

# Scala简介

Scala是将面向对象和函数式编程结合在一起的语言，Scala的任务主要基于JVM基础之上，同时Scala语言是基于Java语言开发的。

应用：Spark、Kafka

作者：马丁 奥德斯基

发行时间：2004年推出第一版Scala，然后在2009推出的2.X版本

学习Scala的目的：

比较简单：这个框架开发比较简单，而且相对于Java来说，代码少之又少，开发敏捷。

运行速度快：Scala语言表达能力强，一行相当于Java的多行代码，开发比较快，Scala是静态编译的语言。

可以整合Hadoop生态：能融入Hadoop生态，然后可以使用基于Scala的Spark技术栈，进行Hadoop内的操作，主要还是能无缝对接HDFS。

特点：它是一门多范式，命令行式编程语言，实现面向对象编程和函数式编程的语言，但是Scala还是要基于JVM基础之上，实现代码的开发。

## Scala解释器

1. REPL: Read (取值) ->Evaluation (求值) ->Print (打印) ->Loop (循环)
2. 声明变量两种方式: val 声明的变量是不可变的变量，但是可以调用的使用 var 声明变量是可变变量，并且可以进行调用，在这里官网给我们定义，在声明变量的时候，尽量使用val进行声明，因为我们声明的变量会被调用或者使用，那么在传输（网络）过程中，会发生序列化和反序列化操作，那么有可能会发生数据值被传错的情况，所以尽量用val声明变量。
3. 定义类型时，如果不确定，或者不想写死一个类型可以定义为他们的顶级父类，Any类型
4. 数据类型和操作符

基本数据类型：Byte、Char、Short、Int、Long、Float、Double、Boolean，在scala中也有java的基本数据类型，但是没有java的包装类型

Scala内部提供了一些加强版类型RichInt、RichDouble、RichChar等类型

to方法其实在Int内部是没有这样的方法的，是在他的加强类中RichInt中才有的方法

操作符：和Java类似，比如+、-、\*、/、%以及&，其中有一点要注意，在java中如果进行自增的话，你需要写++，但是在scala中是+=具体值，比如count += 1

5. apply函数（初始化函数）

这个apply函数其实相当于我们在定义一些类的时候，有的类需要new，才能创建，但是在scala编程当中，为了简化使用者的难度，有一些操作，是在内部进行初始化了，相当于说，使用者不需要再去new 类或者对象，而且直接调用，例如“ new 类名() ” 在Scala中直接 " 类名()"

6. if表达式

if表达式的定义:在Scala中，if表达式是有值的，就是if或者else中最后一行语句代表返回值

在条件语句中进行判断的话，那么如果出现返回值不统一的情况下，将取他们的公共父类，进行返回，如果当前if表达式中没有else语句那么就代表else是空值，返回的类型也同样是他们的公共父类，在这里Unit和() 一个意思。

## 7. 块表达式和语句终结符 (;)

在Scala语言当中，它是不需要语句终结符的，默认一行就是一条语句（Java中需要用;终结当前语句）

块表达式：指的是{}中的值，只要被当前这个{}包含的叫块表达式。

## 8. 输入和输出

输出：print和println直接输出

printf 初始化输出（了解）

## 9. 循环 (while、for)

如果跳出循环，有三种方式（变量、return、break）

高级for循环：在for循环中加if守卫的方式，限制循环条件，同时也可以将输出的数据，变成集合（构造集合yield）

## 10. 函数和方法

函数：定义函数 `val f1 = (参数列表) => {函数体}` 函数分为匿名函数和非匿名函数

方法： 方法的关键字，方法名，方法参数，方法返回值类型，方法体

声明方法 `def m1 (x:Int):Int = {x*x}`

在Scala编程中，声明方法后，我们可以将参数设置成函数，因为在Scala语言中，函数是可以通过参数传递到方法内部的，进行使用，如果说想将方法进行参数传递，那么你可以将需要传递的方法，转换成函数，在进行传递到方法内部，而在Java中 方法不能当参数传递

案例：斐波那契数列（递归） `def fab(n:Int):Int={if(n<=1) 1 else fab(n-1)+fab(n-2)}`

## 11. lazy关键字

在Scala中，提供了lazy值的特性，也就是说，如果将一个变量声明为lazy，则只有在第一次使用该变量时候，变量对应的表达式才会发生计算。这种特性针对于特别耗时的计算操作比较有效，比如打开文件进行IO，进行网络IO等。

## 12. 数组（定长和变长）

在Scala中，Array代表的含义和Java类似，也是长度不可变的数组，此外，由于Scala与Java都是运行在JVM之上，双方可以互相调用。

在Scala中，如果需要类似于Java中的ArrayList这种长度可变的集合类，则可以使用ArrayBuffer  
数据基本操作，添加 +=、++=、append，删除 remove，转换 toArray、toBuffer

数据输出：for方式，foreach，反转输出 (reverse)

定长数组：import scala.collection.immutable.\_ (默认导入) 一般在创建数据或者初始化的时候，会使用的数组

变长数组：import scala.collection.mutable.\_ (手动导入) 一般在进行数据累加的时候，会使用，或者进行数据暂时存储的时候会使用

## 13. 数组操作

算方案例：移除第一个负数之后的所有负数

```
val a = ArrayBuffer[Int]
```

```
while(index < arrayLength){  
  if(a(index) > 0){  
    index +=1 }  
  else {  
    if(!flag){flag = true;index +=1}  
    else{a.remove(index);arrayLength -=1}  
  }  
}
```

## 14. Map与Tuple

Map也分为可变与不可变之分，默认还是不可变的Map

通过map里面的get方式获取值的值的时候返回的是Some，为什么？

通过map里面的get方式获取值的值的时候如果key不存在会返回none，不报错，但是如果在.get就会报错，为什么？

## Map

SortedMap和LinkedHashMap

SortedMap 是排序的Map，按照Key进行排序

LinkedHashMap 是一个可变的Map，同时它会记录数据插入顺序

## Tuple (元组)

Scala中的重要特性，Tuple可以存储任何类型的值，同时也可以存储Tuple，并且取值比较简单，操作方便

## ZIP (拉链)

## 面向对象 (重点)

### 1. 面向对象编程之类

伴生类：和object同名的，叫做伴生类，而且在伴生类中定义私有的(private)方法和属性，都可在伴生对象中使用，当然也可调用object私有的方法和属性，如果要更严格的定义方法或属性那么需要用private[this]声明。

构造器：分为主构造器和辅助构造器，辅助构造器在定义完成后，第一步就需要实现主构造器的构造参数，然后才可以去实现辅助构造器，并且在Scala中辅助构造器和主构造器相当于Java中的无参方法和有参方法。

### 2. 面向对象编程之对象

如果在对象当中进行定义变量和方法，那么我们使用的时候，不需要在New当前的这个对象了，直接使用对象中的变量和方法，也就是说，不需要像类一样实例化。

对象可以实现序列化功能，不需要继承序列化接口

在对象中定义的方法和属性，会初始化一次，下次不需要在进行创建和初始化，直接使用，同时对对象就相当于静态方法一样

一般定义的一些常量和connection连接使用对象进行声明或者创建

伴生对象：和class类同名的，叫做伴生对象，可以相互访问私有的属性和方法

让Object继承抽象类

Object可以继承抽象类，同时实现抽象方法，但是在object中不可以进行传参，也就是不能实现构造器

Apply初始化

在我们操作时，比如用Object进行定义的方法属性都会进行默认的初始化操作，也就是说会默认调用.apply()方法

### 3. 面向对象编程之继承 (extends)

在Scala中继承特性和Java一样，都是子类继承父类，拥有父类的方法和属性，当然私有的除外，同时也可去覆盖父类的方法和属性 (Override) 如果想调用父类方法或属性，也可以用super进行调用

父类子类类型转换 (asInstanceOf和isInstanceOf)

类型转换，可以进行子类与父类之间转换，同时也可以进行基本数据类型之间的转换操作，而且可以先进行判断，在进行转换操作

getClass和ClassOf 相比上面的转换，它更精准一些，但是本质是一样的，都是转换类型

抽象类：抽象类不可以被实例化

抽象方法：没有被实现方法，叫做抽象方法

抽象变量：定义变量，但是没有给出具体的初始值，则称为抽象变量

### 4. 面向对象编程之trait (接口)

Scala中的Trait是一个特殊的概念

Trait与Java中的接口类似，但是进行继承的时候，不使用implement，而是使用extends进行继承，一般在Spark的代码中，主要是用于打印日志，如果进行继承接口 (Trait) 用extends，那么如果继承多个trait，用with。

