# 千锋好程序员大数据面试宝典

## 第一部分：Java语言基础

#### 1、面向对象的特征有哪些方面?

* 1. 1)抽象：抽象就是忽略一个主题中与当前目标无关的那些方面，以便更充分地注意与当前目标有关的方面。抽象并不打算了解全部问题，而只是选择其中的一部分，暂时不用部分细节。抽象包括两个方面，一是过程抽象,二是数据抽象。
  2. 2)继承：继承是一种联结类的层次模型，并且允许和鼓励类的重用，它提供了一种明确表述共性的方法。对象的一个新类可以从现有的类中派生，这个过程称为类继承。新类继承了原始类的特性，新类称为原始类的派生类（子类），而原始类称为新类的基类（父类）。派生类可以从它的基类那里继承方法和实例变量，并且类可以修改或增加新的方法使之更适合特殊的需要。
  3. 3)封装：封装是把过程和数据包围起来，对数据的访问只能通过已定义的界面。面向对象计算始于这个基本概念，即现实世界可以被描绘成一系列完全自治、封装的对象,这些对象通过一个受保护的接口访问其他对象。
  4. 4)多态性：多态性是指允许不同类的对象对同一消息作出响应。多态性包括参数化多态性和包含多态性。多态性语言具有灵活、抽象、行为共享、代码共享的优势，很好的解决了应用程序函数同名问题。

#### 作用域public,private,protected,以及不写时的区别？

* 1. 答：区别如下：
  2. 作用域 当前类 同包 子孙类 其他
  3. public √ √ √ √
  4. protected √ √ √ ×
  5. default √ √ × ×
  6. private √ × × ×
  7. 不写时默认为default。

#### 3、String 是最基本的数据类型吗? 【基础】

* 1. 答：不是。

#### 4、float 型float f=3.4是否正确? 【基础】

* 1. 答:不正确；
  2. 精度不准确,应该用强制类型转换，如下所示：float f=(float)3.4 。
  3. 或者使用 float f = 3.4f;

#### 5、语句float f=1.3；编译能否通过？【基础】

* 1. 答:不能；
  2. 应该用强制类型转换，如下所示：float f=(float)1.3；

或者使用 float f = 1.3f;

1. **short s1 = 1; s1 = s1 + 1;有什么错?**
   1. short s1 = 1; s1 += 1;有什么错? 【基础】
   2. 答：short s1 = 1; s1 = s1 + 1;s1+1运算结果是int 型，需要强制转换类型；short s1 = 1; s1 += 1;可以正确编译,自动类型提升。
2. **Java 有没有goto? 【基础】**
   1. 答：goto 是java 中的保留字，现在没有在java 中使用。
3. **int 和Integer 有什么区别? 【基础】**
   1. 答：Java 提供两种不同的类型：引用类型和原始类型（或内置类型）；
   2. int 是java 的原始数据类型，Integer 是java 为int 提供的封装类。
   3. Java 为每个原始类型提供了封装类：
   4. 原始类型: boolean,char,byte,short,int,long,float,double
   5. 封装类型：Boolean，Character，Byte，Short，Integer，Long，Float，Double
   6. 引用类型和原始类型的行为完全不同，并且它们具有不同的语义。引用类型和原始类型具有不同的特征和用法，它们包括：大小和速度问题，这种类型以哪种类型的数据结构存储，当引用类型和原始类型用作某个类的实例数据时所指定的缺省值。对象引用实例变量的缺省值为null，而原始类型实例变量的缺省值与它们的类型有关。
4. **&和&&的区别？【基础】**
   1. 答：&是位运算符，表示按位与运算，&&是逻辑运算符，表示逻辑与（and）。
5. **简述逻辑操作(&,|,^)与条件操作(&&,||)的区别？【基础】**
   1. 答：区别主要有两点：
   2. a.条件操作只能操作布尔型的,而逻辑操作不仅可以操作布尔型,而且可以操作数值型
   3. b.逻辑操作不会产生短路。
6. **heap 和stack 有什么区别？【基础】**
   1. 答：栈是一种线形集合，其添加和删除元素的操作应在同一段完成，栈按照后进先出的方式进行处理；堆是栈的一个组成元素。
7. **Math.round(11.5) 等于多少? Math.round(-11.5)等于多少? 【基础】**
   1. 答：Math.round(11.5)==12 Math.round(-11.5)==-11
   2. round 方法返回与参数最接近的长整数，参数加1/2 后求其floor。
8. **swtich 是否能作用在byte 上，是否能作用在long 上，是否能作用在String上? 【基础】**
   1. 答：switch（expr1）中，expr1 是一个整数表达式。因此传递给switch 和case语句的参数应该是int、short、char 或者byte。long,string 都不能作用于swtich。
9. **编程题: 用最有效率的方法算出2 乘以8 等於几? 【基础】**
   1. 答： 2 << 3。
10. **有没有length()这个方法? String 有没有length()这个方法？【基础】**
    1. 答：数组没有length()这个方法，有length 的属性。String 有length()这个方法。
11. **在JAVA 中，如何跳出当前的多重嵌套循环？【基础】**
    1. 答：在最外层循环前加label 标识,然后用break label 方法即可跳出多重循环。
12. **构造器Constructor 是否可被override? 【基础】**
    1. 答：构造器Constructor 不能被继承，因此不能重写Overriding，但可以被重载Overloading。
13. **两个对象值相同(x.equals(y) == true)，但却可有不同的hash code，这句话对不对? 【基础】**
    1. 答：不对，有相同的hash code。
14. **是否可以继承String 类? 【基础】**
    1. 答：String 类是final 类，故不可以继承。
15. **以下二条语句返回值为true 的有：**
    1. A：“beijing”==“beijing”；
    2. B：“beijing”.equalsIgnoreCase（new String（“beijing”））；【基础】
    3. 答：A 和B 。
16. **当一个对象被当作参数传递到一个方法后，此方法可改变这个对象的属性，并可返回变化后的结果，那么这里到底是值传递还是引用传递? 【基础】**
    1. 答：是值传递。Java 编程语言只有值传递参数。当一个对象实例作为一个参数被传递到方法中时，参数的值就是对该对象的引用。对象的内容可以在被调用的方法中改变，但对象的引用是永远不会改变的。
17. **String 和StringBuffer 的区别? 【基础】**
    1. 答：JAVA 平台提供了两个类：String 和StringBuffer，它们可以储存和操作字符串，即包含多个字符的字符数据。
    2. 这个String 类提供了数值不可改变的字符串。
    3. 而这个StringBuffer 类提供的字符串进行修改。当你知道字符数据要改变的时候你就可以使用StringBuffer。典型地，你可以使用StringBuffers 来动态构造字符数据。
18. **String, StringBuffer StringBuilder 的区别。【基础】**
    1. 答：String 的长度是不可变的；StringBuffer 的长度是可变的，如果你对字符串中的内容经常进行操作，特别是内容要修改时，那么使用StringBuffer，如果最后需要String，那么使用StringBuffer 的toString()方法；线程安全；StringBuilder 是从JDK 5 开始，为StringBuffer 该类补充了一个单个线程使用的等价类；通常应该优先使用StringBuilder 类，因为它支持所有相同的操作，但由于它不执行同步，所以速度更快。
19. **Overload 和Override 的区别。Overloaded 的方法是否可以改变返回值的类型? 【基础】**
    1. 答：方法的重写Overriding 和重载Overloading 是Java 多态性的不同表现。重写Overriding 是父类与子类之间多态性的一种表现，重载Overloading 是一个类中多态性的一种表现。如果在子类中定义某方法与其父类有相同的名称和参数，我们说该方法被重写(Overriding)。子类的对象使用这个方法时，将调用子类中的定义，对它而言，父类中的定义如同被“屏蔽”了。如果在一个类中定义了多个同名的方法，它们或有不同的参数个数或有不同的参数类型，则称为方法的重载(Overloading)。Overloaded 的方法是可以改变返回值的类型。
20. **定义类A 和类B 如下：【基础】**
    1. class A {

int a=1;

double d=2.0;

void show(){

System.out.println("Class A: a="+a +"\td="+d);

}

* 1. }
  2. class B extends A{

float a=3.0f;

String d="Java program.";

void show(){

super.show( );

System.out.println("Class B: a="+a +"\td="+d);

}

* 1. }
  2. (1) 若在应用程序的main 方法中有以下语句：
  3. A a=new A();
  4. a.show();
  5. 则输出的结果如何？
  6. (2) 若在应用程序的main 方法中定义类B 的对象b：
  7. A b=new B();
  8. b.show();
  9. 则输出的结果如何？
  10. 答：输出结果为：
  11. 1）Class A: a=1 d=2.0 ；
  12. 2）Class A: a=1 d=2.0
  13. Class B: a=3.0 d=Java program。

