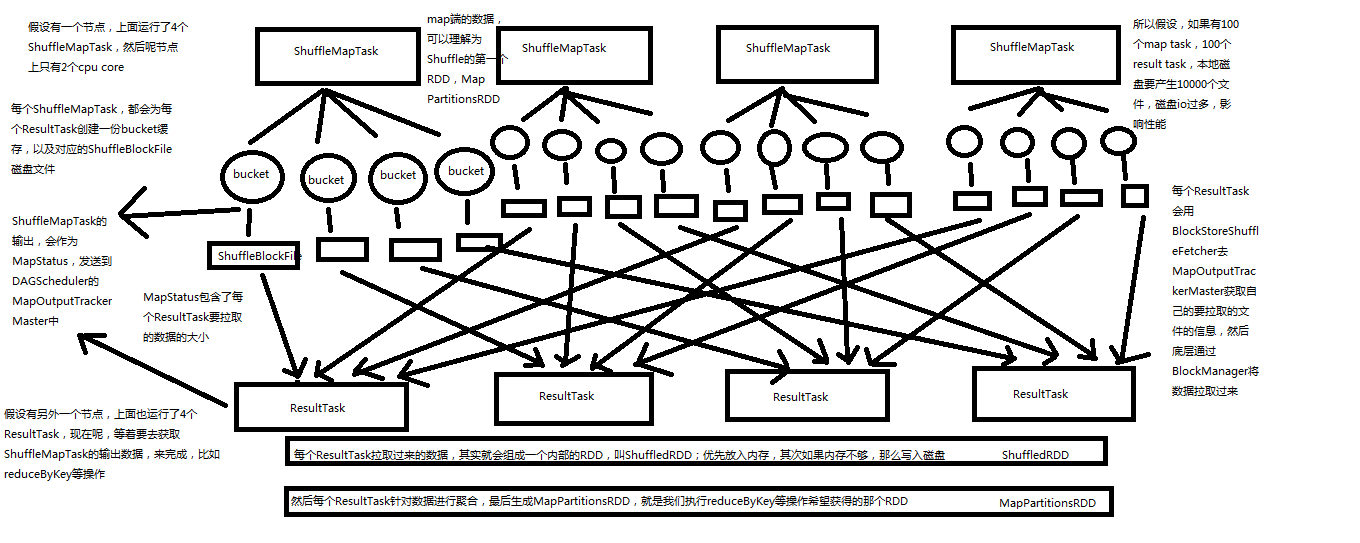
第二个特点，

与MapReduce完全不一样的是，MapReduce它必须将所有的数据都写入本地磁盘文件以后，才能启动reduce操作，来拉取数据。为什么？因为mapreduce要实现默认的根据key的排序！所以要排序，肯定得写完所有数据，才能排序，然后reduce来拉取。

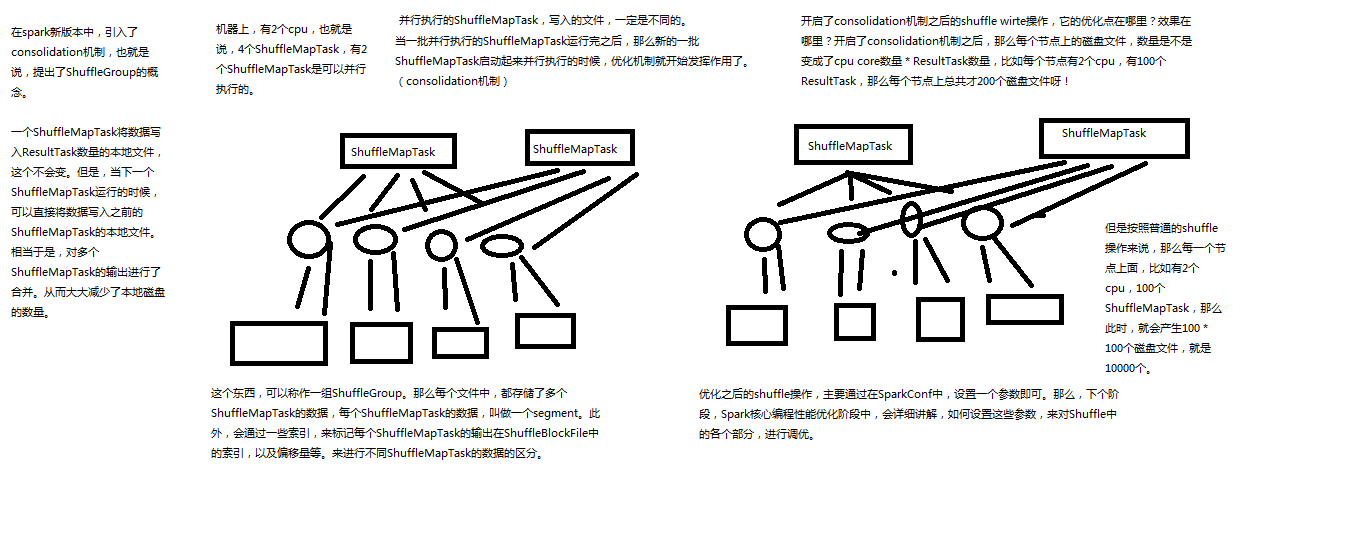
但是Spark不需要，spark默认情况下，是不会对数据进行排序的。因此ShuffleMapTask每写入一点数据，ResultTask就可以拉取一点数据，然后在本地执行我们定义的聚合函数和算子，进行计算。

spark这种机制的好处在于，速度比mapreduce快多了。但是也有一个问题，mapreduce提供的reduce，是可以处理每个key对应的value上的，很方便。但是spark中，由于这种实时拉取的机制，因此提供不了，直接处理key对应的values的算子，只能通过groupByKey，先shuffle，有一个MapPartitionsRDD，然后用map算子，来处理每个key对应的values。就没有mapreduce的计算模型那么方便。

1. 默认的Shuffle操作的原理剖析



1. 优化后的Shuffle操作的原理剖析



1. 基于排序的shuffle操作的原理剖析

### shuffle性能优化

new SparkConf().set("spark.shuffle.consolidateFiles", "true")

spark.shuffle.consolidateFiles：是否开启shuffle block file的合并，默认为false

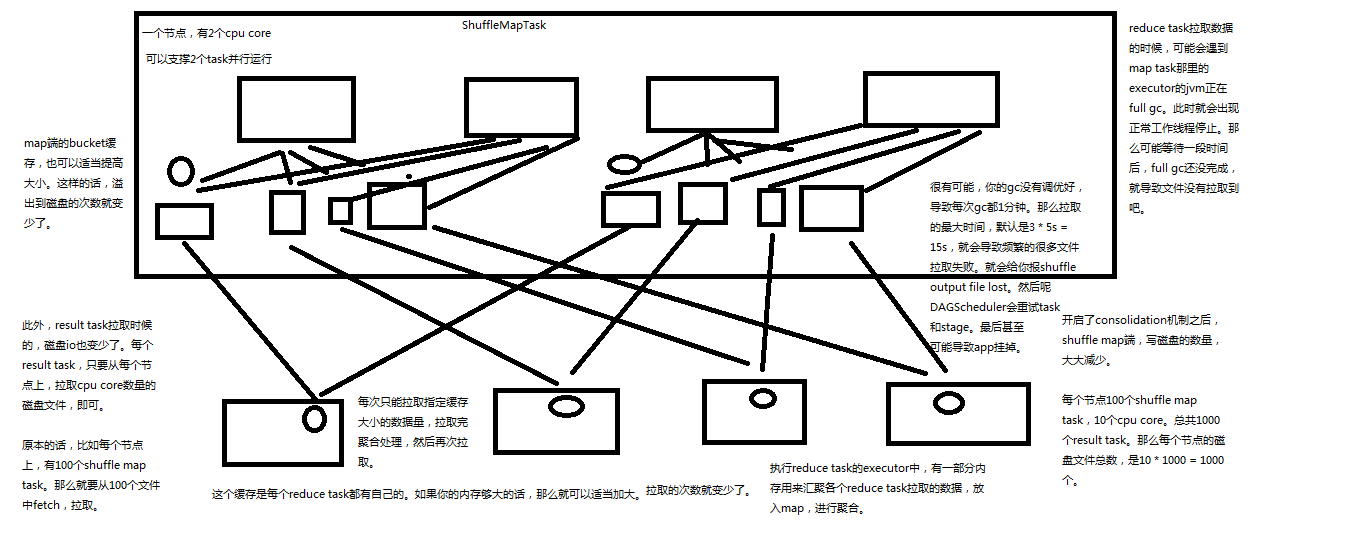
spark.reducer.maxSizeInFlight：reduce task的拉取缓存，默认48m

