**Scala**

1. **Scala 语言特点，Scala与Java的区别？**

Scala 是一种有趣的语言。它一方面吸收继承了多种语言中的优秀特性，一方面又没有抛弃Java这个强大的平台，它运行在Java虚拟机 (Java Virtual Machine)之上，轻松实现和丰富的Java类库互联互通。它既支持面向对象的编程方式，又支持函数式编程。它写出的程序像动态语言一样简洁，但事实上它确是严格意义上的静态语言。

Scala六大特征：

（1）java和scala可以混编。

（2）类型推测（自动推测类型），在可由上下文推断的情况下，可以省略类型。

（3）支持并发和分布式Actor。

（4）Traits特质特性（类似java中interface和abstract结合）。

（5）模式匹配（类似javaswitch）。

（6）高阶函数：函数的参数是函数，或者函数的返回类型是函数，或者函数的参数和函数的返回类型是函数的函数。

Scala与Java的区别：

不同点：

1. scala与java都有七种数值类型int、short、long、byte、float、double、boolean这7种，但是scala中这7种值类型是类，在java中属于基本类型，java中，数据类型分成基本类型和引用类型，scala中不区分。
2. scala包含两种类型的变量，val变量相当于java中的常量，var则是常规变量
3. scala中的变量或函数的类型总是写在变量或函数的后面，变量的类型可以省略。
4. 在scala中操作符是方法，除了字母数字之外的其他特殊字符也可以作为方法，在java中操作符不是方法。
5. scala中的通配符是\_，java中的通配符是\*。
6. scala中的unit相当于java中的void。
7. scala中没有static，但是可以用object来达到java中相同的效果，scala中的object是单例对象，相当于java中的工具类，可以看成是定义静态的方法的类，object不可传参数。
8. scala支持关联映射，如可以用(key -> value)表示一个键值对。
9. scala具有类型推断功能，在可由上下文推断的情况下，可以省略类型，也可以省略分号。
10. scala的return是可选的，方法调用会自动返回最后求值的表达式。如果scala使用了return则需要显示指定方法的返回值Java需要显示的采用return进行值的返回。
11. 类和方法修饰符的默认值：scala默认是public，java默认是protected。
12. scala不支持接口interface，采用trait（类似于Java中的抽象类）。java支持接口。
13. scala是面向对象+面向函数编程的语言，java是面向对象的语言。

相同点：

（1）都是基于JVM的语言

**2.** **Scala Actor通信模型**

Actors将状态和行为封装在一个轻量的进程/线程中，但是不和其他Actors分享状态，每个Actors有自己的世界观，当需要和其他Actors交互时，通过发送事件和消息，发送是异步的，非堵塞的(fire-andforget)，发送消息后不必等另外Actors回复，也不必暂停，每个Actors有自己的消息队列，进来的消息按先来后到排列，这就有很好的并发策略和可伸缩性，可以建立性能很好的事件驱动系统。

Actor的特征：

（1）ActorModel是消息传递模型,基本特征就是消息传递

（2）消息发送是异步的，非阻塞的

（3）消息一旦发送成功，不能修改

（4）Actor之间传递时，自己决定决定去检查消息，而不是一直等待，是异步非阻塞的

（1）单向通信

class MyActor extends Actor{

override def act():unit={

while(true){

receive {

case msg:String => println(“String msg”)+msg

case msg:Int => println(“Int msg”)+msg

case \_ => println(“else msg”)+msg

}

}

}

}

object ActorSend{

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

val actor = new MyActor()

actor.start()

actor ! “hello”

}

}

(2)Actor与Actor通信

case class Msg(actor: Actor, message: String)

class Actor1 extends Actor {

override def act(): Unit = {

while (true) {

receive {

case msg: Msg => {

if ("hello actor1".equals(msg.message)) {

println("我接收到actor2发送的消息:" + msg.message)

msg.actor ! "你好呀！actor2"

} else if ("actor1你在做什么呀".equals(msg.message)) {

println("我接收到actor2发送的消息:" + msg.message)

msg.actor ! "我在想你呀"

}

}

case \_ => println("无法识别")

}

}

}

}

class Actor2(actor1: Actor) extends Actor {

actor1 ! Msg(this, "hello actor1")

override def act(): Unit = {

while (true) {

receive {

case msg: String => {

if ("你好呀！actor2".equals(msg)) {

println("我接收到actor1发送的消息：" + msg)

actor1 ! Msg(this, "actor1你在做什么呀")

}

else if ("我在想你呀".equals(msg)) println(msg)

}

case msg: Int => println("拜拜 actor1")

case \_ => println("无法识别")

}

}

}

}

object ActorToActor {

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

val actor1 = new Actor1()

actor1.start()

val actor2 = new Actor2(actor1)

actor2.start()

}

}

**Spark**

**3.Spark 与MR的区别？**

不同点：

1. Spark基于内存，MR基于HDFS。Spark处理数据的能力一般是MR的十倍以上，Spark中除了基于内存计算外，还有DAG有向无环图来切分任务的执行先后顺序。
2. spark把运算的中间数据存放在内存中，迭代计算效率更高；mapreduce的中间结果需要落地磁盘，这样必然有磁盘I/O操作，因此性能比spark低。
3. sprk容错性高，它通过弹性分布式数据集RDD来实现高效容错，RDD是一组分布式的存储在内存节点内存中的只读性质的数据集，这些集合是弹性的，某一部分丢失或者出错，可以通过整个数据集的计算流程的血缘关系来实现重建；MR丢失数据或者出错只能重新计算，成本较高。
4. spark更加通用，spark提供了transformation和action两类功能强大的算子。
5. spark框架和生态更为复杂，首先有RDD、血缘lineage、执行时的有向无环图DAG、stage划分等等，很多时候spark作业都需要根据不同业务场景的需要进行调优已达到性能要求；mapreduce框架及其生态相对较为简单，对性能的要求也相对较弱，但是运行较为稳定，适合长期后台运行。
6. spark生态更为丰富，功能更为强大、性能更佳，适用范围更广；MR更简单、稳定性好、适合离线海量数据挖掘计算。
7. MR基于YARN运行；spark运行模式有：Local、Standlone、Yarn、Mesos。