1. **描述一下JVM 加载class 文件的原理机制? 【基础】**
   1. 答：JVM 中类的装载是由ClassLoader 和它的子类来实现的,Java ClassLoader是一个重要的Java 运行时系统组件。它负责在运行时查找和装入类文件的类。
2. **char 型变量中能不能存贮一个中文汉字?为什么? 【基础】**
   1. 答：能够定义成为一个中文的，因为java 中以unicode 编码，一个char 占16个字节，所以放一个中文是没问题的。
3. **abstract class 和interface 有什么区别? 【基础】**
   1. 答：声明方法的存在而不去实现它的类被叫做抽象类（abstract class），。然而可以创建一个变量，其类型是一个抽象类，它用于要创建一个体现某些基本行为的类，并为该类声明方法，但不能在该类中实现该类的情况。不能创建abstract 类的实例并让它指向具体子类的一个实例。不能有抽象构造函数或抽象静态方法。Abstract 类的子类为它们父类中的所有抽象方法提供实现，否则它们也是抽象类为。取而代之，在子类中实现该方法。知道其行为的其它类可以在类中实现这些方法。接口（interface）是抽象类的变体。新型多继承性可通过实现这样的接口而获得。接口中的所有方法都是抽象的，所有成员变量都是publicstatic final 的。一个类可以实现多个接口，当类实现特殊接口时，它定义（即
   2. 将程序体给予）所有这种接口的方法。然后，它可以在实现了该接口的类的任何对象上调用接口的方法。由于有抽象类，它允许使用接口名作为引用变量的类型。通常的动态联编将生效。引用可以转换到接口类型或从接口类型转换，instanceof 运算符可以用来决定某对象的类是否实现了接口。
4. **Static Nested Class 和Inner Class 的不同？【基础】**
   1. 答：Static Nested Class 是被声明为静态（static）的内部类，它可以不依赖于外部类实例被实例化。而通常的内部类需要在外部类实例化后才能实例化。
5. **Java 中会存在内存泄漏吗，请简单描述。【基础】**
   1. 答：会；存在无用但可达的对象，这些对象不能被GC 回收，导致耗费内存资源。
6. **abstract 的method 是否可同时是static,是否可同时是native，是否可同时是synchronized? 【基础】**
   1. 答：都不能。
7. **静态变量和实例变量的区别？【基础】**
   1. 答：静态变量也称为类变量，归全类共有，它不依赖于某个对象，可通过类名直接访问；而实例变量必须依存于某一实例，只能通过对象才能访问到它。
8. **是否可以从一个static 方法内部发出对非static 方法的调用？【基础】**
   1. 答：不可以,如果其中包含对象的method()，不能保证对象初始化。
9. **写clone()方法时，通常都有一行代码，是什么？【基础】**
   1. 答：Clone 有缺省行为：super.clone()，他负责产生正确大小的空间，并逐位复制。
10. **GC 是什么? 为什么要有GC? 【基础】**
    1. 答：GC 是垃圾收集的意思（Gabage Collection）,内存处理是编程人员容易出现问题的地方，忘记或者错误的内存回收会导致程序或系统的不稳定甚至崩溃，Java 提供的GC 功能可以自动监测对象是否超过作用域从而达到自动回收内存的目的，Java 语言没有提供释放已分配内存的显示操作方法。Java 程序员不用担心内存管理，因为垃圾收集器会自动进行管理。要请求垃圾收集，可以调用下面的方法之一：System.gc() 或Runtime.getRuntime().gc() 。
11. **垃圾回收的优点和原理。并考虑2 种回收机制。【基础】**
    1. 答：Java 语言中一个显著的特点就是引入了垃圾回收机制，使c++程序员最头疼的内存管理的问题迎刃而解，它使得Java 程序员在编写程序的时候不再需要考虑内存管理。由于有个垃圾回收机制，Java 中的对象不再有“作用域”的概念，只有对象的引用才有“作用域”。垃圾回收可以有效的防止内存泄露，有效的使用可以使用的内存。垃圾回收器通常是作为一个单独的低级别的线程运行，不可预知的情况下对内存堆中已经死亡的或者长时间没有使用的对象进行清楚和回收，程序员不能实时的调用垃圾回收器对某个对象或所有对象进行垃圾回收。回收机制有分代复制垃圾回收和标记垃圾回收，增量垃圾回收。
12. **垃圾回收器的基本原理是什么？垃圾回收器可以马上回收内存吗？有什么办法主动通知虚拟机进行垃圾回收？【基础】**
    1. 答：对于GC 来说，当程序员创建对象时，GC 就开始监控这个对象的地址、大小以及使用情况。通常，GC 采用有向图的方式记录和管理堆(heap)中的所有对象。通过这种方式确定哪些对象是"可达的"，哪些对象是"不可达的"。当GC 确定一些对象为"不可达"时，GC 就有责任回收这些内存空间。可以。程序员可以手动执行System.gc()，通知GC 运行，但是Java 语言规范并不保证GC 一定会执行。
13. **String s=new String(“xyz”);创建了几个String Object? 【基础】**
    1. 答：两个对象，一个是"xyx",一个是指向"xyx"的引用对象s。
14. **接口是否可继承接口? 抽象类是否可实现(implements)接口? 抽象类是否可继承实体类(concrete class)? 【基础】**
    1. 答：接口可以继承接口。抽象类可以实现(implements)接口，抽象类可继承实体类，但前提是实体类必须有明确的构造函数。
15. **Java 的接口和C++的虚类的相同和不同处。【基础】**
    1. 答：由于Java 不支持多继承，而有可能某个类或对象要使用分别在几个类或对象里面的方法或属性，现有的单继承机制就不能满足要求。与继承相比，接口有更高的灵活性，因为接口中没有任何实现代码。当一个类实现了接口以后，该类要实现接口里面所有的方法和属性，并且接口里面的属性在默认状态下面都是public static,所有方法默认情况下是public.一个类可以实现多个接口。
16. **一个“.java”源文件中是否可以包含多个类（不是内部类）？有什么限制？【基础】**
    1. 答：可以；必须只有一个类名与文件名相同。
17. **说出一些常用的类，包，接口，请各举5 个。【基础】**
    1. 答：常用的类：BufferedReader BufferedWriter FileReader FileWirter String Integer；
    2. 常用的包：java.lang java.awt java.io java.util java.sql；
    3. 常用的接口：Remote List Map Document NodeList
18. **Anonymous Inner Class (匿名内部类) 是否可以extends(继承)其它类？是否可以implements(实现)interface(接口)? 【基础】**
    1. 答：可以继承其他类或实现其他接口，在swing 编程中常用此方式。
19. **内部类可以引用他包含类的成员吗？有没有什么限制？【基础】**
    1. 答：一个内部类对象可以访问创建它的外部类对象的内容。
20. **java 中实现多态的机制是什么？【基础】**
    1. 答：方法的覆盖Overriding 和重载Overloading 是java 多态性的不同表现；
    2. 覆盖Overriding 是父类与子类之间多态性的一种表现，
    3. 重载Overloading 是一个类中多态性的一种表现。
21. **在java 中一个类被声明为final 类型，表示了什么意思？【基础】**
    1. 答：表示该类不能被继承，是顶级类。
22. **下面哪些类可以被继承? 【基础】**
    1. 1）java.lang.Thread (T)
    2. 2）java.lang.Number (T)
    3. 3）java.lang.Double (F)
    4. 4）java.lang.Math (F)
    5. 5）java.lang.Void (F)
    6. 6）java.lang.Class (F)
    7. 7）java.lang.ClassLoader (T)
    8. 答：1、2、7 可以被继承。
23. **指出下面程序的运行结果: 【基础】**
    1. class A{

static{

System.out.print("1");

}

public A(){

System.out.print("2");

}

* 1. }
  2. class B extends A{

static{

System.out.print("a");

}

public B(){

System.out.print("b");

}

* 1. }
  2. public class Hello{

public static void main(String[] ars){

A ab = new B(); //执行到此处,结果: 1a2b

ab = new B(); //执行到此处,结果: 1a2b2b

}

* 1. }
  2. 答：输出结果为1a2b2b；类的static 代码段,可以看作是类首次加载(虚拟机加载)执行的代码,而对于类加载,首先要执行其基类的构造,再执行其本身的构造。

1. **继承时候类的执行顺序问题,一般都是选择题,问你将会打印出什么?【基础】**
   1. 父类：
   2. package test;
   3. public class FatherClass {

public FatherClass() {

System.out.println("FatherClass Create");

}

* 1. }
  2. 子类:
  3. package test;
  4. import test.FatherClass;
  5. public class ChildClass extends FatherClass {

public ChildClass() {

System.out.println("ChildClass Create");

}

public static void main(String[] args) {

FatherClass fc = new FatherClass();

ChildClass cc = new ChildClass();

}

* 1. }
  2. 答：输出结果为：
  3. FatherClass Create
  4. FatherClass Create
  5. ChildClass Create

1. **内部类的实现方式? 【基础】**
   1. 答：示例代码如下：
   2. package test;
   3. public class OuterClass {

private class InterClass {

public InterClass() {

System.out.println("InterClass Create");

}

}

public OuterClass() {

InterClass ic = new InterClass();

System.out.println("OuterClass Create");

}

public static void main(String[] args) {

OuterClass oc = new OuterClass();

}

* 1. }
  2. 输出结果为:
  3. InterClass Create
  4. OuterClass Create

1. **You need to insert an inner class declaration at line 3，Which two inner class declarations are valid?(Choose two.)**
   1. public class OuterClass {

private double d1 = 1.0;

//insert code here

}

* 1. A. class InnerOne{

public static double methoda() {return d1;}

* 1. }
  2. B. public class InnerOne{

static double methoda() {return d1;}

* 1. }
  2. C. private class InnerOne{

double methoda() {return d1;}

* 1. }
  2. D. static class InnerOne{

protected double methoda() {return d1;}

* 1. }
  2. E. abstract class InnerOne{

public abstract double methoda();

* 1. }
  2. 答：答案为C、E；说明如下：
  3. 1）静态内部类可以有静态成员，而非静态内部类则不能有静态成员；故A、B错；
  4. 2）静态内部类的非静态成员可以访问外部类的静态变量，而不可访问外部类的非静态变量；故D 错；
  5. 3）非静态内部类的非静态成员可以访问外部类的非静态变量；故C 正确。

1. **数据类型之间的转换:**
   1. 1)如何将数值型字符转换为数字？
   2. 2)如何将数字转换为字符？
   3. 3)如何取小数点前两位并四舍五入? 【基础】
   4. 答：
   5. 1)调用数值类型相应包装类中的方法parse\*\*\*(String)或valueOf(String)即可返回相应基本类型或包装类型数值；
   6. 2)将数字与空字符串相加即可获得其所对应的字符串;另外对于基本类型数字还可调用String 类中的valueOf(…)方法返回相应字符串,而对于包装类型数字则可调用其toString()方法获得相应字符串；
   7. 3)可用该数字构造一java.math.BigDecimal 对象,再利用其round()方法进行四舍五入到保留小数点后两位,再将其转换为字符串截取最后两位。
2. **字符串操作：如何实现字符串的反转及替换？【基础】**
   1. 答：可用字符串构造一StringBuffer 对象,然后调用StringBuffer 中的reverse方法即可实现字符串的反转,调用replace 方法即可实现字符串的替换。
3. **编码转换：怎样将GB2312 编码的字符串转换为ISO-8859-1 编码的字符串？【基础】**
   1. 答：示例代码如下:
   2. String s1 = "你好";
   3. String s2 = new String(s1.getBytes("GB2312"), "ISO-8859-1");
4. **写一个函数，要求输入一个字符串和一个字符长度，对该字符串进行分隔。【基础】**
   1. 答：函数代码如下：
   2. public String[] split(String str, int chars){

int n = (str.length()+ chars - 1)/chars;

String ret[] = new String[n];

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

if(i < n-1){

ret[i] = str.substring(i\*chars , (i+1)\*chars);

}else{

ret[i] = str.substring(i\*chars);

}

}

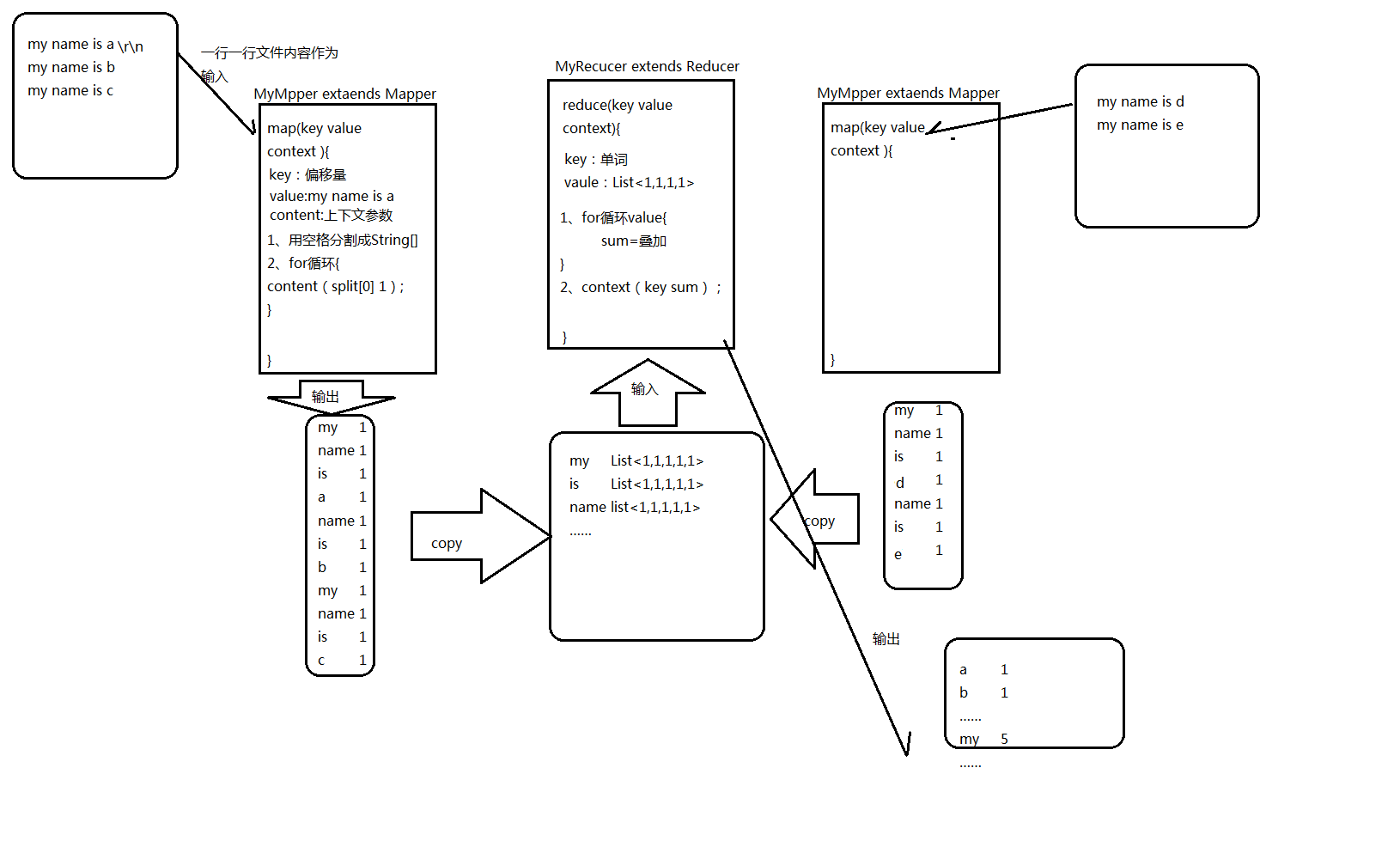
return ret;