spark.shuffle.file.buffer：map task的写磁盘缓存，默认32k

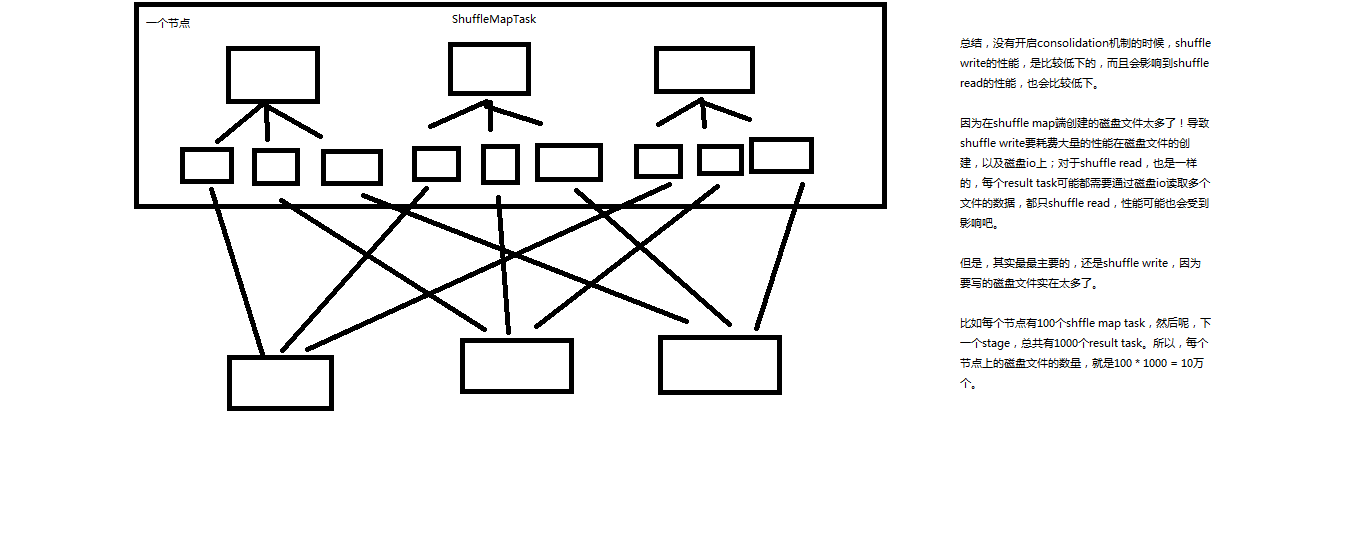
spark.shuffle.io.maxRetries：拉取失败的最大重试次数，默认3次

spark.shuffle.io.retryWait：拉取失败的重试间隔，默认5s

spark.shuffle.memoryFraction：用于reduce端聚合的内存比例，默认0.2，超过比例就会溢出到磁盘上



开启consolidation机制之后对磁盘io性能的提升的原理



没有开启consolidation机制的性能低下的原理剖析

## Spark SQL的调优

1、设置Shuffle过程中的并行度：spark.sql.shuffle.partitions（SQLContext.setConf()）

2、在Hive数据仓库建设过程中，合理设置数据类型，比如能设置为INT的，就不要设置为BIGINT。减少数据类型导致的不必要的内存开销。

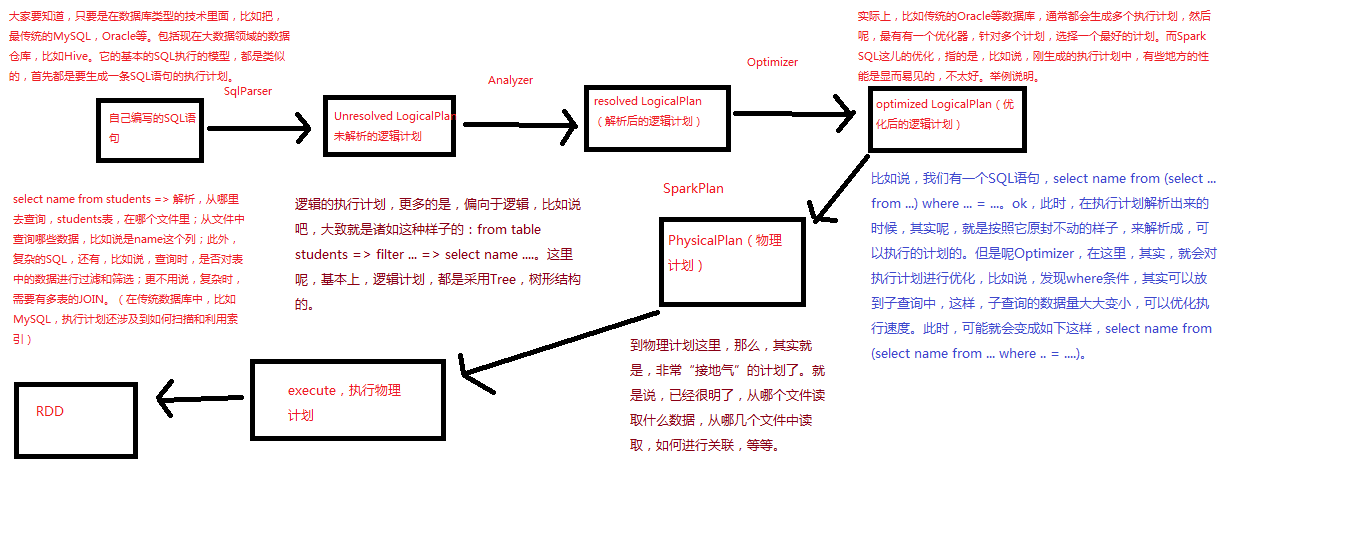
3、编写SQL时，尽量给出明确的列名，比如select name from students。不要写select \*的方式。

4、并行处理查询结果：对于Spark SQL查询的结果，如果数据量比较大，比如超过1000条，那么就不要一次性collect()到Driver再处理。使用foreach()算子，并行处理查询结果。

5、缓存表：对于一条SQL语句中可能多次使用到的表，可以对其进行缓存，使用SQLContext.cacheTable(tableName)，或者DataFrame.cache()即可。Spark SQL会用内存列存储的格式进行表的缓存。然后Spark SQL就可以仅仅扫描需要使用的列，并且自动优化压缩，来最小化内存使用和GC开销。SQLContext.uncacheTable(tableName)可以将表从缓存中移除。用SQLContext.setConf()，设置spark.sql.inMemoryColumnarStorage.batchSize参数（默认10000），可以配置列存储的单位。

6、广播join表：spark.sql.autoBroadcastJoinThreshold，默认10485760 (10 MB)。在内存够用的情况下，可以增加其大小，概参数设置了一个表在join的时候，最大在多大以内，可以被广播出去优化性能。