相同点：

1. 都是分布式计算框架

**4.Spark代码流程**

1. 创建SparkConf对象，可以设置当前程序运行模式，可以设置当前app的名称，也可以设置资源需求。
2. 创建SparkContext对象，用于创建RDD。
3. 基于spark的上下文创建一个RDD，对RDD进行处理。
4. 应用程序中要有Action类算子来触发Transformation类算子执行。
5. 关闭spark上下文SparkContext对象。

**5.解释Spark核心RDD**

（1）概念

RDD（Resilient Distributed Dateset）叫做弹性分布式数据集，是spark中最基本的数据抽象，它代表一个不可变、可分区、里面的元素可以并行计算的集合。RDD具有数据流模型的特点：自动容错、位置感知性调度和可伸缩性。RDD允许用户在执行多个查询时显示的将工作集缓存在内存中，后续的查询能够重用工作集，极大的提升了查询速度。

（2）五大特征

1）RDD是由一系列的partition组成的，Partition是数据集的基本组成单位。每个partition都会被一个计算任务处理，并决定并行计算的粒度。

2）函数是作用在每一个partition（split）上的。Spark中RDD的计算是以分片为单位的，每个RDD都会实现compute函数以达到这个目的。Compute函数会对迭代器进行复合，不需要保存每次计算的结果。

3）RDD之间有一系列的依赖关系。RDD的每次转换都会生成一个新的RDD，所以RDD之间就会形成前后依赖关系，在部分分区数据丢失时，Spark可以通过这个依赖关系重新计算丢失的分区数据，而不是对RDD的所有分区进行重新计算。

4）分区器是作用在K,V格式的RDD上。分区器不但决定了RDD本身的分片数量，也决定了parent RDD Shuffle输出时的分片数量。

5）RDD提供一系列最佳的计算位置，实现数据处理本地化。按照大数据“计算向数据移动”的理念，Spark在进行任务调度的时候，会尽可能地将计算任务分配到其所要处理数据块的存储位置。

（3）注意

1）textFile读取HDFS文件方法底层封装的是MR读取文件的方式，读取文件之前先split，默认split大小是一个block块，每个split对应RDD中的一个partition。

2）RDD实际上不存储数据，存储的的只是真实数据的分区信息getPartitions和针对单个分区的读取方法compute，还有就是数据的类型。

3）K,V格式的RDD就是RDD中存储的数据都是二元组对象。

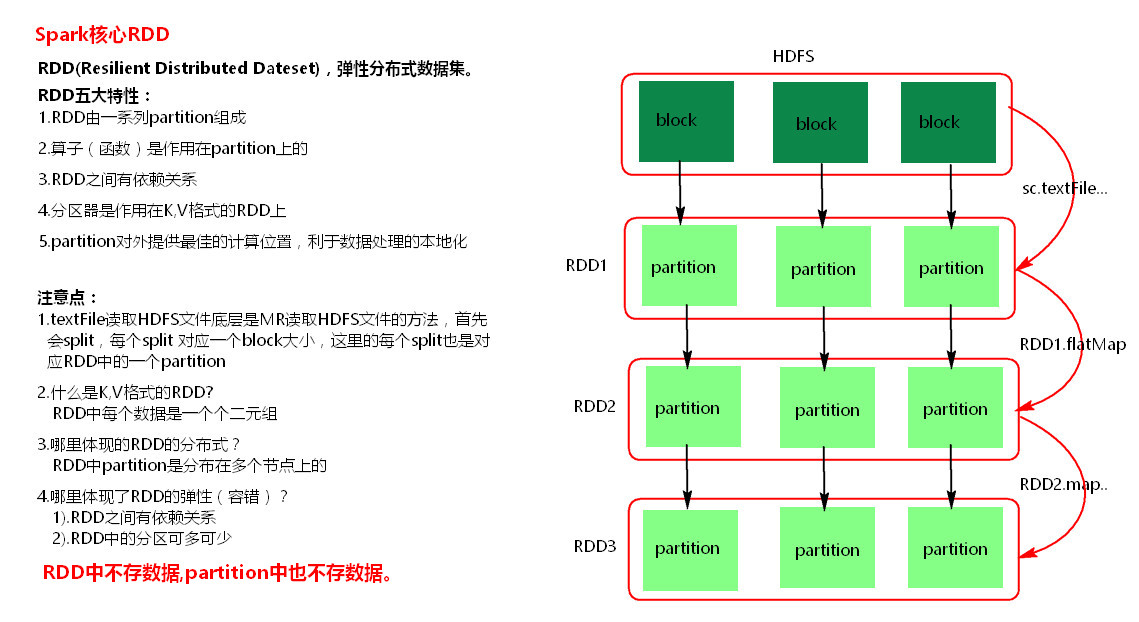
4）RDD的弹性容错体现：partition数量、大小没有限制；RDD之间有依赖关系，可以基于上一个RDD重新计算出RDD。

5）RDD分布式：RDD是由partition组成的，partition是分布在不同的节点上的。

6）RDD提供最佳计算位置，体现数据本地化，体现了“计算移动数据不移动”的理念。

7）初代RDD:处于血统的顶层，存储的是任务所需的数据的分区信息，还有单个分区数据读取的方法，没有依赖的RDD，因为它就是依赖的开始。子代RDD:处于血统的下层，存储的东西就是初代RDD到底干了什么才会产生自己，还有就是初代RDD的引用。