Trait内部可以定义变量 (抽象变量) ,方法 (抽象方法) ，可以实现多重继承，使用with

Trait继承class，同时继承class之后，又可以被继承

## 函数式编程 (重点)

定义：(x:Int,y:Int)=>{x+y}

高阶函数：map、filter、reduce、sortWith、sortBy、sorted、flatMap、flatten等等

## 函数式编程 (重点)

1. 闭包：最简洁的解释，函数在变量不处于有效作用域时，还能够对其变量进行访问，即为闭包

2. 柯里化：将原来接收两个参数的一个函数，转换为两个函数，第一个函数接收原先的第一个参数，然后返回第二个函数，在接收第二个参数得到结果值

### 3. 集合操作 (重点)

Scala中的集合体系包括：Iterator (Iterable) 、Seq、Set、Map。

Scala集合是分为两大类的，可变集合 (mutable) ，不可变集合 (immutable)

Seq：包含了List、ArrayBuffer、Range，其实Range其实它也代表一个序列，通常可以使用 1 to 10

List 方法 head、tail、last

ListBuffer

LinkedList

Set: 无序 不重复

HashSet分为可变和不可变两种，需要导入不同的包

LinkedHashSet: 记录插入顺序

SortedSet: 排序Set

#### 4. ScalaWC

```
// 读取本地文件
val lines = Source.fromFile("D:\\hello.txt").getLines()
// 切分数据
val arr: Iterator[Array[String]] = lines.map(_.split(" "))
// 过滤空格
val filtered = arr.filter(_ != " ")
// 压平
val flat: Iterator[String] = filtered.flatten
// 聚合操作
val tuples: Iterator[(String, Int)] = flat.map(x=>(x,1))
// 分组
val group: Map[String, List[(String, Int)]] =
tuples.toList.groupBy(_._1)
println(group)
// 求值
//val sum = group.map(x=>(x._1,x._2.size))
val sum = group.mapValues(_._2.size)
// 排序
val sum2 = sum.toList.sortBy(_._2).reverse
// 输出
println(sum2)
```

#### 5. 特殊 par (多线程)

```
val arr = Array(1,2,3,4,5,6,7,8,9,10)
```

```
arr.fold(0)(+)
```

第一个参数代表初始值，第二个参数代表逻辑计算表达式

```
arr = 1,2,3,4,5
```

```
fold(0)(+)
```

```
((0+1)+2)+3
```

par代表多线程

例如 arr.par.reduce(+) 代表多线程 arr.reduce 代表单线程

交集 intersect

并集 union

差集 diff

```
List(1,2,3,4,5) '=='
```

用递归方式来给List中每个元素都加上前缀，并打印加上前缀的元素

## 模式匹配 (重点)

Scala是没有Java中的switch case 语法的，相对应的，Scala提供了更强大的match case语法，也就是模式匹配

首先Scala的模式匹配和Java有很大不同，在匹配过程中，可以匹配多种类型，如List，Array，String等等，并且每次匹配到最符合条件的就会跳出匹配

对类型进行模式匹配

数组和集合模式匹配

样例类模式匹配

样例类：用case class声明的类，叫样例类，样例类不需要New，默认实现了序列化接口

Option类型：

通过map里面的get方式获取值的值的时候返回的是Some，为什么？

因为Some继承自Option，同时获取数据返回值的时候只有两个结果一个就是Some一个就是None 不会有Null

通过map里面的get方式获取值的值的时候如果key不存在会返回none，不报错，但是如果在.get就会报错，为什么？

如果当前值不存在，那么还继续.get取值的话，通过源码发现，会抛个异常

```
def get = throw new NoSuchElementException("None.get")
```

偏函数：被包在花括号内没有match的一组case语句是一个偏函数，它是PartialFunction[A,B] 的一个实例，A代表参数类型，B代表返回值类型

## 泛型

上边界：<: 我们在定义泛型的时候，可以进行泛型的边界 限定，而不是任意类型，比如，我们可能要去某个泛型类型，他必须是某个类的子类，这样程序中就可以放心的调用类型方法或者属性

上边界

```
/**
 * 上边界
 */
class CmpInt(a:Int,b:Int) {
  def bigger = if (a>b) a else b
}

// 进行了上界限定
class CmpComm[T <: Comparable[T]](o1:T,o2:T){
  def bigger = if(o1.compareTo(o2)>0) o1 else o2
}

object CmpTest{
  def main(args: Array[String]): Unit = {
    val int = new CmpInt(1,2)

    // 上界限定就会报错，其实是Int源码中没有继承Comparable
    // val cmpInt = new CmpComm(1,2)
    //
    val cmpInt2 = new CmpComm(Integer.valueOf(1),Integer.valueOf(2))
    new CmpComm[Integer](11111111,22222222)
```

```
}  
}
```

下边届:

上下边界:


## 泛型

上边界: <: 我们在定义泛型的时候, 可以进行泛型的边界 限定, 而不是任意类型, 比如, 我们可能要去某个泛型类型, 他必须是某个类的子类, 这样程序中就可以放心的调用类型方法或者属性

上边界

```
/**  
 * 上边界  
 */  
class CmpInt(a:Int,b:Int) {  
    def bigger = if (a>b) a else b  
}  
  
// 进行了上界限定  
class CmpComm[T <: Comparable[T]](o1:T,o2:T){  
    def bigger = if(o1.compareTo(o2)>0) o1 else o2  
}  
  
object CmpTest{  
    def main(args: Array[String]): Unit = {  
        val int = new CmpInt(1,2)  
  
        // 上界限定就会报错, 其实是Int源码中没有继承Comparable  
        // val cmpInt = new CmpComm(1,2)  
        //  
        val cmpInt2 = new CmpComm(Integer.valueOf(1),Integer.valueOf(2))  
        new CmpComm[Integer](11111111,2222222)  
    }  
}
```

下边届: >: 是指泛型类型必须是某个类的父类 例如: T >: A

1572485559831

视界 (View Bounds) : 可以在转换类型中, 使用隐式转换操作, 完成父类子类的转换

```
package day04  
  
/**  
 * 泛型 视界 (context)  
 */  
class Personed(val name:String){  
    def sayHello = println("Hello, I'm "+name)
```

```

    def makePerson(p: Personed): Unit = {
        p.sayHello
    }
}

class Stu(name: String) extends Personed(name)

class Dog(val name: String)

class Party[T <% Personed](p1: T) {
    def play = p1.makePerson(p1)
}

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

        import day04.dogTest.dog2Person

        val stu = new Stu("zhangsan")

        val lisi = new Personed("lisi")

        val dog = new Dog("大黄")

        val py = new Party(dog)

        py.play

    }
}

package day04

/**
 * 隐式转换（为了和人交朋友）
 */
object dogTest {
    implicit def dog2Person(dogs: Object): Personed = {
        // 先进行判断
        if (dogs.isInstanceOf[Dog]) {
            val _dog = dogs.asInstanceOf[Dog]
            new Personed(_dog.name)
        } else {
            Nil
        }
    }
}

```

上下边界（Context Bounds）：它其实和视界没什么区别，都是可以实现隐式转换操作，也可以实现隐式参数的传递和调用

```
package day04
```



```

/**
 * 上下边界
 */
class contextBounds[T:Ordering](val number1:T,val number2:T){
  def max(implicit ordering: Ordering[T])=
    if(ordering.compare(number1,number2)>0) number1 else number2
}

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


    val maxs = new contextBounds(1,2)
    println(maxs.max)
  }
}

```

协变(+T)和逆变(-T)

协变：如果我们用的是[+T]那么代表协变，也就是说（例子）Master 是Professional的父类，那么Card[Master]也会是Card[Professional]的父类，这就是协变

逆变：如果我们是[-T]这样的泛型，那么此时Master还是Professional父类，但是Card[Master]就变成了Card[Professional]的子类了

1572489796655

1572489814547

## 隐式转换（重点，难点）关键字：Implicit

隐式转换：Scala会自动使用隐式转换函数，隐式转换和其他函数区别主要是语法上，隐式转换需要用implicit开头，而普通函数是没有的。

作用：隐式转换可以帮助开发者，实现不同类型的转换（自动），如果Scala内部无法自动实现转换操作，那么开发者可以自己定义隐式转换的方法或者参数，实现类型转换或隐式值的注入