* 1. }

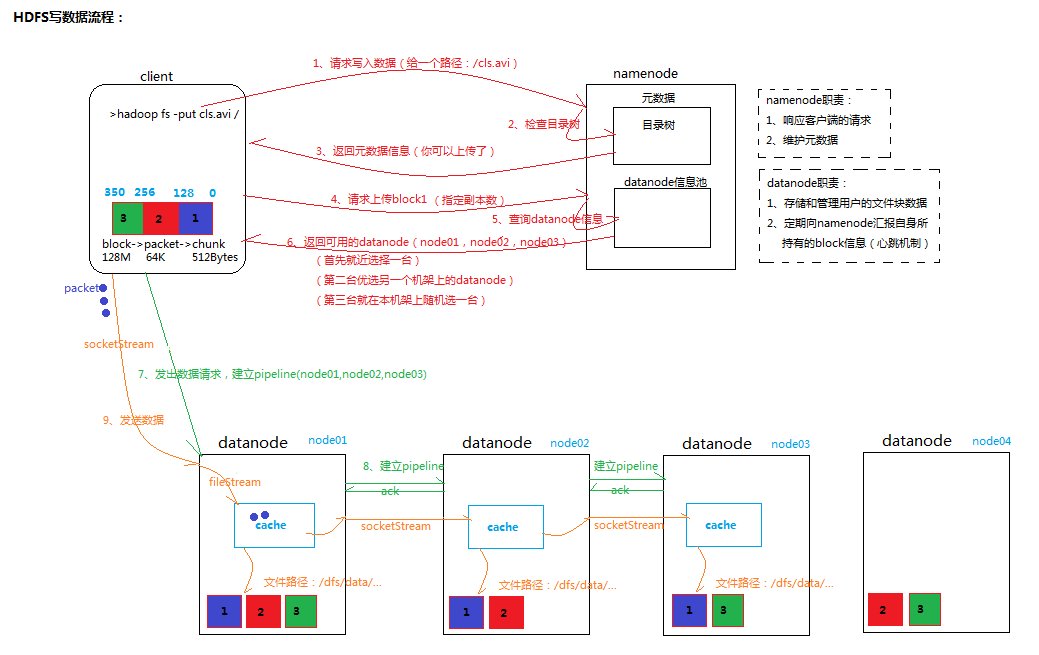
1. **字符串操作：如何实现字符串的反转及替换？【基础】**
   1. 答：可用字符串构造一StringBuffer 对象,然后调用StringBuffer 中的reverse方法即可实现字符串的反转,调用replace 方法即可实现字符串的替换。
2. **日期和时间：**
   1. 1)如何取得年月日、小时分秒？
   2. 2)如何取得从1970 年到现在的毫秒数？
   3. 3)如何取得某个日期是当月的最后一天？
   4. 4)如何格式化日期？【基础】
   5. 答：
   6. 1)创建java.util.Calendar 实例(Calendar.getInstance()),调用其get()方法传入不同的参数即可获得参数所对应的值,如：calendar.get(Calendar.YEAR);//获得年
   7. 2)以下方法均可获得该毫秒数:
   8. Calendar.getInstance().getTimeInMillis();System.currentTimeMillis();
   9. 3)示例代码如下:
   10. Calendar time = Calendar.getInstance();
   11. time.set(Calendar.DAY\_OF\_MONTH,
   12. time.getActualMaximum(Calendar.DAY\_OF\_MONTH));
3. **什么时候用assert？【中等难度】**
   1. 答：assertion(断言)在软件开发中是一种常用的调试方式，很多开发语言中都支持这种机制。一般来说，assertion 用于保证程序最基本、关键的正确性。assertion 检查通常在开发和测试时开启。为了提高性能，在软件发布后， assertion 检查通常是关闭的。在实现中，断言是一个包含布尔表达式的语句，在执行这个语句时假定该表达式为true；如果表达式计算为false，那么系统会报告一个Assertionerror。
   2. 断言用于调试目的：
   3. assert(a > 0); // throws an Assertionerror if a <= 0
   4. 断言可以有两种形式：
   5. assert Expression1 ;
   6. assert Expression1 : Expression2 ;
   7. Expression1 应该总是产生一个布尔值。
   8. Expression2 可以是得出一个值的任意表达式；这个值用于生成显示更多调试信息的String 消息。
   9. 断言在默认情况下是禁用的，要在编译时启用断言，需使用source 1.4 标记：
   10. javac -source 1.4 Test.java
   11. 要在运行时启用断言，可使用-enableassertions 或者-ea 标记。
   12. 要在运行时选择禁用断言，可使用-da 或者-disableassertions 标记。
   13. 要在系统类中启用断言，可使用-esa 或者-dsa 标记。还可以在包的基础上启用或者禁用断言。可以在预计正常情况下不会到达的任何位置上放置断言。断言可以用于验证传递给私有方法的参数。不过，断言不应该用于验证传递给公有方法的参数，因为不管是否启用了断言，公有方法都必须检查其参数。不过，既可以在公有方法中，也可以在非公有方法中利用断言测试后置条件。另外，断言不应该以任何方式改变程序的状态。
4. **Java 中的异常处理机制的简单原理和应用？【基础】**
   1. 答：当JAVA 程序违反了JAVA 的语义规则时，JAVA 虚拟机就会将发生的错误表示为一个异常。违反语义规则包括2 种情况。一种是JAVA 类库内置的语义检查。例如数组下标越界,会引发IndexOutOfBoundsException;访问null 的对象时会引发NullPointerException。另一种情况就是JAVA 允许程序员扩展这种语义检查，程序员可以创建自己的异常，并自由选择在何时用throw 关键字引发异常。所有的异常都是java.lang.Thowable 的子类。
5. **error 和exception 有什么区别? 【基础】**
   1. 答：error 表示系统级的错误和程序不必处理的异常，是恢复不是不可能但很困难的情况下的一种严重问题；比如内存溢出，不可能指望程序能处理这样的情况；exception 表示需要捕捉或者需要程序进行处理的异常，是一种设计或实现问题；也就是说，它表示如果程序运行正常，从不会发生的情况。
6. **try {}里有一个return 语句，那么紧跟在这个try 后的finally {}里的code会不会被执行，什么时候被执行，在return 前还是后? 【基础】**
   1. 答：会执行，在return 前执行。
7. **JAVA 语言如何进行异常处理，关键字：throws,throw,try,catch,finally分别代表什么意义？在try 块中可以抛出异常吗？【基础】**
   1. 答：Java 通过面向对象的方法进行异常处理，把各种不同的异常进行分类，并提供了良好的接口。在Java 中，每个异常都是一个对象，它是Throwable 类或其它子类的实例。当一个方法出现异常后便抛出一个异常对象，该对象中包含有异常信息，调用这个对象的方法可以捕获到这个异常并进行处理。Java 的异常处理是通过5 个关键词来实现的：try、catch、throw、throws 和finally。一般情况下是用try 来执行一段程序，如果出现异常，系统会抛出（throws）一个异常，这时候你可以通过它的类型来捕捉（catch）它，或最后（finally）由缺省处理器来处理；try 用来指定一块预防所有“异常”的程序；catch 子句紧跟在try 块后面，用来指定你想要捕捉的“异常”的类型；throw 语句用来明确地抛出一个“异常”；throws 用来标明一个成员函数可能抛出的各种“异常”；Finally 为确保一段代码不管发生什么“异常”都被执行一段代码；可以在一个成员函数调用的外面写一个try 语句，在这个成员函数内部写另一个try 语句保护其他代码。每当遇到一个try 语句，“异常”的框架就放到堆栈上面，直到所有的try 语句都完成。如果下一级的try 语句没有对某种“异常”进行处理，堆栈就会展开，直到遇到有处理这种“异常”的try 语句。
8. **运行时异常与一般异常有何异同？【基础】**
   1. 答：异常表示程序运行过程中可能出现的非正常状态，运行时异常表示虚拟机的通常操作中可能遇到的异常，是一种常见运行错误。java 编译器要求方法必须声明抛出可能发生的非运行时异常，但是并不要求必须声明抛出未被捕获的运行时异常。
9. **给我一个你最常见到的runtime exception？【基础】**
   1. 答：
   2. ArithmeticException, ArrayStoreException, BufferOverflowException,
   3. BufferUnderflowException, CannotRedoException, CannotUndoException,
   4. ClassCastException, CMMException, ConcurrentModificationException,
   5. DOMException, EmptyStackException, IllegalArgumentException,
   6. IllegalMonitorStateException, IllegalPathStateException,
   7. IllegalStateException, ImagingOpException, IndexOutOfBoundsException,
   8. MissingResourceException, NegativeArraySizeException,
   9. NoSuchElementException, NullPointerException, ProfileDataException,
   10. ProviderException, RasterFormatException, SecurityException,
   11. SystemException, UndeclaredThrowableException,
   12. UnmodifiableSetException, UnsupportedOperationException
10. **final, finally, finalize 的区别? 【基础】**
    1. 答：final：修饰符（关键字）；如果一个类被声明为final，意味着它不能再派生出新的子类，不能作为父类被继承，因此一个类不能既被声明为abstract的，又被声明为final 的；将变量或方法声明为final，可以保证它们在使用中不被改变；被声明为final 的变量必须在声明时给定初值，而在以后的引用中只能读取，不可修改；被声明为final 的方法也同样只能使用，不能重载。
    2. finally：再异常处理时提供finally 块来执行任何清除操作；如果抛出一个异常，那么相匹配的catch 子句就会执行，然后控制就会进入finally 块（如果有的话）。
    3. finalize：方法名；Java 技术允许使用finalize() 方法在垃圾收集器将对象从内存中清除出去之前做必要的清理工作。这个方法是由垃圾收集器在确定这个对象没有被引用时对这个对象调用的。它是在Object 类中定义的，因此所有的类都继承了它。子类覆盖finalize() 方法以整理系统资源或者执行其他清理工作。finalize() 方法是在垃圾收集器删除对象之前对这个对象调用的。
11. **介绍JAVA 中的Collection FrameWork(及如何写自己的数据结构)【基础】**
    1. 答：Collection FrameWork 如下：
    2. Collection
    3. ├List
    4. │├LinkedList
    5. │├ArrayList
    6. │└Vector
    7. │ └Stack
    8. └Set
    9. Map
    10. ├Hashtable
    11. ├HashMap
    12. └WeakHashMap
    13. Collection 是最基本的集合接口，一个Collection 代表一组Object，即Collection 的元素（Elements）； Map 提供key 到value 的映射。
12. **List,Set,Map 是否继承自Collection 接口？【基础】**
    1. 答：List,Set 是；Map 不是。
13. **你所知道的集合类都有哪些？主要方法？【基础】**
    1. 答：最常用的集合类是List 和Map。List 的具体实现包括ArrayList 和Vector，它们是可变大小的列表，比较适合构建、存储和操作任何类型对象的元素列表。List 适用于按数值索引访问元素的情形。Map 提供了一个更通用的元素存储方法。Map 集合类用于存储元素对（称作“键”和“值”），其中每个键映射到一个值。
14. **说出ArrayList,Vector, LinkedList 的存储性能和特性？【基础】**
    1. 答：ArrayList 和Vector 都是使用数组方式存储数据，此数组元素数大于实际存储的数据以便增加和插入元素，它们都允许直接按序号索引元素，但是插入元素要涉及数组元素移动等内存操作，所以索引数据快而插入数据慢，Vector 由于使用了synchronized 方法（线程安全），通常性能上较ArrayList 差，而LinkedList 使用双向链表实现存储，按序号索引数据需要进行前向或后向遍历，但是插入数据时只需要记录本项的前后项即可，所以插入速度较快。
15. **Collection 和Collections 的区别？【基础】**
    1. 答：Collection 是java.util 下的接口，它是各种集合的父接口，继承于它的接口主要有Set 和List；Collections 是个java.util 下的类，是针对集合的帮助类，提供一系列静态方法实现对各种集合的搜索、排序、线程安全化等操作。
16. **HashMap 和Hashtable 的区别? 【基础】**
    1. 答：二者都实现了Map 接口，是将惟一键映射到特定的值上；主要区别在于：
    2. 1)HashMap 没有排序，允许一个null 键和多个null 值,而Hashtable 不允许；
    3. 2)HashMap 把Hashtable 的contains 方法去掉了，改成containsvalue 和containsKey,因为contains 方法容易让人引起误解；
    4. 3)Hashtable 继承自Dictionary 类，HashMap 是Java1.2 引进的Map 接口的实现；
    5. 4)Hashtable 的方法是Synchronize 的，而HashMap 不是，在多个线程访问Hashtable 时，不需要自己为它的方法实现同步，而HashMap 就必须为之提供外同步。Hashtable 和HashMap 采用的hash/rehash 算法大致一样，所以性能不会有很大的差异。
17. **Arraylist 与Vector 区别？【基础】**
    1. 答：就ArrayList 与Vector 主要从二方面来说：
    2. 1）同步性：Vector 是线程安全的（同步），而ArrayList 是线程序不安全的；