1. RDD理解图



**6.举例Spark 几类算子**

（1）Transformations转换算子

1）概念：

Transformations类算子是一类算子（函数）叫做转换算子，Transformations算子是延迟执行，也叫做懒加载执行。

2）类别

①filter，过滤符合条件的记录数，true保留，false过滤掉。

代码var filters=lines.filter("hello world".equals(\_))

②map，将一个RDD中的每个数据项通过map中的函数映射变为一个新的元素。特点：输入一条，输出一条。

代码val maps : RDD[(String, Int)] = words.map(word => {new Tuple2(word, 1) })

③flatMap，先map后flat，与map类似，每个输入项映射0到多个输出项。

代码val words=lines.flatMap(line=>{line.split(" ")})

④sample，随机抽样算子，根据传进去的小数按比例进行有放回或无放回的抽样。true为有放回，false为无放回。

代码val samples = lines.sample(true,0.5)

⑤redueByKey，将相同的key根据相应的逻辑进行处理

代码pairWords.reduceByKey((v1, v2) => {v1 + v2 })

⑥sortByKey/sortBy，作用在K,V格式的RDD上，对key进行升序或者降序排序。默认为true升序，false为降序。

代码reduceResult.sortBy(tp => {tp.\_2}, false)

（2）Action行动算子

1）概念

Action类算子也是一类算子叫做行动算子，Action类算子是

触发执行。一个application应用程序中有几个Action类算

子执行，就有几个kob运行。

2）类别

①count，返回数据集中的元素总数。会在结果计算完成后回收到Driver端。

代码val count: Long = lines.count()

②take(n),返回一个包含数据集前n个元素的集合（数组）。

代码val c: Array[String] = lines.take(5)

③first，first=take(1)，返回数据集中的第一个元素。

代码val d: String = lines.first()

④foreach，循环遍历数据集中的每个元素，运行相应的逻辑。

代码lines.foreach(line => {println(line)})

⑤collect，将计算结果回收到Driver端。

代码val f: Long = lines.count()

（3）控制算子

1）概念

控制算子有三种，cache、persist、checkpoint，都可以将

RDD持久化，持久化的单位是partition。cache都是懒执行。

必须有一个action类算子触发执行。checkpoint算子不仅能

将RDD持久化到磁盘，还能切断RDD之间的依赖关系。

2）类别

①cache，默认将RDD数据持久化到内存中，cache是懒执行。

代码

val conf = new SparkConf()

conf.setMaster("local")

conf.setAppName("test")

val sc = new SparkContext(conf)

//获取数据

var lines = sc.textFile("./data/persistData.txt")

//将数据保存到内存中

lines = lines.cache()

//第一次计算从磁盘获取数据

val count1: Long = lines.count()

//第二次计算从内存中获取数据

val count2: Long = lines.count()

sc.stop()

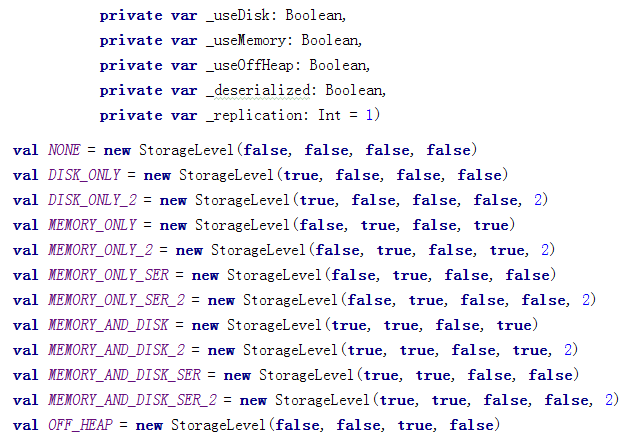
②persist，可以指定持久化级别，最常用的是MEMORY\_ONLY

和MEMORY\_AND\_DISK。“2”代表有副本数，两个。

持久化级别：参数1.是否磁盘存储，2.是否内存存储，3.是否是外部存储（对外），4.是否不序列号，5.是否有副本

注意：尽量避免使用“\_2”级别和DISK\_ONLY级别

cache()=persist()=persist(StorageLevel.MEMORY\_ONLY)



③checkpoint，将RDD持久化到磁盘,不仅持久化数据，还会

持久化元数据，还可以切断RDD之间的依赖关系。Checkpoint

目录数据当application执行完后不会被清除。

执行原理：1.当RDD的job执行完毕后，会从finalRDD从后

往前回溯。2.当回溯到一个RDD调用了checkpoint方法，会

对当前的RDD 做一个标记。3.spark框架会自动启动一个新的

job，重新计算这个RDD的数据，将数据持久化到HDFS上。4.

优化：对RDD执行checkpoint之前，最好对这个RDD先执行

cache，这样新启动的job只需要将内存中的数据拷贝到Hdfs

上就可以，省去重新计算着一步。

注意：不要频繁进行checkpoint，最多使用一个checkpoint。

代码使用：

SparkConf conf = new SparkConf();

conf.setMaster("local").setAppName("checkpoint");

JavaSparkContext sc = new JavaSparkContext(conf);

sc.setCheckpointDir("./checkpoint");

JavaRDD<Integer> parallelize =

sc.parallelize(Arrays.asList(1,2,3));

parallelize.checkpoint();

parallelize.count();

sc.stop();

3）cache和persist的注意事项：

①cache和persist都是懒执行，必须有一个action类算子

触发执行

②cache和persist算子的返回值可以赋值给一个变量，在其

他job中直接使用这个变量就是使用持久化的数据了。持久化

的单位是partition。

③cache和persist算子后不能立即紧跟action算子，例：

rdd.cache().count()返回的不是持久化的RDD，是一个数值。

④cache和persist算子持久化的数据当application执行完

成之后会被清除。

**7.cache、persist、checkpoint区别**

（1）cache与persist区别

①RDD的cache()方法其实调用的就是persist方法，缓存策略均为MEMORY\_ONLY。

②cache与persist的唯一区别在于：cache只有一个默认的缓存级别MEMORY\_ONLY，而persist可以根据StorageLevel设置其它的缓存级别。