隐式转换发生的时机：

1. 调用某个方法，但传入的参数类型不匹配，可以使用隐式转换调用
2. 使用某个类型的对象，调用某个方法，但是当前对象内没有需要使用的方法，那么需要隐式转换进行调用

隐式参数：所谓隐式参数，指的是在函数或者方法中，定义一个用implicit修饰的参数，此时，Scala会找到指定的implicit修饰的类型，进行隐式值的注入

什么叫隐式转换：如果当前类或者类型中不存在能换行的值，或者类型，那么需要转换的话，导入隐式转换或者隐式转换参数即可，但是如果scala内部没有提供这些类型和类，需要自己实现。如果当前类想使用其他类的方法或者属性，就必须导入隐式转换。

## Actor（了解）

Scala提供了Actor这样的多线程编程，类似于Java的Thread和Runnable一样，我们只需要写出Actor的act方法，即可实现线程，类似于Java的run方法，Actor已经被弃用了（在Akka时底层实现还是Actor，当到Spark2.0的时候Akka和Actor都被弃用，使用Netty模型代替）

start是启动方法

receive方法进行数据传输和交互

发送消息的方式：

! 代表异步发送无返回值

!? 代表同步发送无返回值

!! 代表异步发送有返回值 返回值类型 Future

## Akka

ActorSystem对象：单例对象，可以通过ActorSystem来创建Actor

PreStart方法：该方法是在receive方法之前键执行，然后它只会执行一次，相当于初始化

Receive方法：该方法在一直不停的循序接收发送消息

模拟Spark底层通信

Master Worker

Master实现ActorSystem对象，注入对应Actor类，然后对接Worker实现通信，在Spark（1.6之前）中，底层通信就用的Akka实现的，Master检测心跳（Worker发来的workerId和时间戳），判断当前Worker是否存活，如果发送失败，那么将重试，如果重试失败，代表这个Worker挂了，Master就不会给他分配任务。

Akka内部的Actor执行原理

1572507283379

## Netty

实现Netty模型的Server和Client通信

创建Server Client ServerHandler ClientHandler

## Spark

---

spark的应用和对比

1572593720645

## Spark简介

1. 快，相比较Hadoop的MapReduce，Spark基于内存计算要快100倍，主要是内部实现了DAG计算执行引擎
2. 易用，Spark支持Java、Scala、Python、R、SQL的API进行代码开发
3. 通用，Spark提供了统一的解决方案，Spark可以用于批处理，交互式查询，实时流计算，机器学习，图计算
4. 兼容性，可以支持Hadoop，Hbase，Hive，Yarn等等

Spark迭代式计算

Hadoop和Spark都是分布式计算模型，但是区别在于Spark基于内存计算，并且可以多次进行迭代计算，但是Hadoop的MR是能进行一次的任务计算，一次任务分为Map和Reduce两端，而Spark可以利用DAG执行计算引擎，进行多次任务迭代，直到得到最终结果值。

## Spark的运行模式

Local：用在本地测试

Standalone：分布式集群测试

Yarn：基于Yarn进行资源调度，测试环境

Mesos：资源调度测试环境

## Spark-WC

```
./bin/spark-submit --class day06.SparkWC --master spark://node1:7077 --executor-memory 512m --total-executor-cores 2 original-sz1901-1.0-SNAPSHOT.jar
```

## 名词解释 (Standalone)：(重要)

Client：客户端，负责提交作业到集群

Master：代表集群的主节点，负责接收Client提交的作业，管理资源调度，负责任务分配

Worker：代表集群中的守护进程，负责创建任务进程，定时向Master汇报，起到与Driver交互的作用

Driver：Spark启动的主进程，负责任务的提交，同时和Master、Worker进行交互，更重要一点负责保存元数据信息

Executor：执行作业的容器，集群可以有多个Executor，每个Executor都是独立的和Driver进行交互，里面主要是封装资源和Task

Stage：阶段，Spark的任务会包含多个阶段

Task：一个Stage中包含多个Task，Task数量=Partition数量，一个Task就是用一个任务

DAGScheduler：主要是负责任务划分（Stage），根据Partition的数量确定Task数量，然后提交所有Task到TaskSet中

TaskSet：是Task的容器，包含多个Task

TaskScheduler：负责和Executor进行交互，然后将Task提交至Executor运行

Job：一个程序的任务（在Spark中，一个Action算子触发一个Job）

SparkContext：整个任务的执行入口（相当于任务触发位置）

## RDD (重点)

## Resilient Distributed Dataset (RDD)

叫做弹性分布式数据集，是Spark中数据抽象，它代表一个不可变，可分区，元素可并行计算的数据集合

RDD弹性：

### 1) 自动进行内存和磁盘数据存储的切换

Spark优先把数据放到内存中，如果内存放不下，就会放到磁盘里面，程序进行自动的存储切换

### 2) 基于血统的高效容错机制

在RDD进行转换和动作的时候，会形成RDD的Lineage依赖链，当某一个RDD失效的时候，可以通过重新计算上游的RDD来重新生成丢失的RDD数据。

### 3) Task如果失败会自动进行特定次数的重试

RDD的计算任务如果运行失败，会自动进行任务的重新计算，默认次数是4次。

### 4) Stage如果失败会自动进行特定次数的重试

如果Job的某个Stage阶段计算失败，框架也会自动进行任务的重新计算，默认次数也是4次。

### 5) Checkpoint和Persist可主动或被动触发

RDD可以通过Persist持久化将RDD缓存到内存或者磁盘，当再次用到该RDD时直接读取就行。也可以将RDD进行检查点，检查点会将数据存储到HDFS中，该RDD的所有父RDD依赖都会被移除。

### 6) 数据调度弹性

Spark把这个JOB执行模型抽象为通用的有向无环图DAG，可以将多Stage的任务串联或并行执行，调度引擎自动处理Stage的失败以及Task的失败。

### 7) 数据分片的高度弹性

可以根据业务的特征，动态调整数据分片的个数，提升整体的应用执行效率。

存储的弹性：内存与磁盘的

自动切换容错的弹性：数据丢失可以

自动恢复计算的弹性：计算出错重试机制

分片的弹性：根据需要重新分片

总结：

RDD是一个逻辑概念，一个RDD中有多个分区，一个分区在Executor节点上执行的时候，就是一个迭代器。

一个RDD有多个分区，一个分区肯定在一台机器上，但是一台机器可以有多个分区，我们要操作的是分布多台机器上的数据，而RDD相当于数据的一个代理，对RDD操作其实就是对每个分区进行操作，就是对每台机器上的迭代器进行操作，因为迭代器引用着我们要操作的数据

对RDD操作其实就是对每个分区进行操作，也就是对数据进行操作

RDD的特性：

A list of partitions

一系列分区是个数据集

A function for computing each split

每个分区都作用着一个函数

A list of dependencies on other RDDs

RDD之间有依赖关系

Optionally, a Partitioner for key-value RDDs (e.g. to say that the RDD is hash-partitioned)

如果RDD里的数据是K-V形式的，那么会作用着HashPartitioner分区器

Optionally, a list of preferred locations to compute each split on (e.g. block locations for an HDFS file)

如果是读取HDFS文件，那么会有一个最优位置

Spark在进行任务调度的时候，会尽可能的将计算任务分配到所要处理数据的存储位置

## 深度剖析WC（重点）



## RDD的算子操作（练习）

如果在操作数据的时候，返回的是结果值，在进行操作的话就是Scala的方法，如果返回的是RDD，在进行操作就是RDD的算子

## RDD算子（重点）

collect（特殊算子）

应用场景：一般用于读取本地的字典文件（公司会有自己的过滤词库，过滤规则库）才会使用，在数据结果输出的时候，不会用它，会使用其他算子代替，比如foreach（保存到第三方数据库）

创建RDD方式

1. 并行化创建RDD（parallelize、makeRDD）
2. 读取本地或者HDFS文件创建RDD
3. RDD转换创建RDD
4. 通过外部数据源创建RDD