7、钨丝计划：spark.sql.tungsten.enabled，默认是true，自动管理内存。



Spark sql 是如何把sql解析成rdd执行的

## Spark Streaming的调优方式

### 数据接收并行度调优

通过网络接收数据时（比如Kafka、Flume），会将数据反序列化，并存储在Spark的内存中。如果数据接收称为系统的瓶颈，那么可以考虑并行化数据接收。每一个输入DStream都会在某个Worker的Executor上启动一个Receiver，该Receiver接收一个数据流。因此可以通过创建多个输入DStream，并且配置它们接收数据源不同的分区数据，达到接收多个数据流的效果。比如说，一个接收两个Kafka Topic的输入DStream，可以被拆分为两个输入DStream，每个分别接收一个topic的数据。这样就会创建两个Receiver，从而并行地接收数据，进而提升吞吐量。多个DStream可以使用union算子进行聚合，从而形成一个DStream。然后后续的transformation算子操作都针对该一个聚合后的DStream即可。

int numStreams = 5;

List<JavaPairDStream<String, String>> kafkaStreams = new ArrayList<JavaPairDStream<String, String>>(numStreams);

for (int i = 0; i < numStreams; i++) {

kafkaStreams.add(KafkaUtils.createStream(...));

}

JavaPairDStream<String, String> unifiedStream = streamingContext.union(kafkaStreams.get(0), kafkaStreams.subList(1, kafkaStreams.size()));

unifiedStream.print();

数据接收并行度调优，除了创建更多输入DStream和Receiver以外，还可以考虑调节block interval。通过参数，spark.streaming.blockInterval，可以设置block interval，默认是200ms。对于大多数Receiver来说，在将接收到的数据保存到Spark的BlockManager之前，都会将数据切分为一个一个的block。而每个batch中的block数量，则决定了该batch对应的RDD的partition的数量，以及针对该RDD执行transformation操作时，创建的task的数量。每个batch对应的task数量是大约估计的，即batch interval / block interval。

例如说，batch interval为2s，block interval为200ms，会创建10个task。如果你认为每个batch的task数量太少，即低于每台机器的cpu core数量，那么就说明batch的task数量是不够的，因为所有的cpu资源无法完全被利用起来。要为batch增加block的数量，那么就减小block interval。然而，推荐的block interval最小值是50ms，如果低于这个数值，那么大量task的启动时间，可能会变成一个性能开销点。

除了上述说的两个提升数据接收并行度的方式，还有一种方法，就是显式地对输入数据流进行重分区。使用inputStream.repartition(<number of partitions>)即可。这样就可以将接收到的batch，分布到指定数量的机器上，然后再进行进一步的操作。

### 任务启动调优

如果每秒钟启动的task过于多，比如每秒钟启动50个，那么发送这些task去Worker节点上的Executor的性能开销，会比较大，而且此时基本就很难达到毫秒级的延迟了。使用下述操作可以减少这方面的性能开销：

1、Task序列化：使用Kryo序列化机制来序列化task，可以减小task的大小，从而减少发送这些task到各个Worker节点上的Executor的时间。

2、执行模式：在Standalone模式下运行Spark，可以达到更少的task启动时间。

上述方式，也许可以将每个batch的处理时间减少100毫秒。从而从秒级降到毫秒级。

### 数据处理并行度调优

如果在计算的任何stage中使用的并行task的数量没有足够多，那么集群资源是无法被充分利用的。举例来说，对于分布式的reduce操作，比如reduceByKey和reduceByKeyAndWindow，默认的并行task的数量是由spark.default.parallelism参数决定的。你可以在reduceByKey等操作中，传入第二个参数，手动指定该操作的并行度，也可以调节全局的spark.default.parallelism参数。

### 数据序列化调优

数据序列化造成的系统开销可以由序列化格式的优化来减小。在流式计算的场景下，有两种类型的数据需要序列化。

1、输入数据：默认情况下，接收到的输入数据，是存储在Executor的内存中的，使用的持久化级别是StorageLevel.MEMORY\_AND\_DISK\_SER\_2。这意味着，数据被序列化为字节从而减小GC开销，并且会复制以进行executor失败的容错。因此，数据首先会存储在内存中，然后在内存不足时会溢写到磁盘上，从而为流式计算来保存所有需要的数据。这里的序列化有明显的性能开销——Receiver必须反序列化从网络接收到的数据，然后再使用Spark的序列化格式序列化数据。

1. 流式计算操作生成的持久化RDD：流式计算操作生成的持久化RDD，可能会持久化到内存中。例如，窗口操作默认就会将数据持久化在内存中，因为这些数据后面可能会在多个窗口中被使用，并被处理多次。然而，不像Spark Core的默认持久化级别，StorageLevel.MEMORY\_ONLY，流式计算操作生成的RDD的默认持久化级别是StorageLevel.MEMORY\_ONLY\_SER ，默认就会减小GC开销。

在上述的场景中，使用Kryo序列化类库可以减小CPU和内存的性能开销。使用Kryo时，一定要考虑注册自定义的类，并且禁用对应引用的tracking（spark.kryo.referenceTracking）。

在一些特殊的场景中，比如需要为流式应用保持的数据总量并不是很多，也许可以将数据以非序列化的方式进行持久化，从而减少序列化和反序列化的CPU开销，而且又不会有太昂贵的GC开销。举例来说，如果你数秒的batch interval，并且没有使用window操作，那么你可以考虑通过显式地设置持久化级别，来禁止持久化时对数据进行序列化。这样就可以减少用于序列化和反序列化的CPU性能开销，并且不用承担太多的GC开销。

### batch interval调优（最重要）

如果想让一个运行在集群上的Spark Streaming应用程序可以稳定，它就必须尽可能快地处理接收到的数据。换句话说，batch应该在生成之后，就尽可能快地处理掉。对于一个应用来说，这个是不是一个问题，可以通过观察Spark UI上的batch处理时间来定。batch处理时间必须小于batch interval时间。

基于流式计算的本质，batch interval对于，在固定集群资源条件下，应用能保持的数据接收速率，会有巨大的影响。例如，在WordCount例子中，对于一个特定的数据接收速率，应用业务可以保证每2秒打印一次单词计数，而不是每500ms。因此batch interval需要被设置得，让预期的数据接收速率可以在生产环境中保持住。

为你的应用计算正确的batch大小的比较好的方法，是在一个很保守的batch interval，比如5~10s，以很慢的数据接收速率进行测试。要检查应用是否跟得上这个数据速率，可以检查每个batch的处理时间的延迟，如果处理时间与batch interval基本吻合，那么应用就是稳定的。否则，如果batch调度的延迟持续增长，那么就意味应用无法跟得上这个速率，也就是不稳定的。因此你要想有一个稳定的配置，可以尝试提升数据处理的速度，或者增加batch interval。记住，由于临时性的数据增长导致的暂时的延迟增长，可以合理的，只要延迟情况可以在短时间内恢复即可。

### 内存调优

优化Spark应用的内存使用和GC行为，在Spark Core的调优中，已经讲过了。这里讲一下与Spark Streaming应用相关的调优参数。

Spark Streaming应用需要的集群内存资源，是由使用的transformation操作类型决定的。举例来说，如果想要使用一个窗口长度为10分钟的window操作，那么集群就必须有足够的内存来保存10分钟内的数据。如果想要使用updateStateByKey来维护许多key的state，那么你的内存资源就必须足够大。反过来说，如果想要做一个简单的map-filter-store操作，那么需要使用的内存就很少。

通常来说，通过Receiver接收到的数据，会使用StorageLevel.MEMORY\_AND\_DISK\_SER\_2持久化级别来进行存储，因此无法保存在内存中的数据会溢写到磁盘上。而溢写到磁盘上，是会降低应用的性能的。因此，通常是建议为应用提供它需要的足够的内存资源。建议在一个小规模的场景下测试内存的使用量，并进行评估。