（2）cache和checkpoint区别

①cache将RDD以及RDD的血统(记录了这个RDD如何产生)缓存到内存中，当缓存的RDD失效的时候(如内存损坏)，它们可以通过血统重新计算来进行恢复。但是checkpoint将RDD缓存到了HDFS中，同时忽略了它的血统(也就是RDD之前的那些依赖)，可以利用HDFS多副本特性保证容错。

②cache机制是每计算出一个要cache的partition就直接将其cache到内存了。但checkpoint没有使用这种第一次计算得到就存储的方法，而是等到job结束后另外启动专门的job去完成checkpoint。也就是说需要checkpoint的RD会被计算两次。因此在使用rdd.checkpoint()的时候建议加上rdd.cache()，这样第二次运行的job就不用再去计算该rdd了，直接读取cache写磁盘。

③checkpoint持久化需要手动指定目录，cache系统自动指定。

④checkpoint将RDD持久化到HDFS，只要不手动删除就会一直存在；而cache在当前application执行完毕后会被清空。也就是说可以被下一个driver program使用，而cache不能被其他driver program使用，cache能够让重复数据在同一个 application 中的jobs间共享。

（3）persist和checkpoint区别

①persist(StorageLevel.DISK\_ONLY)虽然可以将RDD的partition持久化到磁盘，但该Partition由blockManager管理，一但application执行完毕（driver program执行结束，也就是executor所在进程CoarseGrainedExecutorBackend stop，blockManager也会stop），被持久化到磁盘的RDD也会被清空（整个blockManager使用的local文件夹被删除）；checkpoint将RDD持久化到HDFS或者本地文件夹，如果不手动remove掉，是一直存在的，也就是说可以被下一个driver program使用，而persist不能被其他driver program使用。