Transformation（转换算子）

如果返回值是RDD的算子，那么它肯定是一个转换算子，同时，如果想触发转换算子的数据输出，必须有Action算子

map、filter、flatMap、groupByKey、groupBy、reduceByKey、sortByKey、sortBy、join、aggregateByKey、distinct、union、cogroup、MapPartitions、coalesces、Repartition、mapPartitionsWithIndex

Action（行动算子）

如果返回值不是RDD，是其他返回值类型，那么就是一个Action执行算子，它会触发转换算子的数据加载

aggregate、reduce、count、take、sum、foreach、collect、collectAsMap、saveAsTextFile

countByKey

触发Shuffle算子：一系列的by或者bykey, cogroup, distinct、Repartition、countByKey

更多算子了解：<http://homepage.cs.latrobe.edu.au/zhe/ZhenHeSparkRDDAPIExamples.html>

## RDD的持久化

cache、persist

作用：可以对公共代码块进行持久化，复用代码，提高效率，同时持久化机制，一定要触发Action算子，才会有效

cache和persist主要在于使用过程中，如果内存充足，可以直接使用cache，反之要选择其他级别的话，那么用persist进行级别更换，推荐使用内存+磁盘

如果持久化使用结束，程序依然运行，那么需要释放掉缓存，节省空间，用 unpersist()

## RDD的共享变量（重点）

### 1. 广播变量（Broadcast）

广播变量可以将Driver创建的变量广播到每个节点的executor中，作为缓存使用，然后每个Task不需要每次都去拉取Driver的变量，这样减少的大量网络IO和磁盘IO，节省时间，减低资源消耗，同时减少每个Task的冗余性，提升效率

释放广播变量 unpersist

使用场景：比如在本地有字典文件，需要读取，并且不是太大，那么此时可以使用广播变量将其广播到executor内存中，供给每个Task进行使用，减少数据冗余性，提高效率

### 2. 累加器（Accumulator）

累加器可以在每个节点上面进行Task的值，累加操作，有一个Task的共享性操作

新版累加器使用步骤： 1. 创建累加器 2. 注册累加器 3. 使用累加器

自定义累加器：

自定义累加器，可以任意累加不同类型的值，同时也可以在内部分进行计算，或者逻辑编写，如果继承自定义累加器，那么需要实现内部的抽象方法，然后在每个抽象方法内部去累加变量值即可，主要是在全局性累加起到决定性作用。

## Spark编程-----二次排序和分组取TopN

## RDD的宽窄依赖

宽依赖：每一个父RDD的Partition中的数据，都可能传输到子RDD的每个Partition中，这种错综复杂的关系，叫宽依赖


宽依赖划分依据：Shuffle

窄依赖：一个RDD对它的父RDD，只有一个一对一的依赖关系，也就是说，RDD的每个Partition，仅仅依赖于一个父RDD的Partition，一对一的关系叫窄依赖

窄依赖划分依据：没有Shuffle

Join有一个特殊情况，虽然Join是Shuffle算子，但是也会触发窄依赖

例如：

1573009113592

## 血缘

父RDD与子RDD直接存在依赖关系，这种依赖关系叫血缘，同时通过血缘关系，可以达到容错的机制（RDD之间的容错）

## 案例：基站解析案例

根据用户产生日志的信息,在那个基站停留时间最长

19735E1C66.log 这个文件中存储着日志信息

文件组成:手机号,时间戳,基站ID 连接状态(1连接 0断开)

lac\_info.txt 这个文件中存储基站信息

文件组成 基站ID, 经,纬度

在一定时间范围内,求所用户经过的所有基站所停留时间最长的Top2


思路:

- 1.获取用户产生的日志信息并切分
- 2.用户在基站停留的总时长
- 3.获取基站的基础信息
- 4.把经纬度的信息join到用户数据中
- 5.求出用户在某些基站停留的时间top2

## 案例：统计某时间段学科访问量TopN

统计每个时间段内的学科访问量Top2

## Spark任务提交流程（重点）

1573027095147


## Checkpoint（检查点）

Checkpoint相当于缓存，但是他保存的位置是磁盘，并且是永久性保存，会保存到HDFS，这个检查点，比较持久化来说，数据更安全，但是效率很低，一般在程序中设置的位置较少，没有持久化效率高

功能可以在集群运行时候，如果发生了宕机情况，可以通过检查点恢复检查点数据

应用场景：在特别复杂的RDD依赖中，可以使用检查点机制，因为依赖过多，可能会发生集群宕机情况，所以在适合的位置设置检查点是有必要的

## Executor和Task内部原理

1573091966552

## 自定义排序

如果当前的算子，比较排序不满足我们的比较的需求，那么我们可以自己定义比较器，继承Ordered接口，实现比较器的方法，即可

## JDBC连接

如果使用JDBC连接操作的话，那么必须要用到foreachPartition，保证每个分区启动一个连接Connection

## 自定义分区 (Partitioner)

自定义分区器，可以解决HashPartition分区的hash碰撞，用自定义的分区可以减少数据倾斜，提高每个Task的运行效率，并且平均分配数据，在加载自定义分区的时候，可以使用支持自定义分区的算子，如reduceByKey，或者partitionBy

## Stage的划分 (重要)

## 案例解析

这些数据是用户访问所产生的日志信息

http.log是用户访问网站所产生的日志,这些数据的产生是通过后台js埋点所得到的数据

数据组成基本上是:时间戳,IP地址, 访问网址, 访问数据 浏览器信息的等

ip.txt是 ip段数据 记录着一些ip段范围对应的位置

需求: 通过log日志，解析IP，然后根据IP字典，进行统计，分析哪个省份访问量最多，取Top5

```
/**
 * 把ip转换为long类型 直接给 125.125.124.2
 * @param ip
 * @return
 */
def ip2Long(ip: String): Long = {
    val fragments: Array[String] = ip.split(".")
    var ipNum = 0L
    for (i <- 0 until fragments.length) {
        //| 按位或 只要对应的二个二进制位有一个为1时，结果位就为1 <<左位移
        ipNum = fragments(i).toLong | ipNum << 8L
    }
    ipNum
}
```

## Spark-SQL

集成性

Spark SQL允许使用DataFream API进行Spark程序的结构化数据处理，支持Java、Scala、Python、R语言操作

统一数据源



Spark提供了多种数据源，比如，Hive、MySQL、JSON、Parquet、CSV等

集成Hive

SparkSQL支持HiveQL语法以及Hive UDF，也可以操作Hive数仓。

连通性

支持DBC和ODBC的连接

## Spark-SQL使用

### DataFrame (重点)

是Spark基本的API，在Spark1.3推出后，得到广泛应用，主要针对于那些对Spark Core操作比较繁琐的代码，实现SQL语句的开发，同时Spark在推出DataFrame后，到spark1.6以后又推出了Spark的DataSet

```
DataFrame = Dataset[Row]
```



1573444784813

共性：

- 1、RDD、DataFrame、Dataset都是spark平台下的分布式弹性数据集，为处理超大型数据提供便利；
- 2、三者都有惰性机制。在进行创建、转换时（如map方法），不会立即执行；只有在遇到Action时(如foreach)，才会开始遍历运算。极端情况下，如果代码里面仅有创建、转换，但后面没有在Action中使用对应的结果，在执行时会被直接跳过；
- 3、三者都有partition的概念，进行缓存(cache)操作、还可以进行检查点（checkpoint）操作；
- 4、三者有许多相似的函数，如map、filter，排序等；

### RDD转换DataFrame (重点)

为什么要转换？

其实我们可以想象一下，如果使用RDD读取HDFS数据过来，那么能用SQL直接操作，那就相当简单了，并且很符合大众化开发，同时DF执行效率要高于RDD的方式。

转换DF两种方式：1. 编程接口方式 2. 反射方式

使用的方式：首先创建DF，然后根据DF创建临时表，再去执行SQL语句

### DataFrame的数据源 (重点)

读取（Load）：如果使用load的方式，那么不在任何的修改前提下，默认读取的是Parquet文件数据格式，想要修改读取的数据源，需要使用format方法，来改变读取数据源