    3. 2）数据增长：当需要增长时,Vector 默认增长一倍，而ArrayList 却是一半。
18. **List、Map、Set 三个接口，存取元素时，各有什么特点？【基础】**
    1. 答：List 以特定次序来持有元素，可有重复元素。Set 无法拥有重复元素,内部排序。Map 保存key-value 值，value 可多值。
19. **Set 里的元素是不能重复的，那么用什么方法来区分重复与否呢? 是用==还是equals()? 它们有何区别? 【基础】**
    1. 答：Set 里的元素是不能重复的，用equals ()方法来区分重复与否。覆盖equals()方法用来判断对象的内容是否相同，而”==”判断地址是否相等,用来决定引用值是否指向同一对象。
20. **sleep() 和wait() 有什么区别? 【基础】**
    1. 答：sleep 是线程类（Thread）的方法，导致此线程暂停执行指定时间，给执行机会给其他线程，但是监控状态依然保持，到时后会自动恢复。调用sleep 不会释放对象锁。wait 是Object 类的方法，对此对象调用wait 方法导致本线程放弃对象锁，进入等待此对象的等待锁定池，只有针对此对象发出notify 方法（或notifyAll）后本线程才进入对象锁定池准备获得对象锁进入运行状态。
21. **当一个线程进入一个对象的一个synchronized 方法后，其它线程是否可进入此对象的其它方法? 【基础】**
    1. 答：其它线程只能访问该对象的其它非同步方法，同步方法则不能进入。
22. **请说出你所知道的线程同步的方法。【基础】**
    1. 答：
    2. wait():使一个线程处于等待状态，并且释放所持有的对象的lock；sleep():使一个正在运行的线程处于睡眠状态，是一个静态方法，调用此方法要捕捉InterruptedException 异常；
    3. notify():唤醒一个处于等待状态的线程，注意的是在调用此方法的时候，并不能确切的唤醒某一个等待状态的线程，而是由JVM 确定唤醒哪个线程，而且不是按优先级；
    4. notityAll():唤醒所有处入等待状态的线程，注意并不是给所有唤醒线程一个对象的锁，而是让它们竞争。
23. **多线程有几种实现方法,都是什么?同步有几种实现方法,都是什么? 【基础】**
    1. 答：多线程有两种实现方法，分别是继承Thread 类与实现Runnable 接口,同步的实现方面有两种，分别是synchronized,wait 与notify。
24. **同步和异步有何异同，在什么情况下分别使用他们？举例说明。【基础】**
    1. 答：如果数据将在线程间共享。例如正在写的数据以后可能被另一个线程读到，或者正在读的数据可能已经被另一个线程写过了，那么这些数据就是共享数据，必须进行同步存取。当应用程序在对象上调用了一个需要花费很长时间来执行的方法，并且不希望让程序等待方法的返回时，就应该使用异步编程，在很多情况下采用异步途径往往更有效率。
25. **启动一个线程是用run()还是start()?【基础】**
    1. 答：启动一个线程是调用start()方法，使线程所代表的虚拟处理机处于可运行状态，这意味着它可以由JVM 调度并执行。这并不意味着线程就会立即运行。run()方法可以产生必须退出的标志来停止一个线程。
26. **线程的基本概念、线程的基本状态以及状态之间的关系？【基础】**
    1. 答：线程指在程序执行过程中，能够执行程序代码的一个执行单位，每个程序至少都有一个线程，也就是程序本身；Java 中的线程有四种状态分别是：运行、就绪、挂起、结束。
27. **简述synchronized 和java.util.concurrent.locks.Lock 的异同？【中等难度】**
    1. 答：主要相同点：Lock 能完成synchronized 所实现的所有功能；主要不同点：Lock 有比synchronized 更精确的线程语义和更好的性能。synchronized 会自动释放锁，而Lock 一定要求程序员手工释放，并且必须在finally 从句中释放。
28. **java 中有几种方法可以实现一个线程？用什么关键字修饰同步方法?stop()和suspend()方法为何不推荐使用？【中等难度】**
    1. 答：有两种实现方法，分别是继承Thread 类与实现Runnable 接口；用synchronized 关键字修饰同步方法；反对使用stop()，是因为它不安全。它会解除由线程获取的所有锁定，而且如果对象处于一种不连贯状态，那么其他线程能在那种状态下检查和修改它们。结果很难检查出真正的问题所在；suspend()方法容易发生死锁。调用suspend()的时候，目标线程会停下来，但却仍然持有在这之前获得的锁定。此时，其他任何线程都不能访问锁定的资源，除非被“挂起”的线程恢复运行。对任何线程来说，如果它们想恢复目标线程，同时又试图使用任何一个锁定的资源，就会造成死锁。故不应该使用suspend()，而应在自己的Thread 类中置入一个标志，指出线程应该活动还是挂起。若标志指出线程应该挂起，便用wait()命其进入等待状态。若标志指出线程应当恢复，则用一个notify()重新启动线程。
29. **在监视器(Monitor)内部，是如何做线程同步的？程序应该做哪种级别的同步？**
    1. 监视器和锁在Java虚拟机中是一块使用的。监视器监视一块同步代码块，确保一次只有一个线程执行同步代码块。每一个监视器都和一个对象引用相关联。线程在获取锁之前不允许执行同步代码。
30. **什么是迭代器(Iterator)？**
    1. Iterator接口提供了很多对集合元素进行迭代的方法。每一个集合类都包含了可以返回迭代器实例的  
       迭代方法。迭代器可以在迭代的过程中删除底层集合的元素。
    2. 克隆(cloning)或者是序列化(serialization)的语义和含义是跟具体的实现相关的。因此，应该由集合类的具体实现来决定如何被克隆或者是序列化。
31. **Iterator和ListIterator的区别是什么？**
    1. 下面列出了他们的区别：
    2. Iterator可用来遍历Set和List集合，但是ListIterator只能用来遍历List。
    3. Iterator对集合只能是前向遍历，ListIterator既可以前向也可以后向。
    4. ListIterator实现了Iterator接口，并包含其他的功能，比如：增加元素，替换元素，获取前一个和后一个元素的索引，等等。
32. **数组(Array)和列表(ArrayList)有什么区别？什么时候应该使用Array而不是ArrayList？**
    1. 下面列出了Array和ArrayList的不同点：
    2. Array可以包含基本类型和对象类型，ArrayList只能包含对象类型。
    3. Array大小是固定的，ArrayList的大小是动态变化的。
    4. ArrayList提供了更多的方法和特性，比如：addAll()，removeAll()，iterator()等等。
    5. 对于基本类型数据，集合使用自动装箱来减少编码工作量。但是，当处理固定大小的基本数据类型的时候，这种方式相对比较慢。
33. **ArrayList和LinkedList有什么区别？**
    1. ArrayList和LinkedList都实现了List接口，他们有以下的不同点：
    2. ArrayList是基于索引的数据接口，它的底层是数组。它可以以O(1)时间复杂度对元素进行随机访问。与此对应，LinkedList是以元素列表的形式存储它的数据，每一个元素都和它的前一个和后一个元素链接在一起，在这种情况下，查找某个元素的时间复杂度是O(n)。
    3. 相对于ArrayList，LinkedList的插入，添加，删除操作速度更快，因为当元素被添加到集合任意位置的时候，不需要像数组那样重新计算大小或者是更新索引。
    4. LinkedList比ArrayList更占内存，因为LinkedList为每一个节点存储了两个引用，一个指向前一个元素，一个指向下一个元素。
    5. 也可以参考ArrayList vs. LinkedList。
34. **Comparable和Comparator接口是干什么的？列出它们的区别。**
    1. Java提供了只包含一个compareTo()方法的Comparable接口。这个方法可以个给两个对象排序。具体来说，它返回负数，0，正数来表明输入对象小于，等于，大于已经存在的对象。
    2. Java提供了包含compare()和equals()两个方法的Comparator接口。compare()方法用来给两个输入参数排序，返回负数，0，正数表明第一个参数是小于，等于，大于第二个参数。equals()方法需要一个对象作为参数，它用来决定输入参数是否和comparator相等。只有当输入参数也是一个comparator并且输入参数和当前comparator的排序结果是相同的时候，这个方法才返回true。
35. **HashSet和TreeSet有什么区别？**
    1. HashSet是由一个hash表来实现的，因此，它的元素是无序的。add()，remove()，contains()方法的时间复杂度是O(1)。
    2. 另一方面，TreeSet是由一个树形的结构来实现的，它里面的元素是有序的。因此，add()，remove()，contains()方法的时间复杂度是O(logn)。
    3. 垃圾收集器(Garbage Collectors)
36. **System.gc()和Runtime.gc()会做什么事情？**
    1. 这两个方法用来提示JVM要进行垃圾回收。但是，立即开始还是延迟进行垃圾回收是取决于JVM的。
37. **sleep()和wait()的区别?**
    1. 答: 1、这两个方法来自不同的类分别是Thread和Object
    2. 2、最主要是sleep方法没有释放锁，而 wait 方法释放了锁，使得其他线程可以使用同步控制块或者方法。
    3. 3、wait，notify和notifyAll只能在同步控制方法或者同步控制块里面使用，而sleep可以在任何地方使用（使用范围）
    4. 4、sleep必须捕获异常，而wait，notify和notifyAll不需要捕获异常
38. **io流 按照流向分 按照处理内容分? 字节流都有哪些方法?**
    1. 答:【流的分类详细】
    2. [ 输入流---输出流 ]
    3. 按照流的流向，可以将流分为输入流和输出流。
    4. 输入流：从其中读取数据
    5. 输出流：向其种写入数据
    6. 注意：划分输入/输出流是从程序运行时的所在的内存的角度来划分的。
    7. 输入流常用的基础类：InputStream和Reader
    8. 输出流常用的基础类：OutputStream和Writer
    9. 注意：这些积累都是抽象类，无法直接实例化。
    10. [ 字节流---字符流 ]
    11. 字节流和字符流的内的方法几乎一样，区别仅仅是其操作的数据单元大小不同，
    12. 字节流的数据操作单元： 8位的字节
    13. 字符流的数据操作单元： 16位的字节
    14. 字节流常用的基础类：InputStream和OutputStream
    15. 字符流常用的基础类：Reader和Writer
39. **解释一下static关键字被static修饰方法调用静态/非静态函数的方法**
    1. 答: 1. 静态方法
    2. 通常，在一个类中定义一个方法为static，那就是说，无需本类的对象即可调用此方法
    3. 声明为static的方法有以下几条限制：
    4. · 它们仅能调用其他的static 方法。
    5. · 它们只能访问static数据。
    6. · 它们不能以任何方式引用this 或super。
    7. 2. 静态变量
    8. 声明为static的变量实质上就是全局变量。当声明一个对象时，并不产生static变量的拷贝，而是该类所有的实例变量共用同一个static变量。静态变量与静态方法类似。所有此类实例共享此静态变量，也就是说在类装载时，只分配一块存储空间，所有此类的对象都可以操控此块存储空间，当然对于final则另当别论了
    9. 3. static代码块
    10. static关键字还有一个比较关键的作用就是 用来形成静态代码块以优化程序性能。static块可以置于类中的任何地方，类中可以有多个static块。在类初次被加载的时候，会按照static块的顺序来执行每个static块，并且只会执行一次。
40. **堆区和栈区的区别**
    1. 答: 栈区：先进后出 存放局部变量 连续的存储空间
    2. 堆区：存放new出来的对象，不连续的存储空间
41. **Java支持的数据类型有哪些？什么是自动拆装箱？** 
    1. 答: Java语言支持的8种基本数据类型是：
    2. byte/short/int/long/float/double/boolean/char
    3. 自动装箱是Java编译器在基本数据类型和对应的对象包装类型之间做的一个转化。比如：把int转化成Integer，double转化成Double，等等。反之就是自动拆箱。
42. **char型变量中能不能存贮一个中文汉字？为什么？**
    1. java采用unicode，2个字节（16位）来表示一个字符， 无论是汉字还是数字字母，或其他语言。char 在java中是2个字节。所以可以存储中文