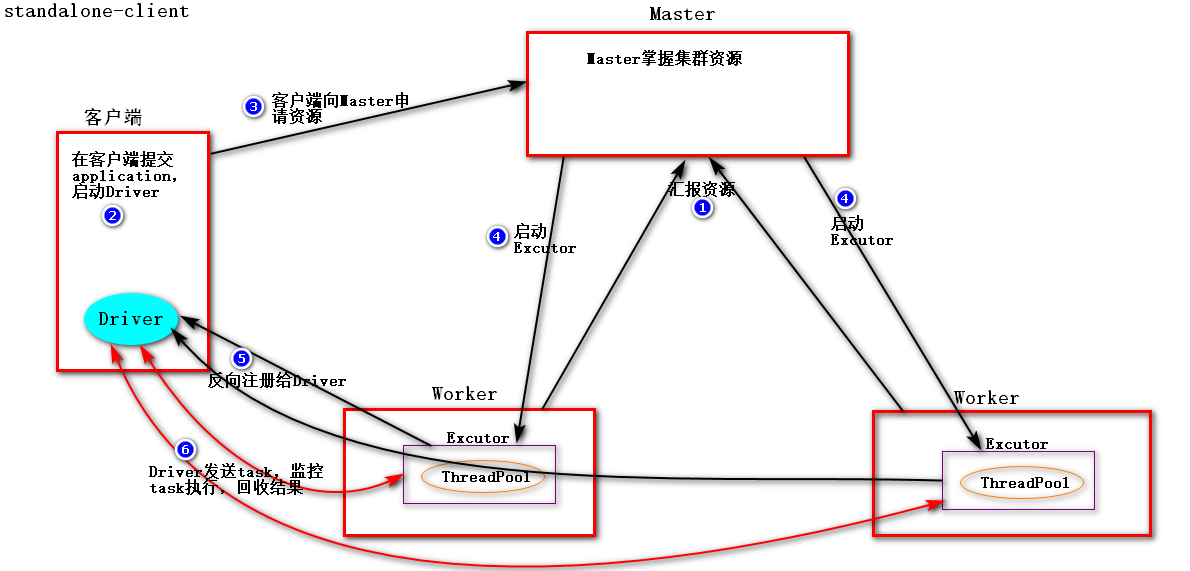
②checkpoint持久化需要手动指定目录，persist系统自动指定。

8.Spark基于Standalone-client,cluster和Yarn-

client，cluster提交任务的流程。

（1）Spark基于Standalone-client模式提交任务

1）流程图



2）流程

①client提交任务后，会在客户端启动Driver进程

②Driver会向Master申请启动Application的资源

③Master找到满足资源的worker节点，启动worker上的Excutor

④Excutor启动后反向注册给Driver

⑤Driver将task发送到worker执行，并监控task执行

⑥worker将task执行结果返回到Driver端

3）总结

Client模式适用于测试调试程序。Driver进程是在客户端启动的，这

里的客户端就是指提交应用程序的当前节点。在Driver端可以看到task

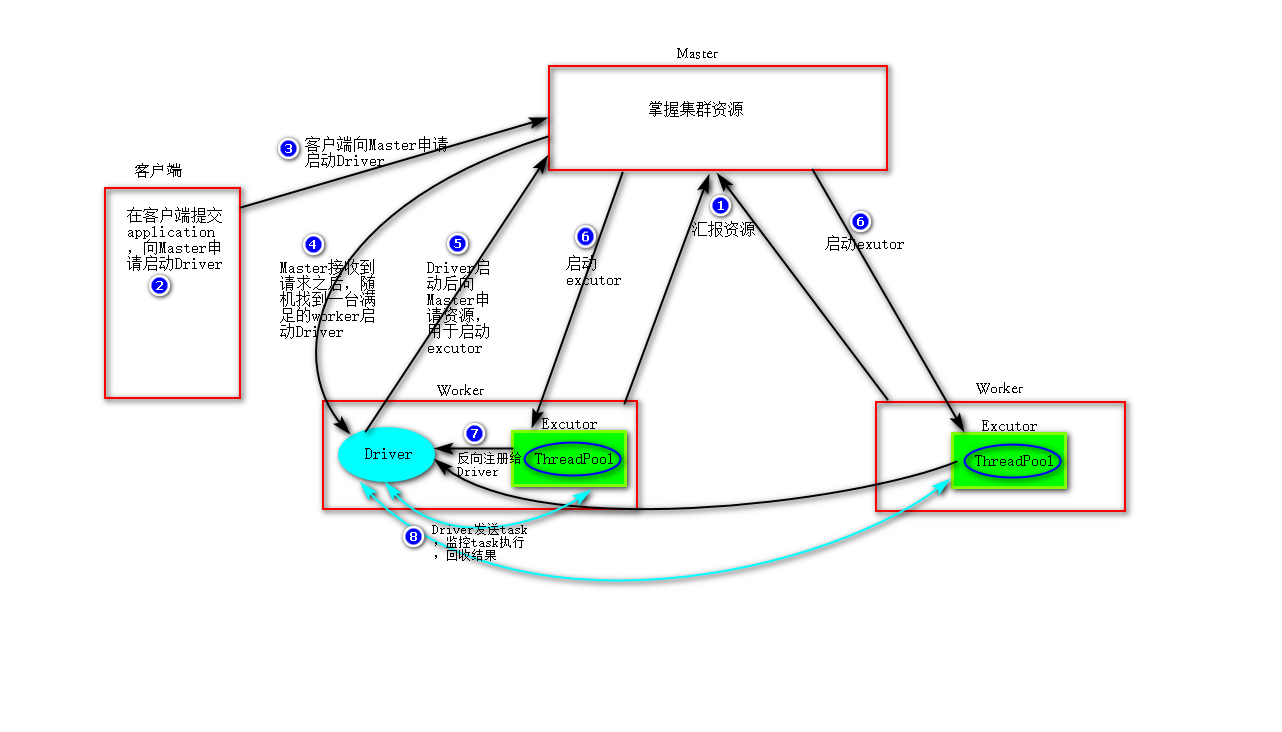
执行的情况。生产环境下不能使用client模式，是因为：每个提交的

application都有自己独立的Driver，假设要提交100个application

到集群运行，Driver每次都会在client端启动，Driver和worker集

群之间有大量的通信，那么就会导致客户端100次网卡流量暴增的问题。

1. standalone-cluster模式
   1. 流程图



* 1. 流程
     1. cluster模式提交应用程序后，会向Master申请启动Driver
     2. Master接收请求后，随机在集群一台满足的节点启动Driver
     3. Driver启动后向Master为当前应用程序申请资源，用于启动excutor
     4. Master找到满足的节点启动Excutor
     5. Executor启动后反向注册给Driver
     6. Driver发送task给worker执行，并监控task执行
     7. Worker将执行情况和结果返回给Driver
  2. 总结

Driver进程是在集群某一台Worker上启动的，在客户端是无法查看task的执行情况的，需要在webui中查看。假设要提交100个application到集群运行,每次Driver会随机在集群中某一台Worker上启动，那么这100次网卡流量暴增的问题就散布在集群上。