存储（Save）：使用Save的话，默认存储格式是Parquet格式，如果改变格式，需要使用format方法，同时也可以存储MySQL数据库

### SparkSQL的内置函数 (重点)

导入: import org.apache.spark.sql.functions.\_

Action

```
df1.count

// 缺省显示20行
df1.union(df1).show()
// 显示2行
df1.show(2)
// 不截断字符
df1.toJSON.show(false)
// 显示10行, 不截断字符
df1.toJSON.show(10, false)

// collect返回的是数组, Array[org.apache.spark.sql.Row]
val c1 = df1.collect()

// collectAsList返回的是List, List[org.apache.spark.sql.Row]
val c2 = df1.collectAsList()

// 返回 org.apache.spark.sql.Row
val h1 = df1.head()
val f1 = df1.first()

// 返回 Array[org.apache.spark.sql.Row], 长度为3
val h2 = df1.head(3)
val f2 = df1.take(3)

// 返回 List[org.apache.spark.sql.Row], 长度为2
val t2 = df1.takeAsList(2)
```

Transformations

```
// map、flatMap操作（与RDD基本类似）
df1.map(row=>row.getAs[Int](0)).show

case class Peoples(age:Int, names:String)

val seq1 = Seq(Peoples(30, "Tom, Andy, Jack"), Peoples(20, "Pand, Gate, Sundry"))
val ds1 = spark.createDataset(seq1)
val ds2 = ds1.map(x => (x.age+1, x.names))
ds2.show

val ds3 = ds1.flatMap{ x =>
  val a = x.age
  val s = x.names.split(",").map(name => (a, name.trim))
  s
}
ds3.show

// filter
df1.filter("sal>3000").show
```

## Spark的两种操作风格（重点）

SQL风格：这种风格和咱们操作MySQL、Hive的语句比较类似，而且写起来也比较简单易懂，同时SQL风格完全支持MySQL的语法结构，并且也可以使用自定义UDF。

使用方式：通过SQLContext对象来去使用SQL方法，将SQL放入方法内，进行运行

例如：sqlContext.sql("select date,sum(memory) from user group by date").show()

DSL (Domain Specific Language) 风格：DSL风格属于Spark的特定领域语法，可以将SQL风格内的关键字，拆解，单独使用，这种方式操作比较灵活，比较适合逻辑比较复杂的业务场景。

使用方式：通过DF进行方法调用，直接使用

例如：df.groupBy(col("date")).agg(sum("memory")).show()

## SparkSession创建方式

## Spark的开窗函数

row\_number() over()

rank() over()

dense\_rank() over()

## DataSet (重点)

Dataset-1.6版本后推出的全新特性

Dataset = RDD[case class].toDS

- 1、Dataset和DataFrame拥有完全相同的成员函数，区别只是每一行的数据类型不同；
- 2、DataFrame 定义为 Dataset[Row]。每一行的类型是Row，每一行究竟有哪些字段，各个字段又是什么类型都无从得知，只能用前面提到的getAs方法或者模式匹配拿出特定字段；
- 3、Dataset每一行的类型都是一个case class，在自定义了case class之后可以很自由的获得每一行的信息；

DataSet分为强弱类型

强类型：使用case class这种方式，自己确定类型，叫强类型（表现形式强）

弱类型：使用统一的Row来封装类型参数，这个时候，你在取值的时候，很难发现数据类型（表现形式弱）

三者之间的转换：

DataFrame/Dataset 转 RDD：

```
df.rdd.foreach(row=> println(row.getAs[String](0)))
ds.rdd.foreach(println)
```

RDD转换为DataFrame：

```
要导入隐式转换
rdd.toDF()
```

RDD 转 Dataet：

要导入隐式转换  
核心是 需要设置样例类的类型 使用case class方式进行转换操作  
rdd.toDS()

Dataset 转 DataFrame:


可以将DS封装的case class转换为Row  
ds.toDF

DataFrame 转 Dataset:

将DF转换为DS, 需要设置case class  
df.as[Coltest2]

三者之间创建临时表或者视图的方法

```
ds2.createOrReplaceTempView() // 如果当前的临时视图存在, 那么将其覆盖
ds2.createTempView() // 创建临时视图
df.registerTempTable() // DF老版本创建临时表
```

1573541030185

## 自定义函数 (UDF) (重点)

自定义函数, 其实是根据自己的业务, 设置函数的条件, 如果当前Spark中没有相关的条件函数, 那么我们就必须要自定义一个函数, 来解决我们的业务开发问题。

使用方式:

首先创建函数 (方法), 注册当前的函数, 如果是方法的话, 需要转换成函数, 使用函数 (放在SQL里面进行使用), 我们在注册函数的时候, 那么需要设置函数名

UDAF (了解)

## 案例:

需求:

1. 统计每天每个搜索词用户访问量降序排序
2. 最后输出结果为('日期\_搜索词',UV)
3. 统计分析时候, 我会给你匹配词库, 根据匹配词库, 进行匹配符合规则的数据, 不符合将其过滤

```
val queryFilter = Map(
  "city" -> List("beijing"),
  "platform" -> List("android"),
  "version" -> List("1.0")
)
```

可以使用RDD进行过滤操作, 但是最终统计结果必须用SQL (DSL)

# Spark Streaming介绍

Spark Streaming是一个实时流处理里计算平台，主要是针对流式数据进行统计分析

Spark Streaming针对流数据处理的框架，它的底层实现，其实也是基于Spark Core，也有 Spark Core 的相关概念，比如RDD的容错性，在Spark Streaming中，它的底层其实就在RDD上面实现了一种封装，将之前处理的RDD概念，转换成流的概念（DStream）


流处理组件：

对比	延迟性	吞吐量	计算方式	扩展性	使用率
SparkStreaming	低延迟	高	离线+实时	图计算/机器学习	频繁
Flink	无	高	离线+实时	图计算/机器学习	居多
Storm	无	低	实时	不支持	历史抛弃

## 实时流

1574044612780

## Spark Streaming 基本工作原理

1574045835725

## 入门案例

注意：在我们本地进行测试跑任务的时候，我们设置执行的批次时间，同时也要注意一点，线程数一定要大于等于2，不能少于两个线程，如果少于两个线程，会发生没有数据结果的情况，这样统计没有意义，防止饥饿情况

## SparkStreaming基本操作

输出方法：

Output Operation	Meaning
<b>print()</b>	在运行流应用程序的 driver 节点上的DStream中打印每批数据的前十个元素。这对于开发和调试很有用。
Python API 这在 Python API 中称为 <b>pprint()</b> 。	
<b>saveAsTextFiles(prefix, [suffix])</b>	将此 DStream 的内容另存为文本文件。每个批处理间隔的文件名是根据 前缀和 后缀: "prefix-TIMEIN_MS[.suffix]"_ 生成的。
<b>saveAsObjectFiles(prefix, [suffix])</b>	将此 DStream 的内容另存为序列化 Java 对象的 <code>SequenceFiles</code> 。每个批处理间隔的文件名是根据 前缀和 后缀: "prefix-TIMEIN_MS[.suffix]"_ 生成的。
Python API 这在Python API 中是不可用的。	
<b>saveAsHadoopFiles(prefix,</b>	将此 DStream 的内容另存为 Hadoop 文件。每个批处理间隔的

Output Operation	Meaning
Python API 这在Python API 中是不可用的。	
<b>foreachRDD</b> ( <i>func</i> )	对从流中生成的每个 RDD 应用函数 <i>func</i> 的最通用的输出运算符。此功能应将每个 RDD 中的数据推送到外部系统，例如将 RDD 保存到文件，或将其通过网络写入数据库。请注意，函数 <i>func</i> 在运行流应用程序的 driver 进程中执行，通常会在其中具有 RDD 动作，这将强制流式传输 RDD 的计算。

转换方法：

Transformation (转换)	Meaning (含义)
<b>map</b> ( <i>func</i> )	利用函数 <i>func</i> 处理原 DStream 的每个元素，返回一个新的 DStream。
<b>flatMap</b> ( <i>func</i> )	与 map 相似，但是每个输入项可用被映射为 0 个或者多个输出项。。
<b>filter</b> ( <i>func</i> )	返回一个新的 DStream，它仅仅包含原 DStream 中函数 <i>func</i> 返回值为 true 的项。
<b>repartition</b> ( <i>numPartitions</i> )	通过创建更多或者更少的 partition 以改变这个 DStream 的并行级别 (level of parallelism) 。