内存调优的另外一个方面是垃圾回收。对于流式应用来说，如果要获得低延迟，肯定不想要有因为JVM垃圾回收导致的长时间延迟。有很多参数可以帮助降低内存使用和GC开销：

1、DStream的持久化：正如在“数据序列化调优”一节中提到的，输入数据和某些操作生产的中间RDD，默认持久化时都会序列化为字节。与非序列化的方式相比，这会降低内存和GC开销。使用Kryo序列化机制可以进一步减少内存使用和GC开销。进一步降低内存使用率，可以对数据进行压缩，由spark.rdd.compress参数控制（默认false）。

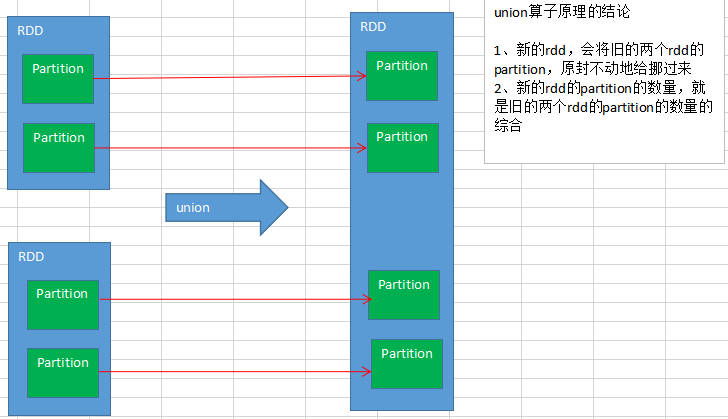
2、清理旧数据：默认情况下，所有输入数据和通过DStream transformation操作生成的持久化RDD，会自动被清理。Spark Streaming会决定何时清理这些数据，取决于transformation操作类型。例如，你在使用窗口长度为10分钟内的window操作，Spark会保持10分钟以内的数据，时间过了以后就会清理旧数据。但是在某些特殊场景下，比如Spark SQL和Spark Streaming整合使用时，在异步开启的线程中，使用Spark SQL针对batch RDD进行执行查询。那么就需要让Spark保存更长时间的数据，直到Spark SQL查询结束。可以使用streamingContext.remember()方法来实现。

3、CMS垃圾回收器：使用并行的mark-sweep垃圾回收机制，被推荐使用，用来保持GC低开销。虽然并行的GC会降低吞吐量，但是还是建议使用它，来减少batch的处理时间（降低处理过程中的gc开销）。如果要使用，那么要在driver端和executor端都开启。在spark-submit中使用--driver-java-options设置；使用spark.executor.extraJavaOptions参数设置。-XX:+UseConcMarkSweepGC。