4）Standalone两种方式提交任务，Driver与集群的通信包括：

1. Driver负责应用程序资源的申请

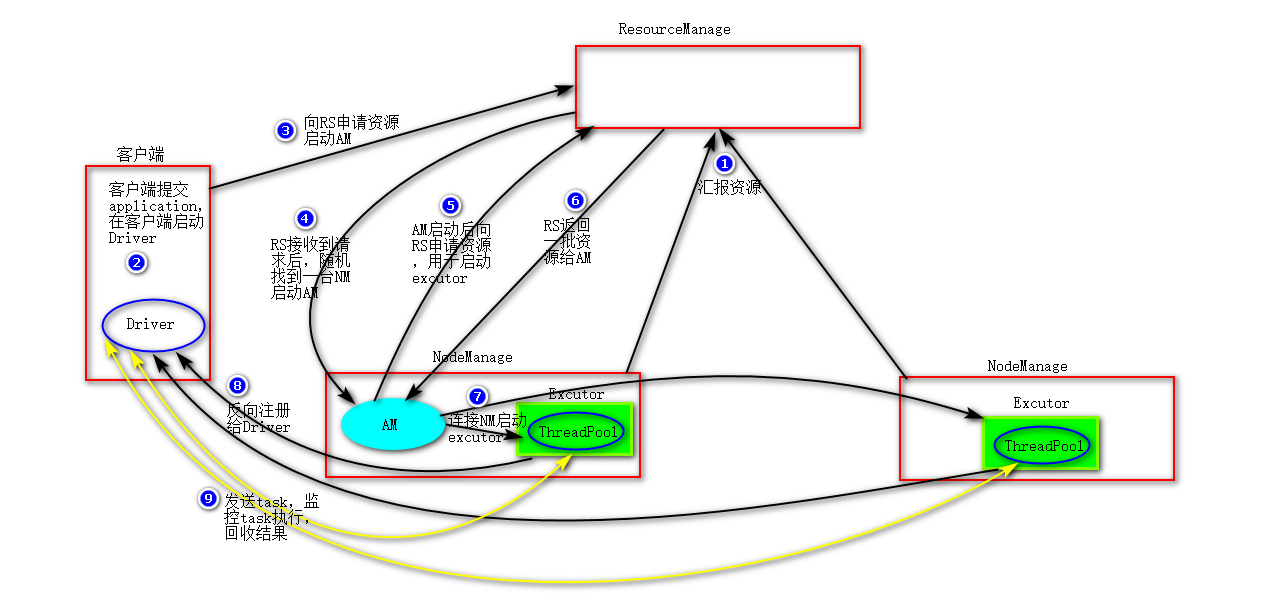
2. 任务的分发。

3. 结果的回收。

4. 监控task执行情况。

1. yarn-client模式

1）流程图



2）流程

①客户端提交一个application，在客户端启动一个Driver进程

②application启动后向RS发送请求，申请启动AM的资源

③RS接收到请求后，随机选择一台NM启动AM（NM相当于worker）

④AM启动后，向RS请求一批container资源，用于启动Excutor

⑤RS找到一批NM返回给AM，用于启动excutor

⑥AM连接NM发送命令启动excutor

⑦excutor启动后，会反向注册给Driver

⑧Driver发送task到Excutor

⑨Excutor将执行情况和结果返回给Driver端

3）总结

Yarn-client模式同样是适用于测试，因为Driver运行在本地，Driver会与yarn集群中的Executor进行大量的通信，会造成客户机网卡流量的大量增加。

4）ApplicationMaster的作用：

1. 为当前的Application申请资源

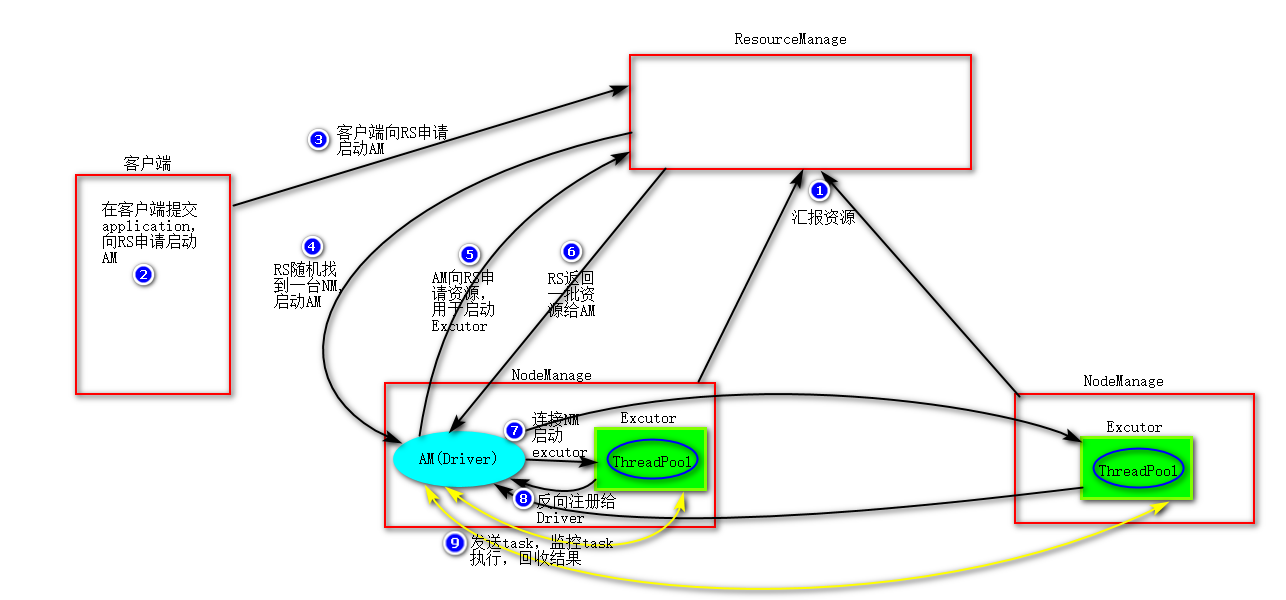
2. 给NameNode发送消息启动Executor。

注意：ApplicationMaster有launchExecutor和申请资源的功能，

并没有作业调度的功能。

1. yarn-cluster模式

1）流程图



2）流程

①客户端提交application，发送请求到RS请求启动AM

②RS接收到请求后随机在一台NN上启动AM

③AM启动后发送请求到RS，请求一批container用于启动

Executor

④RS返回一批NM给AM

⑤AM连接NM，发送请求到NM启动excutor

⑥excutor启动后反向注册到AM所在节点的Driver

⑦Driver发送task给excutor执行

⑧excutor将执行情况和结果返回给Driver

3）总结

Yarn-Cluster主要用于生产环境中，因为Driver运行在Yarn集群中某一台nodeManager中，每次提交任务的Driver所在的机器都是随机的，不会产生某一台机器网卡流量激增的现象，缺点是任务提交后不能看到日志。只能通过yarn查看日志。

4）ApplicationMaster的作用：

1. 为当前的Application申请资源

2. 给NameNode发送消息启动Excutor。

3. 任务调度。

**9.所有算子代码**

（1）transformation算子

1）join：将两个(k1,v)、(k2,w)格式的RDD中key相同的转换成

(k,(v,w))格式的RDD。

代码

|  |
| --- |
| val rdd1: RDD[(String, Int)] =  sc.parallelize(List[(String, Int)](("zhangsan", 18),  ("lisi", 19), ("wangwu", 20), ("zhaoliu", 21)), 1)  val rdd2: RDD[(String, Int)] =  sc.parallelize(List[(String, Int)](("zhangsan", 100),  ("lisi", 200), ("wangwu", 300), ("tianqi", 400)), 3)  val result: RDD[(String, (Int, Int))] = rdd1.join(rdd2) |

2）leftOuterJoin：将两个(k1,v)、(k2,w)格式RDD以leftOuterJoin

左边的RDD的key为基准转换为(k1,(v,Option(w)))。

代码

|  |
| --- |
| val rdd1: RDD[(String, Int)] =  sc.parallelize(List[(String, Int)](("zhangsan", 18),  ("lisi", 19), ("wangwu", 20), ("zhaoliu", 21)), 1)  val rdd2: RDD[(String, Int)] =  sc.parallelize(List[(String, Int)](("zhangsan", 100),  ("lisi", 200), ("wangwu", 300), ("tianqi", 400)), 3)  val result: RDD[(String, (Int, Option[Int]))] =  rdd1.leftOuterJoin(rdd2) |