Transformation (转换)	Meaning (含义)
<b>union</b> ( <i>otherStream</i> )	返回一个新的 DStream，它包含源 DStream 和 <i>otherDStream</i> 的所有元素。
<b>count</b> ()	通过 count 源 DStream 中每个 RDD 的元素数量，返回一个包含单元素 (single-element) RDDs 的新 DStream。
<b>reduce</b> ( <i>func</i> )	利用函数 <i>func</i> 聚集源 DStream 中每个 RDD 的元素，返回一个包含单元素 (single-element) RDDs 的新 DStream。函数应该是相关联的，以使计算可以并行化。
<b>countByValue</b> ()	在元素类型为 K 的 DStream 上，返回一个 (K,long) pair 的新的 DStream，每个 key 的值是在原 DStream 的每个 RDD 中的次数。
<b>reduceByKey</b> ( <i>func</i> , [ <i>numTasks</i> ])	当在一个由 (K,V) pairs 组成的 DStream 上调用这个算子时，返回一个新的，由 (K,V) pairs 组成的 DStream，每一个 key 的值均由给定的 reduce 函数聚合起来。 <b>注意</b> ：在默认情况下，这个算子利用了 Spark 默认的并发任务数去分组。你可以用 numTasks 参数设置不同的任务数。
<b>join</b> ( <i>otherStream</i> , [ <i>numTasks</i> ])	当应用于两个 DStream (一个包含 (K,V) 对，一个包含 (K,W) 对)，返回一个包含 (K, (V, W)) 对的新 DStream。
<b>cogroup</b> ( <i>otherStream</i> , [ <i>numTasks</i> ])	当应用于两个 DStream (一个包含 (K,V) 对，一个包含 (K,W) 对)，返回一个包含 (K, Seq[V], Seq[W]) 的 tuples (元组)。
<b>transform</b> ( <i>func</i> )	通过对源 DStream 的每个 RDD 应用 RDD-to-RDD 函数，创建一个新的 DStream。这个可以在 DStream 中的任何 RDD 操作中使用。
<b>updateStateByKey</b> ( <i>func</i> )	返回一个新的 "状态" 的 DStream，其中每个 key 的状态通过在 key 的先前状态应用给定的函数和 key 的新 values 来更新。这可以用于维护每个 key 的任意状态数据。

操作输出方法或者转换方法，其实内部都是在操作RDD，比如我们调用的foreachRDD，它就将我们的数据流，转换为RDD进行执行，完成数据结果输出，只不过是在流的内部完成的转换操作，最后结果输出还是一个批次流

## DStream (离散流)

它是SparkStreaming的数据抽象，一个批次会产生多个DStream，这点和RDD类似，同时在使用DStream的时候，其实底层会转换RDD操作，DStream操作，其实就是RDD在转换操作，SparkStreaming是一个准实时（秒级延迟） 候流处理的计算引擎，达不到实时处理状态。

## SparkStreaming对接Kafka (重点)

使用Kafka010版本后，我们Kafka内部自动维护Offset，看起来不错，主要Kafka不出问题，一切都没问题，但是你不能保证Kafka不会出现宕机情况，假如Kafka一旦出现宕机情况，那么你的Offset没法维护到Kafka内部，如果再次重启Kafka集群后，没有接收到上次提交的Offset，SparkStreaming再次消费数据的时候，就会发生数据重复消费的情况，这种情况是一定不允许，此时，我们需要将自动提交偏移量修改成false，找一个地方保存当前Offset，下次消费不去维护KafkaOffset，直接维护第三方保存Offset的数据库或者其他存储位置，这样就不会发生数据重复消费的情况。

对接kafka有两种方式：

## 1. Receiver

## 2. Direct（直连方式）

可以配置序列化方式：使用Spark内部的序列化方式，默认的是Java序列化方式  
`.set("spark.serializer","org.apache.spark.serializer.KryoSerializer")`

# 数据累加操作（UpdateStateByKey和MapWithState）了解

可以对每批次的结果进行累加操作，得到每次的结果总数据集

## MapWithState 了解

MapWithState和UpdateStateByKey都是做批次累加操作，都可以将每个批次结果进行累加计算，但是UpdateStateByKey是真正基于磁盘存储的，所有批次结果都会累加至磁盘，每次取值的时候也会直接访问磁盘，不管当前批次是否有值，都会获取之前批次结果数据，而MapWithState，虽然也是基于磁盘存储，但是它合理使用内存，也就是缓存，MapWithState会首先将数据刷入内存，进行缓存操作，然后在把结果写入磁盘，而当批次没有数据的时候，不会访问磁盘，也不会打印当前数据，当有数据进来才会输出，并且首先访问内存，如果内存数据被清空再去访问磁盘数据，来做每次累加操作。

## Transform

可以在DStream中，使用RDD进行操作，然后返回的还是RDD，最后转换为DStream，一句话，在流中使用RDD

可以解决RDD与RDD之间的join等，比如我们读取的字段文件，那么现在想把字典文件进行关联获取的数据流，需要将流转换为RDD处理。

案例：过滤广告黑名单

## foreachRDD

输出方法，也是对RDD操作，但是无返回值，一般用于调用各种Connection连接使用，和Transform的区别在，一个是转换操作，一个是输出操作

## 滑动窗口（Window）

两个概念：

滑动时间：必须是每个批次提交的倍数 比如每批次提交5秒，那么滑动时间就必须是5的倍数

滑动距离：滑动距离是你批次的倍数，比如 每次滑动时间10秒，那么滑动距离可以是10，也可以是20或者15

1574135483618

滑动窗口的应用：

可以使用滑动窗口进行多批次统计，比如按照时间需求可以统计前几次批次结果集，例如：1分钟统计前10分钟的批次的结果集



例如：每五分钟显示前1小时的数据结果集

注意：每个窗口长度一定是批次的倍数

## 手动维护Offset (Zookeeper) 重点

1. 连接Zookeeper和Kafka
2. 获取Offset (判断当前的Offset是否是第一次消费)
3. 处理数据后，将Offset维护到ZK

好处：保证数据不丢失，同时保证每次数据消费的精准性，同时Zookeeper分布式的，可以分布式存储，保存offset安全，如果当前消费数据程序挂掉，计算结果没出来，那么Offset将不会被更新，下次继续消费当前数据

## SparkStreaming的反压机制

设置最大接收速率- 如果集群资源不够大，streaming 应用程序能够像接收到的那样快速处理数据，则可以通过设置 记录/秒 的最大速率限制。Direct Kafka 方法的

`spark.streaming.kafka.maxRatePerPartition`，在Spark 1.5中，引入了一个称为背压的功能，无需设置此速率限制，因为Spark Streaming会自动计算速率限制，并在处理条件发生变化时动态调整速率限制。可以通过将 [配置参数](#) `spark.streaming.backpressure.enabled` 设置为 `true` 来启用此backpressure。

1574152488706

## 数据序列化

可以使用Spark中的内部序列化机制 Kryo序列化方式，进行序列化操作，相比较Java的序列化方式，速度要快10倍以上，序列化的大小要小于Java的序列化

## 保证消费一次性

*dempotent updates (幂等更新)*：多次尝试总是写入相同的数据。例如，`saveAs***Files` 总是将相同的数据写入生成的文件。

*Transactional updates (事务更新)*：所有更新都是事务性的，以便更新完全按原子进行。这样做的一个方法如下。

## Redis存储Offset (重点)

1. 获取连接
2. 获取Offset (有或者无)
3. 更新Offset

在使用Redis连接的时候，注意一点，Connection连接无法被序列化，也就是说在Driver创建的连接，无法发送到Executor内部使用，需要在Executor内部创建单独的连接操作，才可以使用数据的累加或者计算操作，而且Driver负责维护Offset，此连接和Executor内部的连接不冲突，就算在Executor内部关闭连接，也不会影响Driver的连接，因为在Executor内部关闭的连接是Executor的Jedis连接，和Driver无关。

## Spark On Yarn

原理：

cluster模式

1574316705821

1. 客户端（Spark集群的某台服务器）提交任务到集群（RM）
2. RM接收任务后，将任务发送到对应的NM（NM会根据RM分配的任务启动对应的Container）
3. NM接收到任务，首先启动一个APP Master（Driver），再去创建对应的Container
4. Container内部封装对应的Executor资源，Executor启动后，反向注册到Driver（APP Master）
5. APP Master接收到Executor反向注册的信息，监控集群内的Task运行 状态
6. APP Master将监控信息返回给RM，供给客户端显示（log）
7. 任务运行完成，各个NM会释放对应的资源，注销当前的任务