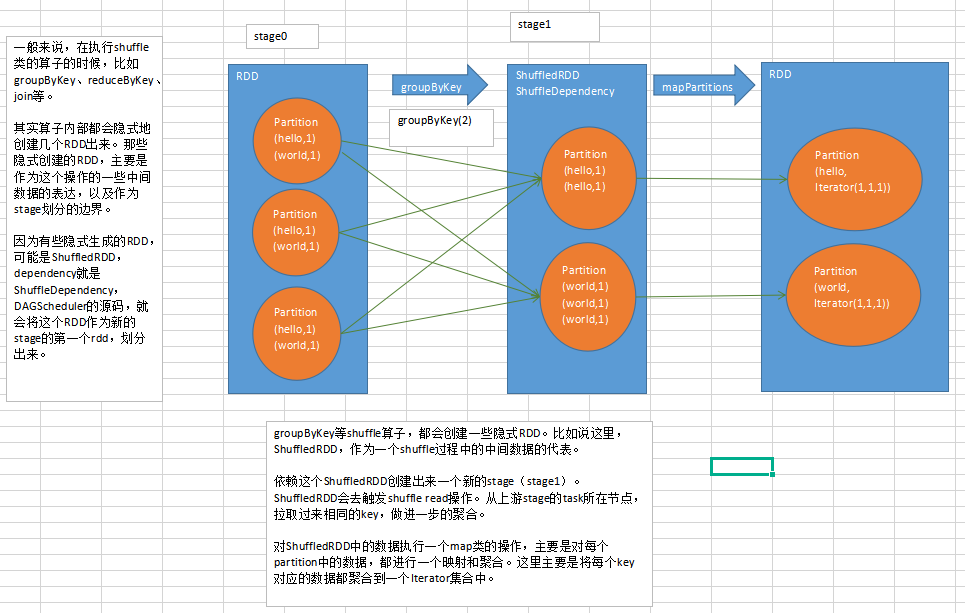
# Spark算子详解

## Spark core

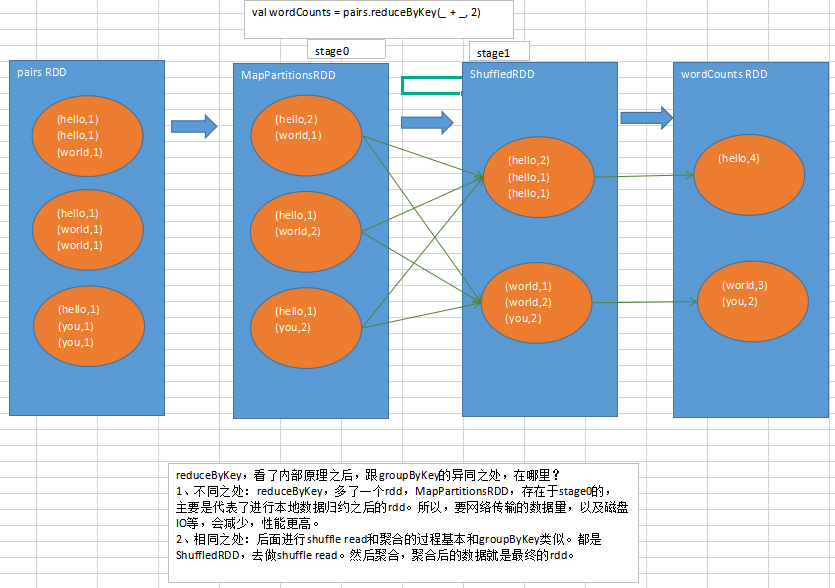
### union算子内部实现原理剖析



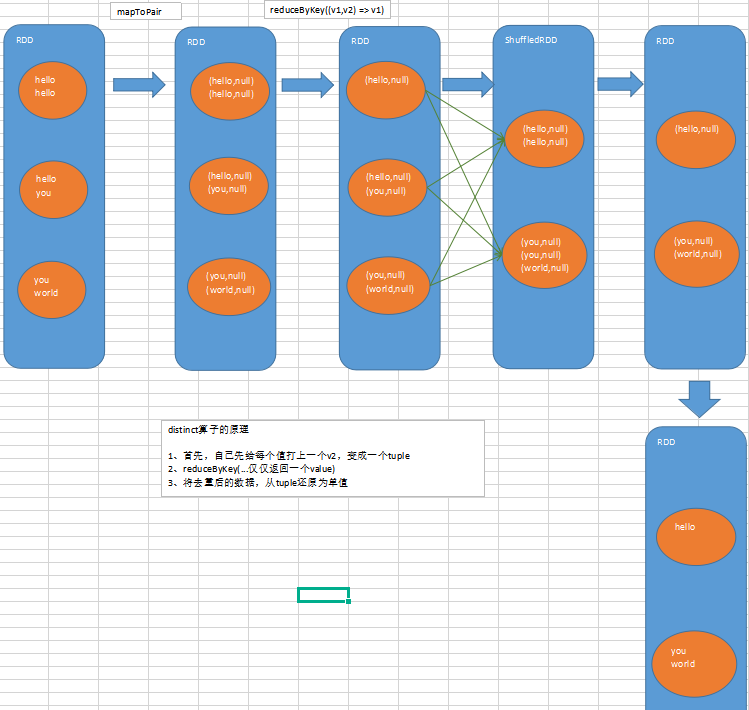
### groupByKey算子内部实现原理剖析



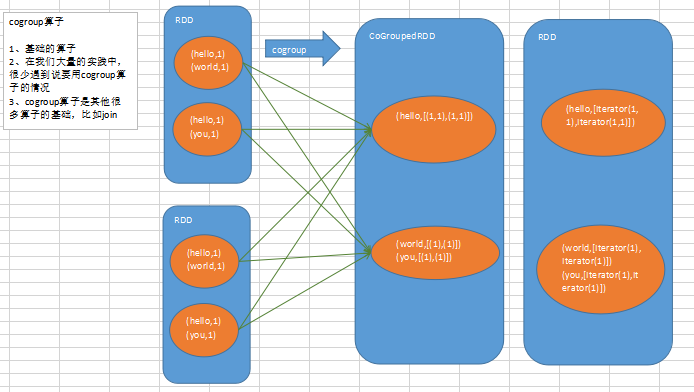
### reduceByKey算子内部实现原理剖析



### distinct算子内部实现原理剖析



### cogroup算子内部实现原理剖析

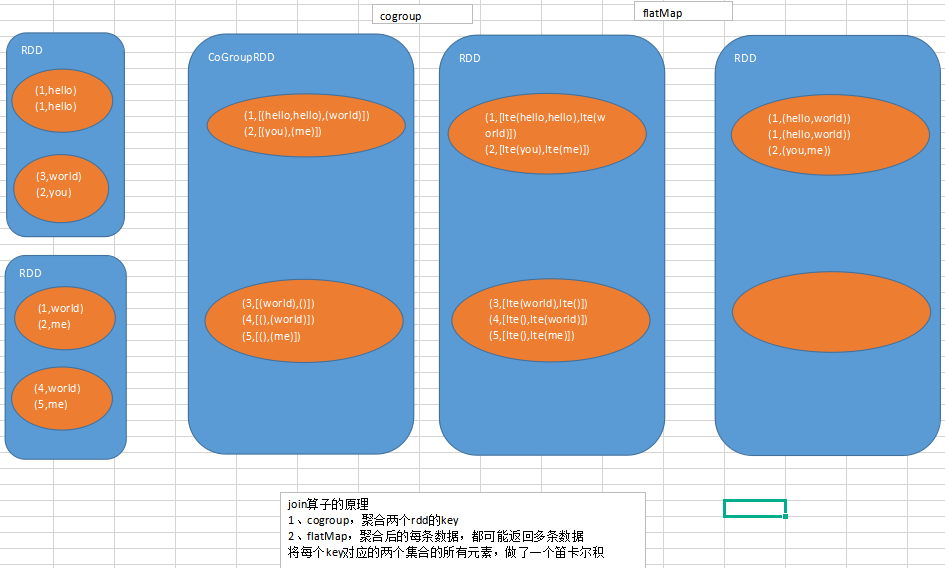


### intersection算子内部实现原理剖析

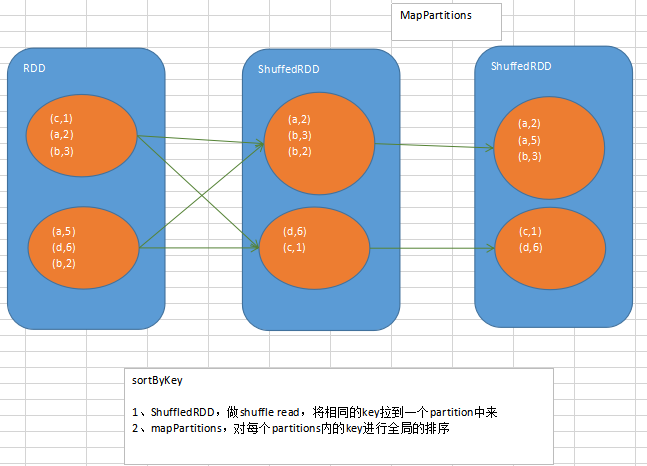
求交集。



### join算子内部实现原理剖析

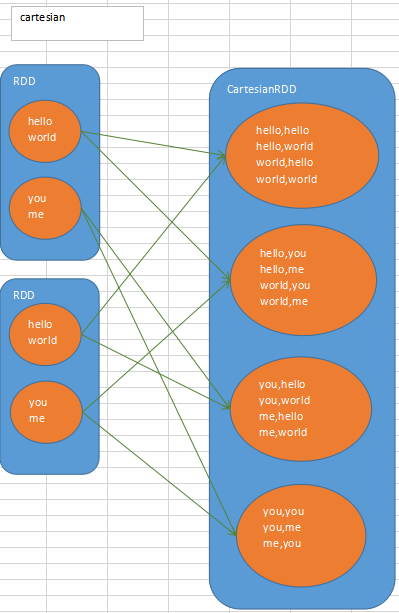


### Spark内核原理进阶-sortByKey算子内部实现原理剖析



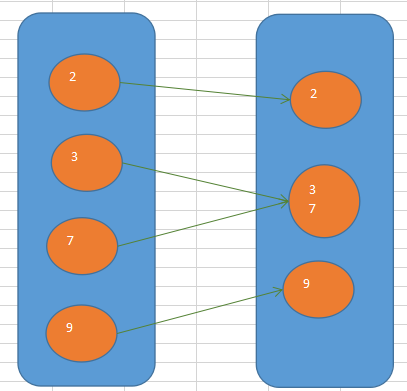
### cartesian算子内部实现原理剖析

笛卡儿积

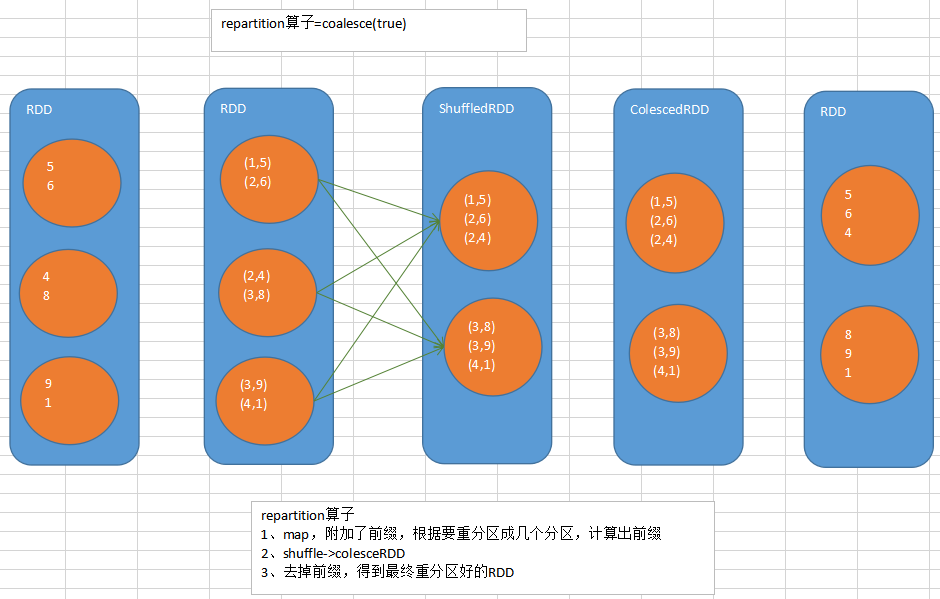


### coalesce算子内部实现原理剖析

减少partition数量

r

### repartition算子内部实现原理剖析



## Spark streaming

### updateStateByKey

updateStateByKey操作，可以让我们为每个key维护一份state，并持续不断的更新该state。1、首先，要定义一个state，可以是任意的数据类型；

2、其次，要定义state更新函数——指定一个函数如何使用之前的state和新值来更新state。

对于每个batch，Spark都会为每个之前已经存在的key去应用一次state更新函数，无论这个key在batch中是否有新的数据。如果state更新函数返回none，那么key对应的state就会被删除。