## 第二部分：Hadoop

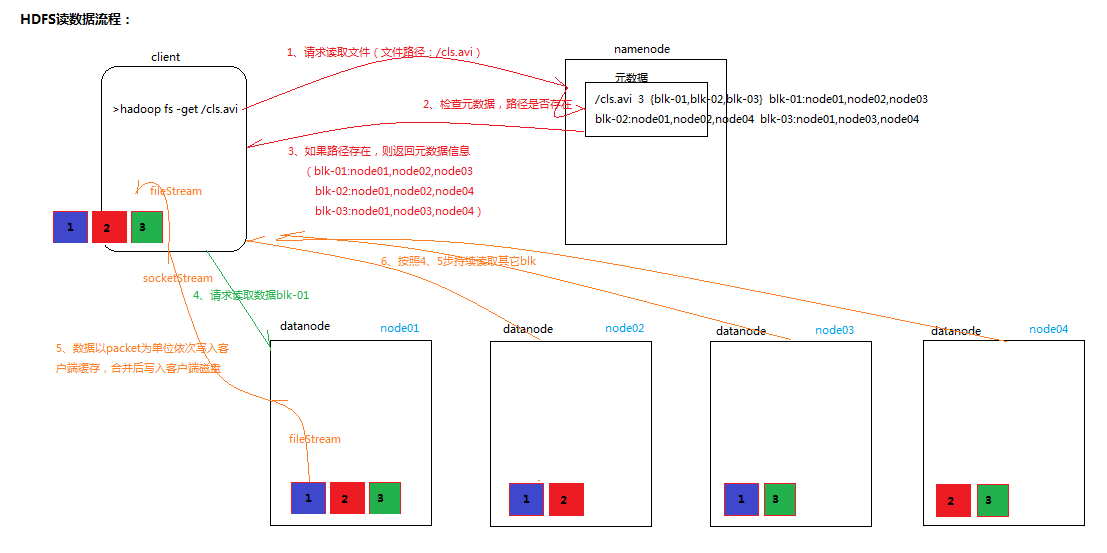
1. **通常情况下集群模式的瓶颈在哪?**
   1. 首先，瓶颈一般是指在整体中的关键限制因素，磁盘IO是指数据往磁盘读写，现在科技速度最快的属[固态硬盘](https://www.baidu.com/s?wd=%E5%9B%BA%E6%80%81%E7%A1%AC%E7%9B%98&tn=SE_PcZhidaonwhc_ngpagmjz&rsv_dl=gh_pc_zhidao%22%20%5Ct%20%22https://zhidao.baidu.com/question/_blank)了，读的速度很大有1G/秒左右，但是写入速度最快几百兆/秒，集群中数据在cpu和内存之间速度快的可以忽略，处理速度也可以忽略，相对这些速度，磁盘读写就显得慢了，旁贷一下现在好一点的数据库oracle存储数据都是写日志先暂存然后等机器空闲再写入到磁盘，这些都是为了提高效率，不然执行一条操作等半天
2. **SecondaryNameNode的作用是什么?**
   1. SecondaryNameNode它的职责是合并NameNode的edit logs到fsimage文件中
   2. 首先，SecondaryNameNode定时到NameNode去获取edit logs，并更新到fsimage上。一旦它有了新的fsimage文件，SecondaryNameNode将其拷贝回NameNode中。NameNode在下次重启时会使用这个新的fsimage文件，从而减少重启的时间。
   3. Secondary NameNode的整个目的是在HDFS中提供一个检查点。它只是NameNode的一个助手节点。这也是它在社区内被认为是检查点节点的原因。
3. **请用map,reduce实现wordCount?**
   1. public class WordCount {
   2. public static class MyMapper extends Mapper<Object, Text, Text, IntWritable>{
   3. private final static IntWritable one = new IntWritable(1);
   4. private Text word = new Text();
   5. @Override
   6. protected void map(Object key, Text value,Context context)
   7. throws IOException, InterruptedException {
   8. StringTokenizer itr = new StringTokenizer(value.toString());
   9. while (itr.hasMoreTokens()) {
   10. word.set(itr.nextToken());
   11. context.write(word, one);
   12. }
   13. }
   14. }
   15. public static class MyReducer extends Reducer<Text, IntWritable, Text, IntWritable>{
   16. private IntWritable result = new IntWritable();
   17. @Override
   18. protected void reduce(Text key, Iterable<IntWritable> values, Context context)
   19. throws IOException, InterruptedException {
   20. int sum = 0;
   21. for (IntWritable val : values) {
   22. sum += val.get();
   23. }
   24. result.set(sum);
   25. context.write(key, result);
   26. }
   27. }
   28. public static void main(String[] args) throws Exception {
   29. Configuration conf = new Configuration();
   30. Job job = Job.getInstance(conf, "word count");
   31. job.setJarByClass(WordCount.class);
   32. job.setMapperClass(MyMapper.class);
   33. job.setCombinerClass(MyReducer.class);
   34. job.setReducerClass(MyReducer.class);
   35. job.setOutputKeyClass(Text.class);
   36. job.setOutputValueClass(IntWritable.class);
   37. FileInputFormat.addInputPath(job, new Path(args[0]));
   38. FileOutputFormat.setOutputPath(job, new Path(args[1]));
   39. System.exit(job.waitForCompletion(true)?0:1);
   40. }
   41. }
4. **Hadoop的核心配置是什么？**
   1. Hadoop的核心配置通过四个xml文件来完成：
   2. 1. core-site.xml；2. HDFS-site.xml；3. mapred-site.xml；4. yarn-site.xml。
   3. 这些文件都使用xml格式，因此每个xml中都有一些属性，包括名称和值。
5. **“jps”命令的用处？**
   1. 显示当前所有Java进程pid
6. **MapReduce的原理?**

****

1. **HDFS存储的机制?**



* 1. 写流程：
  2. 1. Client连接NameNode请求写数据
  3. 2. NameNode记录一条数据位置信息（元数据），告诉Client存哪。
  4. 3. Client用HDFS的API将数据块（默认是128M）存储到DataNode上。
  5. 4. DataNode将数据水平备份，并且备份完将反馈给Client。
  6. 5. Client通知NameNode存储块完毕。
  7. 6. NameNode将元数据同步到内存中。
  8. 7. 另一块循环上面的过程。



* 1. 读流程：
  2. 1. Client链接NameNode，查看元数据，找到数据的存储位置。
  3. 2. Client通过HDFS的API并发读取数据。
  4. 3. 关闭连接。

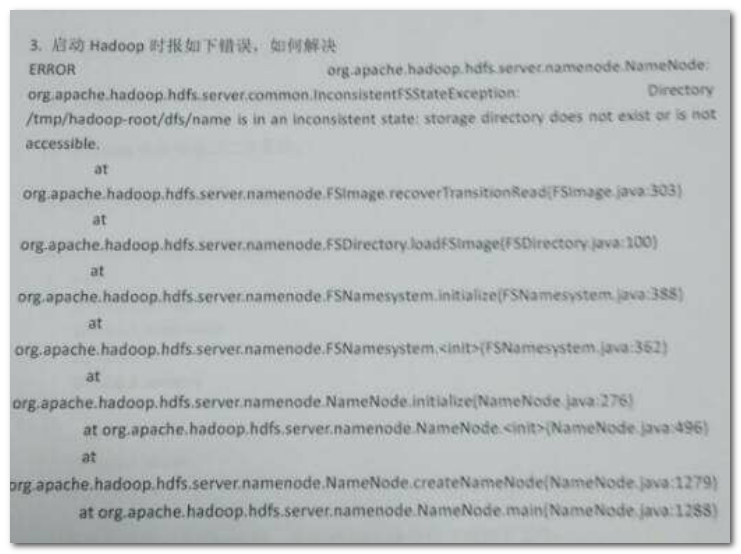
1. **举一个简单的例子说明MapReduce是怎么来运行的 ?**
   1. WordCount的例子
2. **用MapReduce来实现下面需求？现在有10个文件夹,每个文件夹都有1000000个url.现在让你找出top1000000url。**

topk

1. **Hadoop中Combiner的作用?**
   1. Combiner是Reduce的实现，在Map端运行计算任务，减少Map端的输出数据。
   2. 作用就是优化。
   3. 但是Combiner的使用场景是MapReduce的Map和Reduce输入输出一样
2. **简述Hadoop安装**



1. **请列出正常工作的Hadoop集群中Hadoop都分别需要启动哪些进程，它们的作用分别是什么？**
   1. NameNode：管理集群，并记录DataNode文件信息。
   2. SecondaryNameNode：主要做合并元数据。也可以做冷备，对一定范围内数据做快照性备份。
   3. DataNode：存储数据。
   4. ResourceManager：负责集群中所有资源的统一管理和分配，它接收来自各个节点（NodeManager）的资源汇报信息，并把这些信息按照一定的策略分配给各个应用程序（实际上是ApplicationManager）。
   5. NodeManager：是YARN中每个节点上的代理，它管理Hadoop集群中单个计算节点，包括与ResourceManger保持通信，监督Container的生命周期管理，监控每个Container的资源使用（内存、CPU等）情况，追踪节点健康状况，管理日志和不同应用程序用到的附属服务。
2. **解决下面的错误**



1. 权限问题，可能曾经用root启动过集群。
2. 可能是文件夹不存在
3. **写出下面的命令**

杀死一个job

Hadoop job –list 拿到jobid，Hadoop job –kill jobid

删除HDFS上的/tmp/aaa目录

Hadoop fs –rm –r /tmp/aaa

加入一个新的存储节点、删除一个计算节点（需要刷新集群状态命令）

加新节点：

Hadoop-daemon.sh start datanode Hadoop-daemon.sh start nodemanager

均衡数据节点上的数据：

./bin/start-balancer.sh

删除节点：

Hadoop mradmin –refreshnodes Hadoop dfsadmin -refreshnodes

1. **简述Hadoop的调度器**
   1. FIFO Scheduler：默认，先进先出的原则。
   2. Capacity Scheduler：计算能力调度器，选择占用最小、优先级高的先执行，以此类推。
   3. Fair Scheduler：公平调度，所有的Job具有相同的资源。
2. **Hive有哪些保存元数据的方式，每个有什么特点。**
   1. 内嵌数据库derby，安装小，基于Java、JDBC和SQL标准。
   2. MySQL数据库，数据存储模式可以自己设置，持久化好，查看方便。
3. **MapReduce中Combiner和Partition的作用**
   1. Combiner是Reduce的实现，在Map端运行计算任务，减少Map端的输出数据。
   2. 作用就是优化。
   3. 但是Combiner的使用场景是MapReduce的Map输出结果必须和Reduce输入输出一样。
   4. Partition的默认实现是HashPartition，是Map端将数据按照Reduce个数取余，进行分区，不同的Reduce来Copy自己的数据。
   5. Partiton的作用是将数据分到不同的Reduce进行计算，加快计算效果
4. **Hive内部表和外部表的区别**
   1. 内部表：加载数据到Hive所在的HDFS目录，删除时，元数据和数据文件都删除
   2. 外部表：不加载数据到Hive所在的HDFS目录，删除时，只删除表结构。
5. **HBASE的rowkey怎么创建好？列族怎么创建比较好?**
   1. HBASE存储时，数据按照Row key的字典序(byte order)排序存储。设计key时，要充分排序存储这个特性，将经常一起读取的行存储放到一起。(位置相关性)
   2. 一个列族在数据底层是一个文件，所以将经常一起查询的列放到一个列族中，列族尽量少，减少文件的寻址时间。
6. **用MapReduce怎么处理数据倾斜问题？**
   1. 数据倾斜：map /reduce程序执行时，reduce节点大部分执行完毕，但是有一个或者几个reduce节点运行很慢，导致整个程序的处理时间很长，这是因为某一个key的条数比其他key多很多（有时是百倍或者千倍之多），这条key所在的reduce节点所处理的数据量比其他节点就大很多，从而导致某几个节点迟迟运行不完，此称之为数据倾斜。
   2. 用Hadoop程序进行数据关联时，常碰到数据倾斜的情况，这里提供一种解决方法。
   3. 自己实现partition类，用key和value相加取hash值：
   4. 方式1：
   5. 源代码：
   6. public int getPartition(K key, V value, int numReduceTasks) {
   7. return (key.hashCode() & Integer.MAX\_VALUE % numReduceTasks);
   8. }
   9. 修改后
   10. public int getPartition(K key, V value,int numReduceTasks) {
   11. return ((（key).hashCode()+value.hashCode()） & Integer.MAX\_VALUE、 % numReduceTasks;
   12. }
   13. 方式2：
   14. public class HashPartitioner<K, V> extends Partitioner<K, V> {

private int aa= 0;

public int getPartition(K key, V value,int numReduceTasks) {

return (key.hashCode()+(aa++) & Integer.MAX\_VALUE、 % numReduceTasks;