其实Spark On Yarn 的模式，说白了就是Yarn的原理，在这里面没有Master节点，也没有Worker节点，有的就是RM，NM，调度操作都是有RM来执行，任务运行都是NM来执行。任务的监控由内部创建的APP Master完成

client模式

1574316692112

1. 由客户端（不是集群内的服务器）提交任务到集群（创建APP Master）
2. RM接收任务，会将任务分配到应用的NM上面
3. NM会启动对应的Container，内部封装Executor的资源
4. Executor启动后，反向注册到Driver（单独提交任务的那台服务器）
5. Driver实时监控当前集群的运行状态（Task运行状态）
6. 任务执行完成，log任务会在Driver显示，同时RM会收集log到服务器

两者提交方式都可以正常运行任务，但是注意，Cluster模式提交任务后，Driver在内部创建，并且实时监控，也就是说，只要集群正常运行，Driver就正常工作，最后输出结果，但是Client的模式，如果提交任务的这台服务器挂了，那么任务失败

## Spark源码分析（了解）

DAG Scheduler：有向无环图，根据宽窄依赖划分Stage，将所有Task提交至Task Scheduler上

1. 首先DAG触发条件是Action（例如：foreach）点入runjob方法
2. 进入DAG的runjob方法，内部开始提交当前任务，找到submitJob方法提交
3. submitJob内部主要是将Job赋值Id，创建阻塞线程，让每个任务 按循序执行
4. 通过Event事件队列，进行匹配，匹配当前提交任务的事件
5. 找到当执行方（首先创建ResultStage）
6. 通过getOrCreateShuffleMapStage，划分所有的shuffle依赖
7. 倒推完成后，得到finalStage，然后将finalStage传入getMissingParentStages方法进行划分Stage
8. 内部通过Visit方法，进行递归 调用，循环所有的依赖，如果遇到宽依赖就保存，如果是 窄依赖就push回去继续循环
9. 当所有Stage划分完成后，将finalStage提交，通过submitStage方法提交
10. 接下里划分通过 TasksubmitMissingTasks
11. 内部通过Stage划分Task，Task的数量等于Partition数量，内部会划分两种Task
12. 划分完成Task提交所有Tasks到TaskSet中
13. 通过taskScheduler.submitTasks 提交至TaskSchedule上面，进行本地化的划分
14. 这里调用的是 实现taskScheduler特质的TaskSchedulerImpl
15. 找到submitTasks的重写方法，在TaskSchedulerImpl实现Task提交
16. 每个Task在执行之前都会监控通过TaskSetManager
17. 执行Task是FIFO的调度模式（先进先出）
18. 开始分配所有Task到每个节点内部，通过resourceOffers方法

19. 找到taskSet.executorAdded() 划分本地化级别
20. 本地化级别通过枚举方式实现computeValidLocalityLevels()
21. 五种本地化级别 PROCESS\_LOCAL, NODE\_LOCAL, NO\_PREF, RACK\_LOCAL, ANY 分别是，节点本地化，进程本地化，机架本地化，无本地化，没有
22. TaskSetManager内部实现本地化级别划分（最优位置）
23. Task会执行Run方法，然后每个Task会通过执行句柄发送到Executor内部执行
24. Executor内部会将Task反序列化，根据核数运行Task

Task Scheduler：接收反向注册信息，发送Task（本地级别）

## Shuffle（重点）

HashShuffle（了解）

未优化：首先通过ShuffleMapTask写入数据，然后每个ShuffleMapTask创建reduce个数的缓冲区，每个缓冲区都会溢写文件到本地磁盘，生成shuffleBlockFile本地文件，每个缓冲区都会创建一个shuffleBlockFile文件，然后在创建文件后，会创建对应的索引文件，将所有的本地文件元数据信息发送到Driver端，Driver会保存着元数据信息，然后reduce拉取本地文件的时候，会根据driver内的元数据信息拉取数据。

优化后：开启合并小文件机制（consolidation 机制），这样减少小文件输出，复用文件和缓存区，减少资源浪费

spark1.6以前这种shuffle还是不错的，减少磁盘IO，和文件的创建，性能还可以，但是2.0以后，因为底层内存模型发生改变，造成这种shuffle操作，不能完全合理利用内存，所以被弃用，换成SortShuffle机制

SortShuffle（掌握）

普通机制：内部有两种数据结构，一个是Map，一个Array，Map这种会做局部的聚合操作，提前进行分区内的合并，而Array直接写入内存，然后排序到每个缓冲区，缓冲区满了以后，会溢写磁盘文件，溢写后的小文件会被合并（这点跟MR比较像），然后合并的时候，都会记录当前的索引信息，当reduce拉取数据的时候，那么会根据索引信息找到位置去拉取merge文件内的数据

触发条件是大于200个Task

ByPass机制：其实前身就是HashShuffle机制，通过溢写文件，然后在去合并当前文件，而hash的是复用之前的本地文件，触发的条件是Task小于200个

1574326055812

钨丝Shuffle（TungstenShuffle了解）

其实内部实现的是SortShuffle普通机制，只不过对内存和缓存进行的合理利用，但是很多算子还是不支持钨丝shuffle这种shuffle操作。

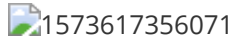
## Spark对接Hive（必会）

Spark操作Hive相比较Hive执行的MR，效率要高，因为Spark是基于DAG有向无环图，实现的内存迭代式计算，MR是基于磁盘计算引擎，相比Spark的迭代计算，要慢的多，并且磁盘IO太大，没有太好的优化，Spark是内存处理，速度要快的多，所以使用Spark对接Hive已经成为主流，例如：SparkSQL来实现的数仓操作

实现方式：将Hive中的hive-site.xml文件和hdfs-site.xml、core-site.xml拷贝过来，放入idea中resource中即可

## Kafka

消息中间件、消息队列



Kafka适合什么样的场景？

它可以用于两大类的应用：

1. 构造实时流数据管道，它可以在系统或应用之间可靠地获取数据。（相当于message queue）
2. 构建实时流式应用程序，对这些流数据进行转换或者影响。（就是流处理，通过kafka stream topic和topic之间内部进行变化）

kafka的组件：

1. 消息生产者：Producer，是消息的生产源头，负责生产消息并发送到Kafka服务器上
2. 消息消费者：Consumer，是消息的使用方，负责消费Kafka服务器上的消息
3. 主题：Topic，由用户定义在Kafka服务器上面，用于建立生产者和消费者之间的消费关系，生产者生产消息到指定Topic下面，消费者从这个Topic消费数据
4. 分区：Partition，一个Topic下面会有多个分区主要是做备份，实现高容错性，每个分区都有一台server作为“leader”，零台或者多台server作为followers。leader server处理一切对partition（分区）的读写请求，而followers只需被动的同步leader上的数据。当leader宕机了，followers中的一台服务器会自动成为新的leader。每台server都会成为某些分区的leader和某些分区的follower，因此集群的负载是平衡的。
5. Broker：kafka的服务器，用户存储消息，分布式
6. 消费者组：Group，用于归纳组别，主要是在消费的时候，可以定义多个消费者，同时消费数据，保证在一个组别下即可，注意：消费者组可以消费一个或者多个分区的数据，但是一个分区的数据同一时刻只能被一个消费者来消费，防止重复消费
7. Offset：消息的偏移量，在消费Kafka分区内数据的时候，Kafka会记录消息消费的偏移量，在内部也会被存储

Kafka集群：

1. Kafka集群可以保存多种数据类型的数据，每个数据都会保存到Topic下面
2. Kafka集群可以创建多个Topic，并且每个Topic的分区都有副本机制，可以自己指定
3. 每个分区的数据是由多个segment组成，里面包含一个或者多个Index文件和.log文件
4. 每个分区的副本，不会和主分区在一起，会分发到其他节点

ActionMQ、RabbitMQ、Kafka（支持动态扩容）

Kafka的常用命令（0.8）：

注意：Kafka1.0以上版本的命令和0.8有些不同

新建主题：

```
./bin/kafka-topics.sh --create --zookeeper node4:2181 --partitions 3 --replication-factor 3 --topic sz1901
```

查看主题：

```
./bin/kafka-topics.sh --list --zookeeper node4:2181
```

查看主题详情：

```
./bin/kafka-topics.sh --describe --zookeeper node4:2181 --topic sz1901test
```