3）rightOuterJoin:将两个(k1,v)、(k2,w)格式RDD以leftOuterJoin

右边的RDD的key为基准转换为(k2,(Option(v),w))。

代码

|  |
| --- |
| val rdd1: RDD[(String, Int)] =  sc.parallelize(List[(String, Int)](("zhangsan", 18),  ("lisi", 19), ("wangwu", 20), ("zhaoliu", 21)), 1)  val rdd2: RDD[(String, Int)] =  sc.parallelize(List[(String, Int)](("zhangsan", 100),  ("lisi", 200), ("wangwu", 300), ("tianqi", 400)), 3)  val result: RDD[(String, (Option[Int], Int))] =  rdd1.rightOuterJoin(rdd2) |

4）fullOuterJoin: 将两个(k1,v)、(k2,w)格式RDD中所有key转换

为(k,( Option(v), Option(w)))k包含两个RDD中所有相同和不

同的key

代码

|  |
| --- |
| val rdd1: RDD[(String, Int)] =  sc.parallelize(List[(String, Int)](("zhangsan", 18),  ("lisi", 19), ("wangwu", 20), ("zhaoliu", 21)), 1)  val rdd2: RDD[(String, Int)] =  sc.parallelize(List[(String, Int)](("zhangsan", 100),  ("lisi", 200), ("wangwu", 300), ("tianqi", 400)), 3)  val result: RDD[(String, (Option[Int], Option[Int]))] = rdd1.fullOuterJoin(rdd2) |

总结：以上算子join后的分区数与父RDD分区数多的那一个相同。

5）union：合并两个数据集。两个数据集的类型要一致，返回新的RDD

的分区数是合并RDD分区数的总和。

代码

|  |
| --- |
| val rdd1: RDD[(String, Int)] =  sc.parallelize(List[(String, Int)](("zhangsan", 18),  ("lisi", 19), ("wangwu", 20), ("zhaoliu", 21)), 3)  val rdd2: RDD[(String, Int)] =  sc.parallelize(List[(String, Int)](("zhangsan", 100),  ("lisi", 200), ("wangwu", 300), ("tianqi", 400)), 2)  val result: RDD[(String, Int)] = rdd1.union(rdd2) |

6）intersection：取两个数据集的交集，返回新的RDD与父RDD

分区多的一致。

代码

|  |
| --- |
| val rdd1: RDD[String] =  sc.parallelize(List[String]("a","b","c","d"))  val rdd2: RDD[String] =  sc.parallelize(List[String]("a","b","c","f"))  val result: RDD[String] = rdd1.intersection(rdd2) |

7）subtract：取两个数据集的差集，返回subtract右边集合中元素

没在左边集合出现的元素，结果RDD的分区数与subtract前面的

RDD的分区数一致。

代码

v

|  |
| --- |
| val rdd1: RDD[String] =  sc.parallelize(List[String]("a","b","c","d"))  val rdd2: RDD[String] =  sc.parallelize(List[String]("a","b","c","f"))  al result: RDD[String] = rdd2.subtract(rdd1) |

8）mapPartitions：遍历每个partition上的数据。

代码

|  |
| --- |
| val rdd1 = sc.parallelize(List[String]("a", "b", "c", "d", "e", "f", "g"), 3)  val result: RDD[String] = rdd1.mapPartitions(iter => {  val list = ArrayBuffer[String]()  println("创建数据库连接。。。")  while (iter.hasNext) {  list.append(iter.next())  println("插入数据库连接。。。")  }  println("关闭数据库连接。。。")  list.iterator  })  result.count() |

应用场景：数据遍历之后还要继续做一系列操作

9）distinct(map+reduceByKey+map)：去重，先将集合中元素进行map

操作，再进行reduceByKey操作，最后再进行map操作。

代码

|  |
| --- |
| val rdd1: RDD[String] =  sc.parallelize(List[String]("a","c","d","a","d","d"))  val result: RDD[String] = rdd1.distinct() |

10）cogroup：当调用类型（K,V）和（K，W）的数据上时，返回一个

数据集（K，（Iterable<V>,Iterable<W>）），子RDD的分区与父

RDD多的一致。

代码：

|  |
| --- |
| val rdd1: RDD[(String, Int)] =  sc.parallelize(List[(String,Int)](("zhangsan",18),("zhangsan",19),("wangwu",20),("zhaoliu",21)))  val rdd2: RDD[(String, Int)] =  sc.parallelize(List[(String,Int)](("zhangsan",100),("lisi",200),("lisi",300),("tianqi",400)))  val result: RDD[(String, (Iterable[Int],Iterable[Int]))] = rdd1.cogroup(rdd2) |

11）zip：将两个格式为RDD[k]、RDD[v]的RDD组成RDD(k,v)格

式的RDD，分区内元素个数必须相同，分区数必须相同。

代码：

|  |
| --- |
| val rdd1 = sc.parallelize(List[String]("a","b","c","d"))  val rdd2 = sc.parallelize(List[Int](100,200,300,400))  val result: RDD[(String, Int)] = rdd1.zip(rdd2) |

12）zipWithIndex：将RDD中元素和元素下标组成(k,v)格式的RDD

代码：

|  |
| --- |
| val rdd1 = sc.parallelize(List[String]("a","b","c","d"))  val result: RDD[(String, Long)] = rdd1.zipWithIndex() |

（2）action算子

1）foreachPartition：遍历每个partition的数据。

代码：

|  |
| --- |
| val rdd1 = sc.parallelize(List[String]("a", "b", "c", "d", "e", "f", "g"), 3)  rdd1.foreachPartition(iter => {  println("创建数据库连接。。。")  while (iter.hasNext) {  println("插入数据库连接。。。" + iter.next())  }  println("关闭数据库连接。。。")  }) |