* 1. }

1. **Hadoop框架怎么优化**
   1. （1） 从应用程序角度进行优化。由于MapReduce是迭代逐行解析数据文件的，怎样在迭代的情况下，编写高效率的应用程序，是一种优化思路。
   2. （2） 对Hadoop参数进行调优。当前Hadoop系统有190多个配置参数，怎样调整这些参数，使Hadoop作业运行尽可能的快，也是一种优化思路。
   3. （3） 从系统实现角度进行优化。这种优化难度是最大的，它是从Hadoop实现机制角度，发现当前Hadoop设计和实现上的缺点，然后进行源码级地修改。该方法虽难度大，但往往效果明显。
   4. （4）Linux内核参数调整
   5. 从应用程序角度进行优化
   6. （1） 避免不必要的reduce任务
   7. 如果MapReduce程序中reduce是不必要的，那么我们可以在map中处理数据, Reducer设置为0。这样避免了多余的reduce任务。
   8. （2） 为job添加一个Combiner
   9. 为job添加一个combiner可以大大减少shuffle阶段从map task拷贝给远程reduce task的数据量。一般而言，combiner与reducer相同。
   10. （3） 根据处理数据特征使用最适合和简洁的Writable类型
   11. Text对象使用起来很方便，但它在由数值转换到文本或是由UTF8字符串转换到文本时都是低效的，且会消耗大量的CPU时间。当处理那些非文本的数据时，可以使用二进制的Writable类型，如IntWritable， FloatWritable等。二进制writable好处：避免文件转换的消耗；使map task中间结果占用更少的空间。
   12. （4） 重用Writable类型
   13. 很多MapReduce用户常犯的一个错误是，在一个map/reduce方法中为每个输出都创建Writable对象。例如，你的Wordcout mapper方法可能这样写：
   14. public void map(...) {
   15. …
   17. for (String word : words) {
   18. output.collect(new Text(worD、, new IntWritable(1));
   19. }
   20. }
   21. 这样会导致程序分配出成千上万个短周期的对象。Java垃圾收集器就要为此做很多的工作。更有效的写法是：
   22. class MyMapper … {
   23. Text wordText = new Text();
   24. IntWritable one = new IntWritable(1);
   25. public void map(...) {
   26. for (String word: words) {
   27. wordText.set(worD、;
   28. output.collect(wordText, onE、;
   29. }
   30. }
   31. }
   32. （5） 使用StringBuffer而不是String
   33. 当需要对字符串进行操作时，使用StringBuffer而不是String，String是read-only的，如果对它进行修改，会产生临时对象，而StringBuffer是可修改的，不会产生临时对象。
   34. （1） 避免不必要的reduce任务
   35. 如果MapReduce程序中reduce是不必要的，那么我们可以在map中处理数据, Reducer设置为0。这样避免了多余的reduce任务。
   36. （2） 为job添加一个Combiner
   37. 为job添加一个combiner可以大大减少shuffle阶段从map task拷贝给远程reduce task的数据量。一般而言，combiner与reducer相同。
   38. （3） 根据处理数据特征使用最适合和简洁的Writable类型
   39. Text对象使用起来很方便，但它在由数值转换到文本或是由UTF8字符串转换到文本时都是低效的，且会消耗大量的CPU时间。当处理那些非文本的数据时，可以使用二进制的Writable类型，如IntWritable， FloatWritable等。二进制writable好处：避免文件转换的消耗；使map task中间结果占用更少的空间。
   40. （4） 重用Writable类型
   41. 很多MapReduce用户常犯的一个错误是，在一个map/reduce方法中为每个输出都创建Writable对象。例如，你的Wordcout mapper方法可能这样写：
   42. public void map(...) {
   43. …
   45. for (String word : words) {
   46. output.collect(new Text(worD、, new IntWritable(1));
   47. }
   48. }
   49. 这样会导致程序分配出成千上万个短周期的对象。Java垃圾收集器就要为此做很多的工作。更有效的写法是：
   50. class MyMapper … {
   51. Text wordText = new Text();
   52. IntWritable one = new IntWritable(1);
   53. public void map(...) {
   54. for (String word: words) {
   55. wordText.set(worD、;
   56. output.collect(wordText, onE、;
   57. }
   58. }
   59. }
   60. （5） 使用StringBuffer而不是String
   61. 当需要对字符串进行操作时，使用StringBuffer而不是String，String是read-only的，如果对它进行修改，会产生临时对象，而StringBuffer是可修改的，不会产生临时对象。
   62. 对参数进行调优
   63. 查看linux的服务，可以关闭不必要的服务
   64. ntsysv
   65. 停止打印服务
   66. #/etc/init.d/cups stop
   67. #chkconfig cups off
   68. 关闭ipv6
   69. #vim /etc/modprobe.conf
   70. 添加内容
   71. alias net-pf-10 off
   72. alias ipv6 off
   73. 调整文件最大打开数
   74. 查看： ulimit -a    结果：open files (-n) 1024
   75. 临时修改： ulimit -n 4096
   76. 持久修改：
   77. vi /etc/security/limits.conf在文件最后加上：
   78. \* soft nofile 65535 \* hard nofile 65535 \* soft nproc 65535 \* hard nproc 65535
   79. 修改linux内核参数
   80. vi /etc/sysctl.conf
   81. 添加
   82. net.core.somaxconn = 32768
   83. #web应用中listen函数的backlog默认会给我们内核参数的net.core.somaxconn限制到128，而nginx定义的NGX\_LISTEN\_BACKLOG默认为511，所以有必要调整这个值。
   84. 调整swap分区什么时候使用：
   85. 查看：cat /proc/sys/vm/swappiness
   86. 设置：vi /etc/sysctl.conf
   87. 在这个文档的最后加上这样一行: vm.swappiness=10
   88. 表示物理内存使用到90%（100-10=90）的时候才使用swap交换区
   89. 关闭noatime
   90. vi /etc/fstab
   91. /dev/sda2    /data     ext3  noatime,nodiratime  0 0
   92. 设置readahead buffer
   93. blockdev --setra READAHEAD 512 /dev/sda
   94. 一下是修改mapred-site.xml文件
   95. 修改最大槽位数
   96. 槽位数是在各个tasktracker上的mapred-site.xml上设置的，默认都是2
   97. <property>
   98. <name>mapred.tasktracker.map.tasks.maximum</name>  #++++maptask的最大数
   99. <value>2</value>
   100. </property>
   101. <property>
   102. <name>mapred.tasktracker.reduce.tasks.maximum</name>  #++++reducetask的最大数
   103. <value>2</value>
   104. </property>
   105. 调整心跳间隔
   106. 集群规模小于300时，心跳间隔为300毫秒
   107. MapReduce.jobtracker.heartbeat.interval.min  心跳时间
   108. mapred.heartbeats.in.second  集群每增加多少节点，时间增加下面的值
   109. MapReduce.jobtracker.heartbeat.scaling.factor 集群每增加上面的个数，心跳增多少
   110. 启动带外心跳
   111. MapReduce.tasktracker.outofband.heartbeat  默认是false
   112. 配置多块磁盘
   113. MapReduce.local.dir
   114. 配置RPC hander数目
   115. mapred.job.tracker.handler.count 默认是10，可以改成50，根据机器的能力
   116. 配置HTTP线程数目
   117. tasktracker.http.threads  默认是40，可以改成100 根据机器的能力
   118. 选择合适的压缩方式
   119. 以snappy为例：
   120. <property>
   121. <name>mapred.compress.map.output</name>
   122. <value>true</value>
   123. </property>
   124. <property>
   125. <name>mapred.map.output.compression.codec</name>
   126. <value>org.apache.Hadoop.io.compress.SnappyCodec</value>
   127. </property>
   128. 启用推测执行机制
   129. 推测执行(Speculative Execution)是指在分布式集群环境下，因为程序BUG，负载不均衡或者资源分布不均等原因，造成同一个job的多个task运行速度不一致，有的task运行速度明显慢于其他task（比如：一个job的某个task进度只有10%，而其他所有task已经运行完毕），则这些task拖慢了作业的整体执行进度，为了避免这种情况发生，Hadoop会为该task启动备份任务，让该speculative task与原始task同时处理一份数据，哪个先运行完，则将谁的结果作为最终结果。
   130. 推测执行优化机制采用了典型的以空间换时间的优化策略，它同时启动多个相同task（备份任务）处理相同的数据块，哪个完成的早，则采用哪个task的结果，这样可防止拖后腿Task任务出现，进而提高作业计算速度，但是，这样却会占用更多的资源，在集群资源紧缺的情况下，设计合理的推测执行机制可在多用少量资源情况下，减少大作业的计算时间。
   131. mapred.map.tasks.speculative.execution  默认是true
   132. mapred.rduce.tasks.speculative.execution  默认是true
   133. 设置是失败容忍度
   134. mapred.max.map.failures.percent   作业允许失败的map最大比例  默认值0，即0%
   135. mapred.max.reduce.failures.percent  作业允许失败的reduce最大比例  默认值0，即0%
   136. mapred.map.max.attemps  失败后最多重新尝试的次数 默认是4
   137. mapred.reduce.max.attemps  失败后最多重新尝试的次数 默认是4
   138. 启动jvm重用功能
   139. mapred.job.reuse.jvm.num.tasks  默认值1，表示只能启动一个task，若为-1，表示可以最多运行数不限制
   140. 设置任务超时时间
   141. mapred.task.timeout  默认值600000毫秒，也就是10分钟。
   142. 合理的控制reduce的启动时间
   143. mapred.reduce.slowstart.completed.maps  默认值0.05  表示map任务完成5%时，开始启动reduce任务
   144. 跳过坏记录
   145. 当任务失败次数达到该值时，才会进入skip mode，即启用跳过坏记录数功能,也就是先试几次，不行就跳过
   146. mapred.skip.attempts.to.start.skipping 默认值 2
   147. map最多允许跳过的记录数
   148. mapred.skip.map.max.skip.records 默认值0，为不启用
   149. reduce最多允许跳过的记录数
   150. mapred.skip.reduce.max.skip.records 默认值0，为不启用
   151. 换记录存放的目录
   152. mapred.skip.out.dir  默认值${mapred.output.dir}/\_logs/
2. **我们开发job时，是否可以去掉reduce阶段。**
   1. 可以。设置reduce数为0 即可。
3. **datanode在什么情况下不会备份**
   1. datanode在强制关闭或者非正常断电不会备份。
4. **combiner出现在哪个过程**
   1. 出现在map阶段的map方法后等。
5. **HDFS的体系结构**
   1. HDFS有namenode、secondraynamenode、datanode组成。
   2. 为n+1模式
   3. namenode负责管理datanode和记录元数据
   4. secondraynamenode负责合并日志
   5. datanode负责存储数据
6. **3个datanode中有一个datanode出现错误会怎样？**
   1. 这个datanode的数据会在其他的datanode上重新做备份。
7. **描述一下Hadoop中，有哪些地方使用了缓存机制，作用分别是什么？**
   1. 在MapReduce提交job的获取id之后，会将所有文件存储到分布式缓存上，这样文件可以被所有的MapReduce共享。
8. **如何确定Hadoop集群的健康状态**
   1. 通过页面监控,脚本监控。
9. **生产环境中为什么建议使用外部表**
   1. 1、因为外部表不会加载数据到Hive，减少数据传输、数据还能共享。
   2. 2、Hive不会修改数据，所以无需担心数据的损坏
   3. 3、删除表时，只删除表结构、不删除数据。
10. **如何检查NameNode是否正常运行？重启NameNode的命令是什么？**
    1. 通过节点信息和浏览器查看，通过脚本监控
    2. Hadoop-deamon.sh start namenode
    3. HDFS-deamon.sh start namenode
11. **避免NameNode故障导致集群宕机的解决方法是什么？**
    1. 自己书写脚本监控重启
12. **HBase数据库对行健的设计要求是什么？**
    1. 行健以字典序排列，设计时充分利用这个特点，将经常一起查询的行健设计在一起，
    2. 例如时间戳结尾，用户名开头（位置相关性）
13. **你们数据库怎么导入Hive 的,有没有出现问题？**
    1. 在导入hive的时候，如果数据库中有blob或者text字段，会报错，解决方案：
    2. clob：在将数据由Oracle数据库导入到Hive时，发现带有clob字段的表的数据会错乱，出现一些字段全为NULL的空行。
    3. 由于在项目中CLOB字段没有实际的分析用途，因此考虑将CLOB字段去掉。
    4. 同时，为了防止CLOB字段产生一些问题，因此将HIVE中CLOB字段禁用，禁用的方式如下：
    5. [Hadoop@master sqoop-1.4.5]$ cd $SQOOP\_HOME/conf
    6. [hadoop@master conf]$ vi oraoop-site.xml
    7. 将以下属性的注释去掉，并且将value改为true
    8. <property>
    9. <name>oraoop.import.omit.lobs.and.long</name>
    10. <value>true</value>
    11. <description>If true, OraOop will omit BLOB, CLOB, NCLOB and LONG columns during an Import.
    12. </description>
    13. </property>
    14. 有些表中虽然有clob字段，但是不能排除掉，因为其他字段使我们所需要，因此在导入的时候采用指定--columns的方式来进行导入
    15. sqoop import --hive-import --hive-database test --create-hive-table --connect jdbc --username user--password user
    16. --bindir //scratch --outdir /Java --table aaa --columns "ID,NAME" -m 1 --null-string '\\N' --null-non-string '\\N'
    17. text：缓存配置量增大
    18. sqoop import --connect jdbc:[MySQL](http://lib.csdn.net/base/mysql)://192.168.56.204:3306/sqoop --username [Hive](http://lib.csdn.net/base/hive) --password hive --table jobinfo --target-dir /sqoop/test5 --inline-lob-limit 16777216 --fields-terminated-by '\t' -m 1
14. **公司技术选型可能利用storm 进行实时计算,讲解一下storm**
    1. 描述下storm的设计模式，是基于work、excutor、task的方式运行代码，由spout、bolt组成等等
15. **一个datanode 宕机,怎么一个流程恢复**
    1. 将datanode数据删除，重新当成新节点加入即可。
16. **HBASE 的特性,以及你怎么去设计 rowkey 和 columnFamily ,怎么去建一个table**
    1. HBASE是列式数据库，rowkey是字典序的，设计时的规则同上。
    2. 每个列族是一个文件，将经常一起查询的列放到同一个列族中，减少文件的寻址时间。
17. **Redis,传统数据库,HBase,Hive 每个之间的区别**
    1. redis：分布式缓存，强调缓存，内存中数据
    2. 传统数据库：注重关系
    3. HBASE：列式数据库，无法做关系数据库的主外键，用于存储海量数据，底层基于HDFS
    4. Hive：数据仓库工具，底层是MapReduce。不是数据库，不能用来做用户的交互存储
18. **MapReduce 的 map 数量 和 reduce 数量 怎么确定 ,怎么配置**
    1. map的数量有数据块决定，reduce数量随便配置