删除主题：

```
./bin/kafka-topics.sh --delete --zookeeper node4:2181 --topic sz1901test
```

注意：需要添加相关参数 `server.properties`中设置`delete.topic.enable=true`

修改分区数量（只能增加不能减少）

```
bin/kafka-topics.sh --zookeeper node4:2181 --alter --partitions 5 --topic sz1901
```

启动生产者（0.8）：

```
./bin/kafka-console-producer.sh --broker-list node1:9092 --topic sz1901
```

启动消费者（0.8）：


```
./bin/kafka-console-consumer.sh --zookeeper node4:2181 --topic sz1901 --from-beginning
```

## Kafka文件存储机制

如果我们在往Kafka内部生产数据的话，那么Kafka会根据分区进行分配数据，并且每个分区内都会有一个log文件，和一个Index索引文件，log文件内存储的是消息（数据），而index索引文件，存储的是offset索引信息

## Kafka分区和消费者的消费策略


通过轮循的方式将每个分区轮循到相应的消费者组里面的每个消费者身上。

1573635273101


## kafkaAPI操作

0.10 可以使用

## Kafka 文件传输机制

1573702397129

## ACK应答机制（重点）

1573703918413

## Offset的维护（重点）

两种维护Offset的方式

自动提交：设置参数，进行自动提交，默认就是自动的 `enable.auto.commit=true`，无论你是否消费成功，我都会给你提交当前的offset

手动提交：设置参数 `enable.auto.commit=false`，手动更新当前的Offset，如果我们的消费者服务器宕机，没有消费到数据，那么此时会将offset进行手动存储到其他位置，再次启动服务，继续消费，会去读取offset的位置

现在都用手动维护Offset，这个Offset我们可以存储起来，比如MySQL、Redis、HDFS、Hbase等等

手动维护Offset，可以保证消费者消费数据的安全性，保存每次消费的精准性

## kafka问题总结（重要）

### 1. 如何保证Kafka的生产者消息安全性？

设置ACK机制，选择最合适参数即可，ACK机制，也叫应答机制，主要是保证producer生产过来的数据，每条数据都写入Leader中，leader返回确认消息，表示写入成功，可以选择级别，比如 0 写入leader即可 1代表写入leader，并且leader刷入磁盘即可，-1 代表写入leader，并且同步到各个follow中，表示写入成功，但是设置ACK会导致性能下降。

### 2. 如何保证消费者消费数据的准确性？

手动维护偏移量，可以将offset参数设置为false，如果我们改成false后，那么需要编写保存offset的代表，可以将offset保存到其他位置，如果当前消费者宕机，那么重启后，可以读取保存好的offset，进行继续消费，这样就能保证数据消费的准确性，但是，这样的话，我们开发程序的代码就需要自己编写，相比较自动的，要复杂的多。

### 3. Kafka为什么读取速度快？

顺序读写=读取一个大文件

随机读写=读取多个小文件

顺序读写比随机读写快的原因

1. 顺序读写，读写时间主要是体现在传输过程，而这个过程是在一个文件中进行的，而随机读写需要多次寻找数据，然后在多个文件中进行随机写入，它的时间主要花在了查找数据和写入数据

2. 顺序写入，磁盘会有预先读取，而随机写入，没有预先读取，自然要比较顺序慢得多

### 4. Kafka的内部存储方式？

索引式存储方式（稀疏存储），在本地会生成两个数据文件，一个是索引文件，一个是数据文件，索引文件只记录当前数据的索引信息（offset），log文件存储真正数据，当消费数据的时候，如果按照Offset查询，那么会通过二分法实现数据的查找

### 5. Kafka文件传输机制？

文件传输机制，分为两个部分，生产者和消费者，但是该无论是生产还是消费都是基于leader实现的，leader主要是负责读写，而follow负责同步消息，当leader挂掉，会在当前的副本中选举新的leader，进行数据处理（读写操作）

### 6. Kafka吞吐量（参考文章）？

<http://ifeve.com/benchmarking-apache-kafka-2-million-writes-second-three-cheap-machine-sl/>

### 7. Kafka如何对接消费者，怎么消费的？

实现方式，用直连方式实现数据消费，手动维护Offset，将Offset存储至Redis中，保证数据读取的准确性

## Flume对接Kafka

Flume充当生产者



Kafka做消息缓存

消费者（命令启动）

## Redis介绍

redis是一个K-V存储系统，它是内存数据库，主要是用于做数据的缓存，同时在大数据领域上，主要是做数据结果的存储，因为它是一个内存数据库，读写效率高，每秒吞吐量达到11W左右，支持多种数据类型（String，list，set等等），C语言开发的K-V数据库

应用场景：Java居多，用于缓存页面，大数据领域，主要是在做实时流处理统计的时候，使用Redis进行数据的存储

Nosql数据库：MongoDB、Hbase、Redis

Nosql优势：非关系数据库，K-V存储，查询效率高，数据存储容量大（Redis除外），可以按照数据的K-V结构处理当前存储的数据

注意：3.0以下版本不支持集群模式，3.0以上支持集群模式

## 安装Redis

## Redis数据类型

String类型 字符串

Hash类型 散列

List类型 列表

Set类型 集合

Zset类型 有序集合

## Redis-API连接

添加redis的java依赖

```
<dependency>
  <groupId>redis.clients</groupId>
  <artifactId>jedis</artifactId>
  <version>2.9.0</version>
</dependency>
```

## 资源设置和使用

序号	参数名	含义	默认值	使用建议
1	maxTotal	资源池中最大连接数	8	设置建议见下节
2	maxIdle	资源池允许最大空闲的连接数	8	设置建议见下节
3	minIdle	资源池确保最少空闲的连接数	0	设置建议见下节
4	blockWhenExhausted	当资源池用尽后，调用者是否要等待。只有当为true时，下面的maxWaitMillis才会生效	true	建议使用默认值
5	maxWaitMillis	当资源池连接用尽后，调用者的最大等待时间(单位为毫秒)	-1： 表示永不超时	不建议使用默认值
6	testOnBorrow	向资源池借用连接时是否做连接有效性检测(ping)，无效连接会被移除	false	业务量很大时候建议设置为false(多一次ping的开销)。
7	testOnReturn	向资源池归还连接时是否做连接有效性检测(ping)，无效连接会被移除	false	业务量很大时候建议设置为false(多一次ping的开销)。
8	jmxEnabled	是否开启jmx监控，可用于监控	true	建议开启，但应用本身也要开启

## 2.空闲资源监测

空闲Jedis对象检测，下面四个参数组合来完成，testWhileIdle是该功能的开关。

序号	参数名	含义	默认值	使用建议
1	testWhileIdle	是否开启空闲资源监测	false	true
2	timeBetweenEvictionRunsMillis	空闲资源的检测周期(单位为毫秒)	-1： 不检测	建议设置，周期自行选择，也可以使用下面JedisPoolConfig中的配置
		资源池中资源最	1000	可根据自身业务决定，大部分默认值即可



序号	参数名	说明	默认值	使用建议
	minEvictionTimeMillis	小空闲时间(单位为毫秒), 达到此值后空闲资源将被移除	60 * 30 = 30分钟	正, 大部分默认值即可, 也可以考虑使用下面 JedisPoolConfig中的配置
4	numTestsPerEvictionRun	做空闲资源检测时, 每次的采样数	3	可根据自身应用连接数进行微调, 如果设置为-1, 就是对所有连接做空闲监测

## Redis 持久化

RDB: 优点在于效率很高, 不会频繁的刷入数据到磁盘, 减少IO操作, 缺点是数据安全没保障, 如果某一时刻数据库挂掉了, 那么没有达到持久化要求, 此时的数据不会被刷入磁盘存储, 所以下次启动后, 无法回复未持久化的数据

AOF: 优点是数据安全, 缺点效率低, 并且频繁执行存储, IO过大, 而且内部存储的是命令, 不是数据, 如果进行恢复, 那么需要执行命令恢复数据。

总结: RDB适用于对数据安全性低的操作, 而AOF恰恰相反, 适用于数据安全性高的。

## Redis集群

Redis集群, 首先保证最少3个主节点, 三个从节点, 保证某一节点宕机半数以上节点存活, 同时, 每个槽位都是固定的, 如果当一个节点宕机, 那么从节点之间顶替主节点工作, 当主节点没问题的时候, 从节点负责同步数据即可

## 案例解析