应用场景：RDD结果遍历之后不再需要这个RDD

（3）Java代码：

|  |
| --- |
| SparkConf conf = new SparkConf();  conf.setMaster("local");  conf.setAppName("suanzi");  JavaSparkContext jsc = new JavaSparkContext(conf);  jsc.setLogLevel("ERROR");  /\*\*  \* join,leftOuterJoin,rightOuterJoin,fullOuterJoin  \*/  // JavaPairRDD<String, Integer> rdd1 = jsc.parallelizePairs(Arrays.asList(  // new Tuple2<String, Integer>("zhangsan", 18),  // new Tuple2<String, Integer>("lisi", 20),  // new Tuple2<String, Integer>("maliu", 30),  // new Tuple2<String, Integer>("wangwu", 28)));  // JavaPairRDD<String, Integer> rdd2 = jsc.parallelizePairs(Arrays.asList(  // new Tuple2<String, Integer>("zhangsan", 100),  // new Tuple2<String, Integer>("lisi", 200),  // new Tuple2<String, Integer>("wangwu", 300)));  // JavaPairRDD<String, Tuple2<Integer, Integer>> join = rdd1.join(rdd2);  // join.foreach(new VoidFunction<Tuple2<String, Tuple2<Integer, Integer>>>() {  // @Override  // public void call(Tuple2<String, Tuple2<Integer, Integer>> t) throws Exception {  // System.out.println(t);  // }  // });  // JavaPairRDD<String, Tuple2<Integer, Optional<Integer>>> leftOuterJoin = rdd1.leftOuterJoin(rdd2);  //// leftOuterJoin.foreach(new VoidFunction<Tuple2<String, Tuple2<Integer, Optional<Integer>>>>() {  //// @Override  //// public void call(Tuple2<String, Tuple2<Integer, Optional<Integer>>> t) throws Exception {  //// System.out.println(t.\_1);  //// System.out.println(t.\_2);  //// System.out.println(t.\_2.\_1);  //// System.out.println(t.\_2.\_2.orElse(1111111));  //// }  //// });  // JavaPairRDD<String, Tuple2<Optional<Integer>, Integer>> rightOuterJoin = rdd1.rightOuterJoin(rdd2);  // rightOuterJoin.foreach(new VoidFunction<Tuple2<String, Tuple2<Optional<Integer>, Integer>>>() {  // @Override  // public void call(Tuple2<String, Tuple2<Optional<Integer>, Integer>> stringTuple2Tuple2) throws Exception {  // System.out.println(stringTuple2Tuple2);  // }  // });  // JavaPairRDD<String, Tuple2<Optional<Integer>, Optional<Integer>>> fullOuterJoin = rdd1.fullOuterJoin(rdd2);  // fullOuterJoin.foreach(new VoidFunction<Tuple2<String, Tuple2<Optional<Integer>, Optional<Integer>>>>() {  // @Override  // public void call(Tuple2<String, Tuple2<Optional<Integer>, Optional<Integer>>> stringTuple2Tuple2) throws Exception {  // System.out.println(stringTuple2Tuple2);  // }  // });  // JavaPairRDD<String, Integer> union = rdd1.union(rdd1);  /\*\*  \* distinct 去重  \*/  // JavaRDD<String> rdd1 = jsc.parallelize(Arrays.asList("a", "b", "a", "b", "c", "c"));  // rdd1.distinct().foreach(new VoidFunction<String>() {  // @Override  // public void call(String s) throws Exception {  // System.out.println(s);  // }  // });  /\*\*  \* intersection取交集 subtract取差集  \*/  // JavaRDD<String> rdd1 = jsc.parallelize(Arrays.asList("a", "b", "c", "d"));  // JavaRDD<String> rdd2 = jsc.parallelize(Arrays.asList("b", "c", "e", "f"));  // rdd1.intersection(rdd2).foreach(new VoidFunction<String>() {  // @Override  // public void call(String s) throws Exception {  // System.out.println(s);  // }  // });  // System.out.println("=============================");  // rdd1.subtract(rdd2).foreach(new VoidFunction<String>() {  // @Override  // public void call(String s) throws Exception {  // System.out.println(s);  // }  // });  /\*\*  \* mapPartitions  \*/  // JavaRDD<String> rdd1 = jsc.parallelize(Arrays.asList("a", "b", "c", "d"), 2);  // rdd1.mapPartitions(new FlatMapFunction<Iterator<String>, String>() {  // @Override  // public Iterator<String> call(Iterator<String> iter) throws Exception {  // List<String> list = new ArrayList<>();  // String str = "";  // while (iter.hasNext()) {  // str = iter.next();  // list.add(str);  // System.out.println(str);  // }  // return list.iterator();  // }  // }).count();  /\*\*  \* foreachPartition  \*/  JavaRDD<String> rdd1 = jsc.parallelize(Arrays.asList("a", "b", "c", "d"), 2);  rdd1.foreachPartition(new VoidFunction<Iterator<String>>() {  @Override  public void call(Iterator<String> iter) throws Exception {  while(iter.hasNext()){  System.out.println(iter.next());  }  }  }); |

**10.自己写文件，数据有倾斜，过滤掉出现次数最多的单词，统计其他剩余单词的次数。**

|  |
| --- |
| val sc: SparkContext =  SparkSession.builder().master("local").appName("zuoye").  getOrCreate().sparkContext  val words: RDD[(String, Int)] =  sc.textFile("./data/data.txt").  flatMap(\_.split(" ")).map((\_,1)).reduceByKey(\_+\_)  val big: String = words.sortBy(\_.\_2,false).first().\_1  words.filter(!\_.\_1.equals(big)).foreach(println)  sc.stop() |