1. **设计日志收集分析系统（日志分布在各个业务系统中，我们需要对当天的日志进行实时的汇总统计，同时又能按天查询历史的汇总数据，可以围绕PV、UV、IP等指标进行阐述）**
   1. 1. 通过flume将不同系统的日志收集到kafka中
   2. 2. 通过storm实时的处理PV、UV、IP
   3. 3. 通过kafka的consumer将日志生产到HBASE中。
   4. 4. 通过离线的MapReduce或者Hive，处理HBASE中的数据

1. **Hive语句实现WordCount（假设数据存在hadoop下，路径为：/home/hadoop/worddata 里面全是一堆单词）**
   1. 1. 建表
   2. 2. 分组（group by）统计wordcount
   3. select word,count(1) from table1 group by word;

1. **给定a、b两个文件，各存放50亿个url，每个url占用64字节，内存限制是4G，找出a、b两个文件共同的url**
   1. 1. 可以估计每个文件的大小为50亿×64=298G，远远大于内存限制的4G。所以不可能将其完全加载到内存中处理。考虑采取分而治之的方法。
   2. 2. 将文件存储到HDFS中，这样每个文件为64M或者是128M
   3. 3. 分别对两个文件的url进行去重、排序输出，这样能排除a文件中相同的url，b文件也一样
   4. 4. 对a、b两个文件处理后的结果进行wordcount，并且在reduce中判断单词个数，个数为2的时候输出，这样就找到了a、b文件中的相同url。
   5. 5.此计算步骤中的每一步加载到内存中的文件大小都不会超过64M，远远小于4G。
2. **实时数据统计会用到哪些技术？他们各自的应用场景和区别是什么？**
   1. flume：日志收集系统，主要用于系统日志的收集
   2. kafka：消息队列，进行消息的缓存和系统的解耦
   3. storm：实时计算框架，进行流式的计算。
3. **一个Hadoop环境，整合了HBASE和HIVE，是否有必要给HDFS和HBASE都分别配置压缩策略？请给出对压缩策略的建议。**
   1. HDFS在存储的时候不会将数据进行压缩，如果想进行压缩，我们可以在向HDFS上传数据的时候进行压缩。
   2. 采用压缩流
   3. //压缩文件
   4. public static void compress(String codecClassNamE、 throws Exception{
   5. Class<?> codecClass = Class.forName(codecClassNamE、;
   6. Configuration conf = new Configuration();
   7. FileSystem fs = FileSystem.get(conF、;
   8. CompressionCodec codec = (CompressionCodeC、ReflectionUtils.newInstance(codecClass, conF、;
   9. //指定压缩文件路径
   10. FSDataOutputStream outputStream = fs.create(new Path("/user/Hadoop/text.gz"));
   11. //指定要被压缩的文件路径
   12. FSDataInputStream in = fs.open(new Path("/user/Hadoop/aa.txt"));
   13. //创建压缩输出流
   14. CompressionOutputStream out = codec.createOutputStream(outputStream);
   15. IOUtils.copyBytes(in, out, conF、;
   16. IOUtils.closeStream(in);
   17. IOUtils.closeStream(out);
   18. }

采用序列化文件

public void testSeqWrite() throws Exception {

* 1. Configuration conf = new Configuration();// 创建配置信息
  2. conf.set("fs.default.name", "HDFS://master:9000");// HDFS默认路径
  3. conf.set("Hadoop.job.ugi", "Hadoop,Hadoop");// 用户和组信息
  4. String uriin = "HDFS://master:9000/ceshi2/";// 文件路径
  5. FileSystem fs = FileSystem.get(URI.create(uriin), conF、;// 创建filesystem
  6. Path path = new Path("HDFS://master:9000/ceshi3/test.seq");// 文件名
  7. IntWritable k = new IntWritable();// key，相当于int
  8. Text v = new Text();// value，相当于String
  9. SequenceFile.Writer w = SequenceFile.createWriter(fs, conf, path,
  10. k.getClass(), v.getClass());// 创建writer
  11. for (int i = 1; i < 100; i++) {// 循环添加
  12. k.set(i);
  13. v.set("abcd");
  14. w.append(k, v);
  15. }
  16. w.close();
  17. IOUtils.closeStream(w);// 关闭的时候flush
  18. fs.close();
  19. }
  20. HBASE为列存数据库，本身存在压缩机制，所以无需设计。

1. **简述Hbase filter的实现原理是什么？结合实际项目经验，写出几个使用filter的场景**
   1. HBASE的filter是通过scan设置的，所以是基于scan的查询结果进行过滤。
   2. 在进行订单开发的时候，我们使用rowkeyfilter过滤出某个用户的所有订单
   3. 在进行云笔记开发时，我们使用rowkey过滤器进行redis数据的恢复。
2. **简述HIVE中的虚拟列作用是什么？使用它的注意事项？**
   1. Hive提供了三个虚拟列：
   2. INPUT\_\_FILE\_\_NAME
   3. BLOCK\_\_OFFSET\_\_INSIDE\_\_FILE
   4. ROW\_\_OFFSET\_\_INSIDE\_\_BLOCK
   5. 但ROW\_\_OFFSET\_\_INSIDE\_\_BLOCK默认是不可用的，需要设置Hive.exec.rowoffset为true才可以。可以用来排查有问题的输入数据。
   6. INPUT\_\_FILE\_\_NAME, mapper任务的输出文件名。
   7. BLOCK\_\_OFFSET\_\_INSIDE\_\_FILE, 当前全局文件的偏移量。对于块压缩文件，就是当前块的文件偏移量，即当前块的第一个字节在文件中的偏移量。
   8. Hive> SELECT INPUT\_\_FILE\_\_NAME, BLOCK\_\_OFFSET\_\_INSIDE\_\_FILE, line
   9. > FROM Hive\_text WHERE line LIKE '%Hive%' LIMIT 2;
   10. har://file/user/Hive/warehouse/Hive\_text/folder=docs/
   11. data.har/user/Hive/warehouse/Hive\_text/folder=docs/README.txt 2243
   12. har://file/user/Hive/warehouse/Hive\_text/folder=docs/
   13. data.har/user/Hive/warehouse/Hive\_text/folder=docs/README.txt 3646
3. **如何存储海量的小文件（大小都是几百K~几M），请简述自己的设计方案**
   1. 将小文件打成har文件存储
   2. 将小文件序列化到HDFS中
4. **mapred找共同好友，数据格式如下**
5. **A B C D E F**
6. **B A C D E**
7. **C A B E**
8. **D A B E**
9. **E A B C D**
10. **F A**

**第一个字母是本人，其他是他的朋友，找出有共同好友的人，和共同好友是谁**

* 1. 思路：例如A，他的朋友是B\C\D\E\F\，那么BC的共同朋友就是A。所以将BC作为key，将A作为value，在map端输出即可！其他的朋友循环处理。