当然，对于每个新出现的key，也会执行state更新函数。

注意，updateStateByKey操作，要求必须开启Checkpoint机制。

### Transform

transform操作，应用在DStream上时，可以用于执行任意的RDD到RDD的转换操作。它可以用于实现，DStream API中所没有提供的操作。比如说，DStream API中，并没有提供将一个DStream中的每个batch，与一个特定的RDD进行join的操作。但是我们自己就可以使用transform操作来实现该功能。

DStream.join()，只能join其他DStream。在DStream每个batch的RDD计算出来之后，会去跟其他DStream的RDD进行join。

实时广告计费日志黑名单过滤

val validAdsClickLogDStream = userAdsClickLogDStream.transform(userAdsClickLogRDD => {

val joinedRDD = userAdsClickLogRDD.leftOuterJoin(blacklistRDD)

val filteredRDD = joinedRDD.filter(tuple => {

if(tuple.\_2.\_2.getOrElse(false)) {

false

} else {

true

}

})

val validAdsClickLogRDD = filteredRDD.map(tuple => tuple.\_2.\_1)

validAdsClickLogRDD

})

### Window

|  |  |
| --- | --- |
| countByWindow | 对每个滑动窗口的数据执行count操作 |
| reduceByWindow | 对每个滑动窗口的数据执行reduce操作 |
| reduceByKeyAndWindow | 对每个滑动窗口的数据执行reduceByKey操作 |
| countByValueAndWindow | 对每个滑动窗口的数据执行countByValue操作 |

### foreachRDD详解

通常在foreachRDD中，都会创建一个Connection，比如JDBC Connection，然后通过Connection将数据写入外部存储。

误区一：在RDD的foreach操作外部，创建Connection

这种方式是错误的，因为它会导致Connection对象被序列化后传输到每个Task中。而这种Connection对象，实际上一般是不支持序列化的，也就无法被传输。

dstream.foreachRDD { rdd =>

val connection = createNewConnection()

rdd.foreach { record => connection.send(record)

}

}

误区二：在RDD的foreach操作内部，创建Connection

这种方式是可以的，但是效率低下。因为它会导致对于RDD中的每一条数据，都创建一个Connection对象。而通常来说，Connection的创建，是很消耗性能的。

dstream.foreachRDD { rdd =>

rdd.foreach { record =>

val connection = createNewConnection()

connection.send(record)

connection.close()

}

}

合理方式一：使用RDD的foreachPartition操作，并且在该操作内部，创建Connection对象，这样就相当于是，为RDD的每个partition创建一个Connection对象，节省资源的多了。

dstream.foreachRDD { rdd =>

rdd.foreachPartition { partitionOfRecords =>

val connection = createNewConnection()

partitionOfRecords.foreach(record => connection.send(record))

connection.close()

}

}

合理方式二：自己手动封装一个静态连接池，使用RDD的foreachPartition操作，并且在该操作内部，从静态连接池中，通过静态方法，获取到一个连接，使用之后再还回去。这样的话，甚至在多个RDD的partition之间，也可以复用连接了。而且可以让连接池采取懒创建的策略，并且空闲一段时间后，将其释放掉。

dstream.foreachRDD { rdd =>

rdd.foreachPartition { partitionOfRecords =>

val connection = ConnectionPool.getConnection()

partitionOfRecords.foreach(record => connection.send(record))

ConnectionPool.returnConnection(connection)

}

}

# 部署、升级和监控应用程序

## 部署应用程序

1、有一个集群资源管理器，比如standalone模式下的Spark集群，Yarn模式下的Yarn集群等。

2、打包应用程序为一个jar包，课程中一直都有演示。

3、为executor配置充足的内存，因为Receiver接受到的数据，是要存储在Executor的内存中的，所以Executor必须配置足够的内存来保存接受到的数据。要注意的是，如果你要执行窗口长度为10分钟的窗口操作，那么Executor的内存资源就必须足够保存10分钟内的数据，因此内存的资源要求是取决于你执行的操作的。

4、配置checkpoint，如果你的应用程序要求checkpoint操作，那么就必须配置一个Hadoop兼容的文件系统（比如HDFS）的目录作为checkpoint目录.

5、配置driver的自动恢复，如果要让driver能够在失败时自动恢复，之前已经讲过，一方面，要重写driver程序，一方面，要在spark-submit中添加参数。

预写日志机制，简写为WAL，全称为Write Ahead Log。从Spark 1.2版本开始，就引入了基于容错的文件系统的WAL机制。如果启用该机制，Receiver接收到的所有数据都会被写入配置的checkpoint目录中的预写日志。这种机制可以让driver在恢复的时候，避免数据丢失，并且可以确保整个实时计算过程中，零数据丢失。

要配置该机制，首先要调用StreamingContext的checkpoint()方法设置一个checkpoint目录。然后需要将spark.streaming.receiver.writeAheadLog.enable参数设置为true。

然而，这种极强的可靠性机制，会导致Receiver的吞吐量大幅度下降，因为单位时间内，有相当一部分时间需要将数据写入预写日志。如果又希望开启预写日志机制，确保数据零损失，又不希望影响系统的吞吐量，那么可以创建多个输入DStream，启动多个Rceiver。

此外，在启用了预写日志机制之后，推荐将复制持久化机制禁用掉，因为所有数据已经保存在容错的文件系统中了，不需要在用复制机制进行持久化，保存一份副本了。只要将输入DStream的持久化机制设置一下即可，persist(StorageLevel.MEMORY\_AND\_DISK\_SER)。（之前讲过，默认是基于复制的持久化策略，\_2后缀）

如果集群资源有限，并没有大到，足以让应用程序一接收到数据就立即处理它，Receiver可以被设置一个最大接收限速，以每秒接收多少条单位来限速。

spark.streaming.receiver.maxRate和spark.streaming.kafka.maxRatePerPartition参数可以用来设置，前者设置普通Receiver，后者是Kafka Direct方式。

Spark 1.5中，对于Kafka Direct方式，引入了backpressure机制，从而不需要设置Receiver的限速，Spark可以自动估计Receiver最合理的接收速度，并根据情况动态调整。只要将spark.streaming.backpressure.enabled设置为true即可。

在企业实际应用场景中，通常是推荐用Kafka Direct方式的，特别是现在随着Spark版本的提升，越来越完善这个Kafka Direct机制。优点：1、不用receiver，不会独占集群的一个cpu core；2、有backpressure自动调节接收速率的机制；3、....。

## 升级应用程序

由于Spark Streaming应用程序都是7 \* 24小时运行的。因此如果需要对正在运行的应用程序，进行代码的升级，那么有两种方式可以实现：

1、升级后的Spark应用程序直接启动，先与旧的Spark应用程序并行执行。当确保新的应用程序启动没问题之后，就可以将旧的应用程序给停掉。但是要注意的是，这种方式只适用于，能够允许多个客户端读取各自独立的数据，也就是读取相同的数据。

2、小心地关闭已经在运行的应用程序，使用StreamingContext的stop()方法，可以确保接收到的数据都处理完之后，才停止。然后将升级后的程序部署上去，启动。这样，就可以确保中间没有数据丢失和未处理。因为新的应用程序会从老的应用程序未消费到的地方，继续消费。但是注意，这种方式必须是支持数据缓存的数据源才可以，比如Kafka、Flume等。如果数据源不支持数据缓存，那么会导致数据丢失。

注意：配置了driver自动恢复机制时，如果想要根据旧的应用程序的checkpoint信息，启动新的应用程序，是不可行的。需要让新的应用程序针对新的checkpoint目录启动，或者删除之前的checkpoint目录。

## 监控应用程序

当Spark Streaming应用启动时，Spark Web UI会显示一个独立的streaming tab，会显示Receiver的信息，比如是否活跃，接收到了多少数据，是否有异常等；还会显示完成的batch的信息，batch的处理时间、队列延迟等。这些信息可以用于监控spark streaming应用的进度。

Spark UI中，以下两个统计指标格外重要：

1、处理时间——每个batch的数据的处理耗时

2、调度延迟——一个batch在队列中阻塞住，等待上一个batch完成处理的时间

如果batch的处理时间，比batch的间隔要长的话，而且调度延迟时间持续增长，应用程序不足以使用当前设定的速率来处理接收到的数据，此时，可以考虑增加batch的间隔时间。

# 容错机制以及事务语义详解

要理解Spark Streaming提供的容错机制，先回忆一下Spark RDD的基础容错语义：

1、RDD，Ressilient Distributed Dataset，是不可变的、确定的、可重新计算的、分布式的数据集。每个RDD都会记住确定好的计算操作的血缘关系，（val lines = sc.textFile(hdfs file); val words = lines.flatMap(); val pairs = words.map(); val wordCounts = pairs.reduceByKey()）这些操作应用在一个容错的数据集上来创建RDD。

2、如果因为某个Worker节点的失败（挂掉、进程终止、进程内部报错），导致RDD的某个partition数据丢失了，那么那个partition可以通过对原始的容错数据集应用操作血缘，来重新计算出来。

3、所有的RDD transformation操作都是确定的，最后一个被转换出来的RDD的数据，一定是不会因为Spark集群的失败而丢失的。

Spark操作的通常是容错文件系统中的数据，比如HDFS。因此，所有通过容错数据生成的RDD也是容错的。然而，对于Spark Streaming来说，这却行不通，因为在大多数情况下，数据都是通过网络接收的（除了使用fileStream数据源）。要让Spark Streaming程序中，所有生成的RDD，都达到与普通Spark程序的RDD，相同的容错性，接收到的数据必须被复制到多个Worker节点上的Executor内存中，默认的复制因子是2。

基于上述理论，在出现失败的事件时，有两种数据需要被恢复：

1、数据接收到了，并且已经复制过——这种数据在一个Worker节点挂掉时，是可以继续存活的，因为在其他Worker节点上，还有它的一份副本。

2、数据接收到了，但是正在缓存中，等待复制的——因为还没有复制该数据，因此恢复它的唯一办法就是重新从数据源获取一份。

此外，还有两种失败是我们需要考虑的：

1、Worker节点的失败——任何一个运行了Executor的Worker节点的挂掉，都会导致该节点上所有在内存中的数据都丢失。如果有Receiver运行在该Worker节点上的Executor中，那么缓存的，待复制的数据，都会丢失。

2、Driver节点的失败——如果运行Spark Streaming应用程序的Driver节点失败了，那么显然SparkContext会丢失，那么该Application的所有Executor的数据都会丢失。

流式计算系统的容错语义，通常是以一条记录能够被处理多少次来衡量的。有三种类型的语义可以提供：

1、最多一次：每条记录可能会被处理一次，或者根本就不会被处理。可能有数据丢失。

2、至少一次：每条记录会被处理一次或多次，这种语义比最多一次要更强，因为它确保零数据丢失。但是可能会导致记录被重复处理几次。

3、一次且仅一次：每条记录只会被处理一次——没有数据会丢失，并且没有数据会处理多次。这是最强的一种容错语义。

在Spark Streaming中，处理数据都有三个步骤：

1、接收数据：使用Receiver或其他方式接收数据。

2、计算数据：使用DStream的transformation操作对数据进行计算和处理。

3、推送数据：最后计算出来的数据会被推送到外部系统，比如文件系统、数据库等。

如果应用程序要求必须有一次且仅一次的语义，那么上述三个步骤都必须提供一次且仅一次的语义。每条数据都得保证，只能接收一次、只能计算一次、只能推送一次。Spark Streaming中实心这些语义的步骤如下：

1、接收数据：不同的数据源提供不同的语义保障。

2、计算数据：所有接收到的数据一定只会被计算一次，这是基于RDD的基础语义所保障的。即使有失败，只要接收到的数据还是可访问的，最后一个计算出来的数据一定是相同的。

3、推送数据：output操作默认能确保至少一次的语义，因为它依赖于output操作的类型，以及底层系统的语义支持（比如是否有事务支持等），但是用户可以实现它们自己的事务机制来确保一次且仅一次的语义。

## 接收数据的容错语义

1、基于文件的数据源

如果所有的输入数据都在一个容错的文件系统中，比如HDFS，Spark Streaming一定可以从失败进行恢复，并且处理所有数据。这就提供了一次且仅一次的语义，意味着所有的数据只会处理一次。

2、基于Receiver的数据源

对于基于Receiver的数据源，容错语义依赖于失败的场景和Receiver类型。

可靠的Receiver：这种Receiver会在接收到了数据，并且将数据复制之后，对数据源执行确认操作。如果Receiver在数据接收和复制完成之前，就失败了，那么数据源对于缓存的数据会接收不到确认，此时，当Receiver重启之后，数据源会重新发送数据，没有数据会丢失。

不可靠的Receiver：这种Receiver不会发送确认操作，因此当Worker或者Driver节点失败的时候，可能会导致数据丢失。

不同的Receiver，提供了不同的语义。如果Worker节点失败了，那么使用的是可靠的Receiver的话，没有数据会丢失。使用的是不可靠的Receiver的话，接收到，但是还没复制的数据，可能会丢失。如果Driver节点失败的话，所有过去接收到的，和复制过缓存在内存中的数据，全部会丢失。

要避免这种过去接收的所有数据都丢失的问题，Spark从1.2版本开始，引入了预写日志机制，可以将Receiver接收到的数据保存到容错存储中。如果使用可靠的Receiver，并且还开启了预写日志机制，那么可以保证数据零丢失。这种情况下，会提供至少一次的保障。（Kafka是可以实现可靠Receiver的）

|  |  |  |
| --- | --- | --- |
| **部署场景** | **Worker失败** | **Driver失败** |
| Spark 1.1以前的版本，或者是Spark 1.2以后版本，但是没有开启预写日志机制 | 1、不可靠Receiver，会导致缓存数据丢失  2、可靠的Receiver，可以保证数据零丢失  3、至少一次的语义 | 1、不可靠Receiver，缓存的数据全部丢失  2、任何Receiver，过去接收的所有数据全部丢失  3、没有容错语义 |
| Spark 1.2以后版本，并开启了预写日志机制 | 1、可靠Receiver，零数据丢失  2、至少一次的语义 | 1、可靠Receiver和文件，零数据丢失 |

从Spark 1.3版本开始，引入了新的Kafka Direct API，可以保证，所有从Kafka接收到的数据，都是一次且仅一次。基于该语义保障，如果自己再实现一次且仅一次语义的output操作，那么就可以获得整个Spark Streaming应用程序的一次且仅一次的语义。

## 输出数据的容错语义

output操作，比如foreachRDD，可以提供至少一次的语义。那意味着，当Worker节点失败时，转换后的数据可能会被写入外部系统一次或多次。对于写入文件系统来说，这还是可以接收的，因为会覆盖数据。但是要真正获得一次且仅一次的语义，有两个方法：

1、幂等更新：多次写操作，都是写相同的数据，例如saveAs系列方法，总是写入相同的数据。

2、事务更新：所有的操作都应该做成事务的，从而让写入操作执行一次且仅一次。给每个batch的数据都赋予一个唯一的标识，然后更新的时候判定，如果数据库中还没有该唯一标识，那么就更新，如果有唯一标识，那么就不更新。

dstream.foreachRDD { (rdd, time) =>

rdd.foreachPartition { partitionIterator =>

val partitionId = TaskContext.get.partitionId()

val uniqueId = generateUniqueId(time.milliseconds, partitionId)

// partitionId和foreachRDD传入的时间，可以构成一个唯一的标识

}

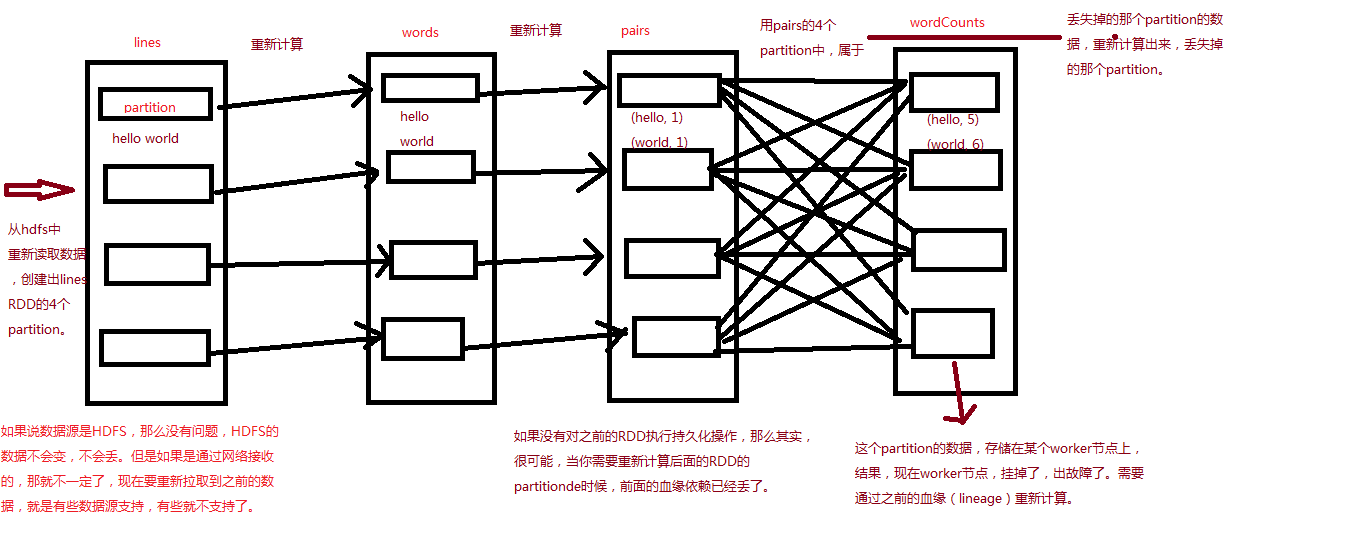
}

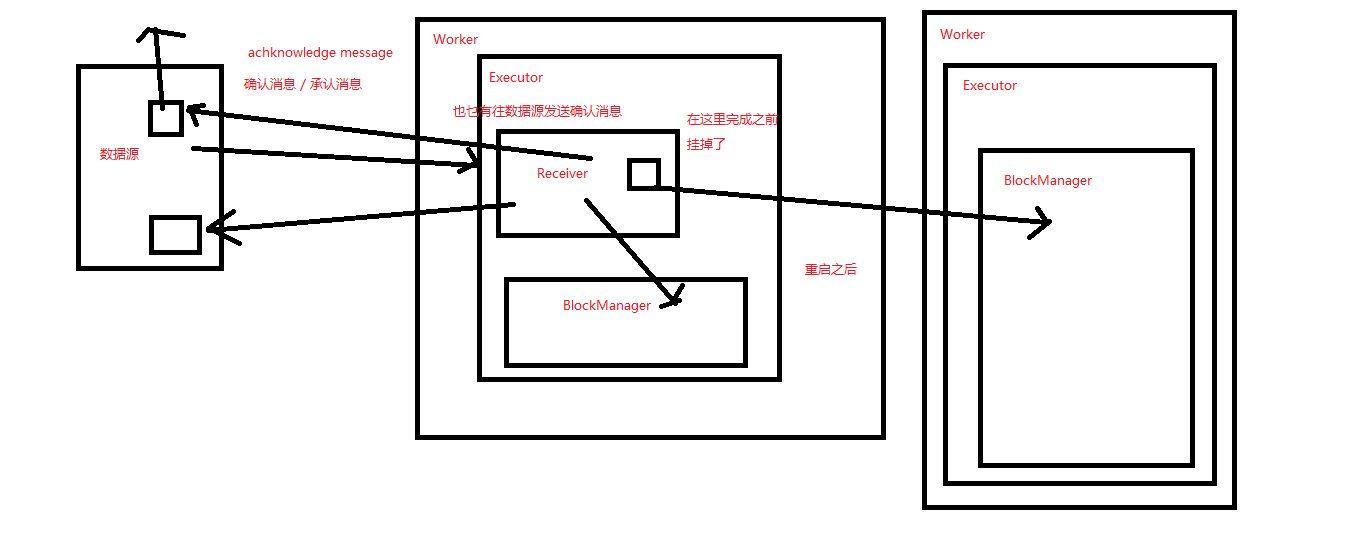
## Storm的容错语义

Storm首先，它可以实现消息的高可靠性，就是说，它有一个机制，叫做Acker机制，可以保证，如果消息处理失败，那么就重新发送。保证了，至少一次的容错语义。但是光靠这个，还是不行，数据可能会重复。

Storm提供了非常非常完善的事务机制，可以实现一次且仅一次的事务机制。事务Topology、透明的事务Topology、非透明的事务Topology，可以应用各种各样的情况。对实现一次且仅一次的这种语义的支持，做的非常非常好。用事务机制，可以获得它内部提供的一个唯一的id，然后基于这个id，就可以实现，output操作，输出，推送数据的时候，先判断，该数据是否更新过，如果没有的话，就更新；如果更新过，就不要重复更新了。

所以，至少，在容错 / 事务机制方面，我觉得Spark Streaming还有很大的空间可以发展。特别是对于output操作的一次且仅一次的语义支持！





# spark各版本对比

## 1.4x新特性

1、Spark Core

1.1 提供REST API供外界开发者获取Spark内部的各种信息（jobs / stages / tasks / storage info），基于这些API，可以搭建自己的Spark监控系统。

1.2 shuffle阶段，默认将map端写入磁盘的数据进行序列化，优化io性能。

1.3 钨丝计划（Project Tungsten），提供了UnsafeShuffleManager，使用缓存友好的排序算法，降低了shuffle的内存使用，提高了排序性能。

2、Spark Streaming

2.1 提供了新的Spark Streaming的UI，能够更好，更清晰的监控Spark Streaming应用程序的运行状况。

2.2 支持Kafka 0.8.2版本

3、Spark SQL and DataFrame

3.1 支持ORCFile

3.2 提供了一些window function（窗口函数）

3.3 优化了join的性能

## Spark 1.5.x的新特性

1、DataFrame底层执行的性能优化（钨丝计划第一阶段）

1.1 Spark自己来管理内存，而不再依靠JVM管理内容。这样就可以避免JVM GC的性能开销，并且能够控制OOM的问题。

1.2 Java对象直接使用内部的二进制格式存储和计算，省去了序列化和反序列化的性能开销，而且更加节省内存开销。

1.3 完善了Shuffle阶段的UnsafeShuffleManager，增加了不少新功能，优化shuffle性能。

1.4 默认使用code-gen，使用cache-aware算法，加强了join、aggregation、shuffle、sorting的性能，增强了window function的性能，性能比1.4.x版本提高数倍

2、DataFrame

2.1 实现了新的聚合函数接口，AggregateFunction2，并且提供了7个新的内置聚合函数。

2.2 实现了100多个新的expression function，例如unix\_timestamp等，增强了对NaN的处理

2.3 支持连接不同版本的hive metastore

2.4 支持Parquet 1.7

3、Spark Streaming：更完善的python支持、非实验的Kafka Direct API等等。

# 爬坑总结

case class要定义在object之外。

sqlContext中3个表不能同时join，需要两两join后再join一次。

sqlContext中不能直接使用values插入数据。