1. **随意使用各种类型的脚本语言实现：批量将指定目录下的所有文件中的$HADOOP\_HOME$替换成/home/ocetl/app/hadoop**
   1. #!/bin/bash
   2. ls $1 | while read line
   3. do
   4. sed -i 's,\$HADOOP\_HOME\$,\/home\/aa,g' $1$line
   5. echo $1$line
   6. done
   7. 将以上代码保存为aaa.sh
   8. 脚本执行命令：替换/home/Hadoop/test/下的所有文件
   9. ./aaa.sh /home/Hadoop/test/
2. **Hadoop的集群搭建步骤**
   1. a) 创建hadoop账号
   2. b) 更改ip
   3. c) 安装Java 更改/etc/profile 配置环境变量
   4. d) 修改host文件域名
   5. e) 安装ssh 配置无密码登录
   6. f) 解压hadoop
   7. g) 配置hadoop  conf下面的配置文件
   8. h) Hadoop namenode -format  格式化
   9. i) Start 启动
3. **hadoop生态体系的构成以及每个组件的作用**
   1. 重点组件：
   2. HDFS：分布式文件系统
   3. MAPREDUCE：分布式运算程序开发框架
   4. HIVE：基于大数据技术（文件系统+运算框架）的SQL数据仓库工具
   5. HBASE：基于HADOOP的分布式海量数据库
   6. ZOOKEEPER：分布式协调服务基础组件
   7. Mahout：基于mapreduce/spark/flink等分布式运算框架的机器学习算法库
   8. Oozie：工作流调度框架azkaban
   9. Sqoop：数据导入导出工具
   10. Flume：日志数据采集框架
4. **数据仓库与hive的关系**
   1. Hive 是基于 Hadoop 的一个数据仓库工具，
   2. 可以将结构化的数据文件映射为 一张数据库表，
   3. 并提供类 SQL 查询功能。本质是将 SQL 转换为 MapReduce 程序。
5. **多表join如何处理?**
   1. inner join / left join /right join /full outer join
6. **手写mr 的join mr中小表join大表 怎么优化**
   1. mapreduce常见的join：reduce端的join、map端的join、semi join
   2. reduce 端的join：
   3. 核心思想：
   4. 在map端将来源不同的数据或者有不同用处的数据打标记输出，
   5. 以便reduce端能够识别并进行连接关系查找。
   6. 适用场景：所有的表都是大表时（几乎所有的业务都满足）
   7. 优点：解决业务范围较广
   8. 缺点：从map端传输到reduce端的数据量较大，且有很多的无效的数据，
   9. 大大的增加了传输时间，大大的增加了shuffle过程的耗时，和连接关系的查询时间
   10. map端的join：
   11. 核心思想：
   12. 将小表进行分布式缓存，然后在map端取出缓存的数据来进行连接查询
   13. 适用场景：大表和小表同时存在（至少有一个小表存在）
   14. 优点：从缓存中读取数据，然后在map端进行关联查找，从而减少map到reduce的数据传输
   15. 缺点：只适合有小表的业务需求
7. **hive中有没有遇到数据倾斜？是怎么解决的？**
   1. 由于key分布不均衡造成的数据往一个方向偏移的现象。
   2. 本身的数据倾斜。
   3. join语句容易造成
   4. count（distinct col）
   5. group by
   6. 解决：
   7. set hive.map.aggr=true
   8. set hive.groupby.skewindata=false/true(建议开启)
   9. set hive.optimize.skewjoin=false/true
8. **MapReduce的Shuffle过程**
   1. 画图，画图，画图
   2. 1.maptask执行，outputcollect收集maptask的输出数据，将数据写入环形缓冲区中，记录起始偏移量
   3. 2.环形缓冲区默认大小为100M，当数据达到80M时，记录终止偏移量。
   4. 3.启动spiller溢出器，将数据进行分区（默认分组根据key的hash值%reduce数量进行分区），分区内进行快速排序
   5. 4.分区，排序结束后，将数据刷写到磁盘（这个过程中，maptask输出的数据写入剩余20%环形缓冲区，同样需要记录起始偏移量）
   6. 5.maptask将形成的多个小文件做归并排序合并成一个大文件
   7. 6.当有一个maptask执行完成后，appMaster申请资源，启动reducetask
   8. 7.reducetask到运行完成maptask的机器上拉取属于自己分区的数据
   9. 8.reducetask将拉取过来的数据进行merge操作，归并排序数据，将数据按相同key“分组”，每组数据调用一次reduce（）方法
   10. 9.执行reduce逻辑，将结果输出到文件
9. **Shffule过程中排序用的什么算法**
   1. 1、溢出过程的分区排序采用快排算法
   2. 2、merge过程中采用归并排序算法
10. **如果Reduce个数和分区数不一致时，会发生什么**
    1. 1、如果reduce个数大于分区数，多的reduce将会生成空的输出文件
    2. 2、如果reduce个数小于分区数，将会报错，数据找不到分区。
    3. 可以自定义Partitioner类，在getPartition中采用分区数%reduce个数进行处理。
11. **使用过Hive解析JSON串么**
    1. 采用jsonSerde解析数据，也可以自动以UDF处理
12. **hbase与mysql的区别**
    1. ①定义：
    2. a)MySQL：关系型数据库，主要面向OLTP，支持事务，支持二级索引，支持sql，支持主从、Group Replication架构模型（此处以Innodb为例，不涉及别的存储引擎）。
    3. b)HBase：基于HDFS，支持海量数据读写（尤其是写），支持上亿行、上百万列的，面向列的分布式NoSql数据库。天然分布式，主从架构，不支持事务，不支持二级索引，不支持sql。
    4. ②数据存储方式
    5. a)MySQL采用行存储
    6. MySQL行存储的方式比较适合OLTP业务。
    7. b)HBase是面向列的NoSql数据库
    8. 列存储的方式比较适合OLAP业务，而HBase采用了列族的方式平衡了OLTP和OLAP，支持水平扩展，如果数据量比较大、对性能要求没有那么高、并且对事务没有要求的话，HBase也是个不错的选择。
    9. ③适用场景：
    10. 比较点 MySQL HBase
    11. \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_
    12. 存储方式 行存储，适用难于OLTP业务 列式存储，平衡了OLTP、OLAP业务
    13. 扩展性 单机、扩展性差 水平扩展
    14. 事务 支持 不支持
    15. 一致性 强一致性 强一致性、时间线一致性
    16. 二级索引 支持 不一致性
    17. 全文索引 支持 不支持
13. **集群规模？集群配置？**
    1. 1）大数据搭建集群规模，需要在正式搭建集群之前，需要准备好，即是计划集群的规模情况，否则后面容易出麻烦。
    2. 如：
    3. 服务器为一台128G内存华为服务器
    4. 硬盘raid0之后为14T。
    5. 安装Esxi6.0系统。
    6. 服务器一台，内存128G，10C，安装windows操作系统
    7. 服务器规划：
    8. 服务器安装ESXI5.5,虚拟出4台centos6.5服务器，3台内存32G，2C，用于搭建大数据环境；另1台内存16G，2C，安装Oracle，mysql，Tomcat等软件作为应用服务器。
    9. 2）具体举例：
    10. ①整体规模分配
    11. 2台（Namenode HA，Resourcemanager HA，Hmaster HA）
    12. 10台（Datanode，nodemanager，journalnode，Regionserver）
    13. 1台（hive，sqoop，oozie，hue）
    14. 3台（zookeeper）
    15. ②内存以及cpu分配
    16. cpu：
    17. 32-64核
    18. 内存：
    19. 64-128G
    20. ③具体分配
    21. Namenode：16G
    22. Resourcemanager：2G
    23. Hmaster：2G
    24. ----------------------------------------------20G
    25. Datanode：1G
    26. nodemanager：1G
    27. journalnode:1G
    28. Regionserver:16G
    29. ----------------------------------------------20G
    30. hive，sqoop，oozie，hue:1G
    31. -----------------------------------------------1G
    32. ZK:2G
    33. 要求磁盘读写快
    34. -----------------------------------------------2G
    35. yarn：
    36. yarn.nodemanager.resource.memory-mb：8192
    37. 给定的是每个NM节点，任务运行允许分配的内存大小，工作中一般都会改大, 16\*1024 或 24\*1024
    38. yarn.nodemanager.resource.cpu-vcores：8
    39. 给定的是每个NM节点，任务运行允许分配的CPU大小，必须和memory-mb转换为GB一样的值一样，工作中一般都会改大, 16 或 24
    40. ④网络配置
    41. 万兆
    42. 千兆：这时，瓶颈出现在网络上
    43. ⑤数据量规划
    44. 1~3千万的数据量，大约400W
    45. 需要磁盘50G。
    46. ⑥job数据规划
    47. 总数量在40个左右，mr程序30个左右，hive不到10个
14. **mr的流程**
    1. 1）map阶段，在这个阶段主要分如下的几个步骤read，map,collect，溢写，combine阶段

① 在read阶段，maptask会调用用户自定义的RecordReader方法，在splitInput中解析出一个个的key-value对

② 在map阶段，maptask会接受由前面读取来的数据，然后按照所需的逻辑对数据进行加工处理，形成新的key-value对

③在collect阶段，map在数据处理完成之后会调用OutputCollector.collect()方法把数据写入环形缓冲区中，这个环形缓冲区被分为了两部分，一半是用来存储数据的索引，一半是用来存储数据，在分区中首先是按照分区号进行排序，在分区里面在按照key进行排序，在环形缓冲区中默认的采用的是哈希分区，如果想自定义分区可以重写一个类继承partition在重新分区方法即可。

④ 溢写阶段，当环形缓冲区中的数据到达整个缓冲区的百分之八十的时候(环形缓冲区默认大小是100M),就会把数据写入本地的临时文件，但是为了提高性能在这儿可以调用combiner首先把数据合并之后再把数据写入零时文件，在环形缓冲区上的数据读写方法时索引存储数据和存储索引占到总大小的百分之八十多的时候，双方在结束的位置同时向开始的位置读取数据，这样循环往复。

⑤ 合并阶段,在前面产生的所有的临时文件，maptask采用轮转的方式进行合并，并且在合并之后的分区中进行排序，这样这个map就会产生一个数据的输出文件。

* 1. 2）reduce阶段主要包括如下的阶段，cpoy阶段，merge阶段，sort阶段，reduce阶段

① copy阶段主要是reducer阶段从远程的map的输出去拷贝数据到本地的内存中，同一个reducer的节点会把不同的map的输出的同一个分区拷贝到本地。

② merge阶段，在开始拷贝数据的时候。reduceTask会启动两个后台线程，合并内存中的数据和磁盘中的数据，防止使用过多的内存和磁盘。

③ sort阶段，由于在reduce的数据是按照key进行聚合排序的，但是在map的输出的时候就已经进行排序，所以在这儿只需要简单的归并排序即可

④ reduce阶段，把已经按照key聚合的数据输出给reducer按照我们的义务逻辑进行处理。

1. **hive开窗函数写过没有？怎么写的？除了row\_number还知道哪些窗口函数？（就那四个排序窗口函数）他们有啥区别？**
   1. Hive中提供了很多分析函数，用于完成负责的统计分析。先看看基础的sum,avg,min,max，用于实现分组内所有和连续累计的统计。
   2. sum(columns) over (partition by col1 order by col2 rows between n/unbounded preceding and m following/current row )
   3. 如果不指定rows between，默认为从起点到当前行；
   4. 如果不指定order by，则将分组内所有值累加；
   5. 关键是理解rows between含义，也叫做window子句：
   6. preceding:往前
   7. following：往后
   8. current rows：当前行
   9. unbounded：起点
   10. unbounded preceding：表示从前面的起点
   11. unbounded following：表示到后面的终点
   12. ntile(n),用于将分组数据按照顺序切分成N片，返回当前切片值。ntile不支持rows between，如果切片不均匀，默认增加第一个切片的分布。
   13. row\_number(),从1开始，按照顺序，生成分组内记录的序列。
   14. rank(),生成数据项在分组中的排名，排名相等会在名次中留下空位。
   15. dense\_rank(),生成数据项在分组中的排名，排名相等会在名词中不会留下空位。
   16. cume\_dist(),小于等于当前值的行数/分组内总行数。
   17. percent\_rank(),分组内当前行的rank值-1/分组内总行数-1
2. **yarn 的参数 mr参数 hive设置过什么参数**
   1. Yarn的参数：
   2. 1）ResourceManager相关配置参数
   3. ① yarn.resourcemanager.address
   4. 参数解释：ResourceManager对客户端暴露的地址。客户端通过该地址向RM提交应用程序，杀死应用程序等。
   5. 默认值：${yarn.resourcemanager.hostname}:8032
   6. ② yarn.resourcemanager.scheduler.address
   7. 参数解释：ResourceManager对ApplicationMaster暴露的访问地址。ApplicationMaster通过该地址向RM申请资源、释放资源等。
   8. 默认值：${yarn.resourcemanager.hostname}:8030
   9. ③ yarn.resourcemanager.resource-tracker.address
   10. 参数解释：ResourceManager对NodeManager暴露的地址.。NodeManager通过该地址向RM汇报心跳，领取任务等。
   11. 默认值：${yarn.resourcemanager.hostname}:8031
   12. ④ yarn.resourcemanager.admin.address
   13. 参数解释：ResourceManager对管理员暴露的访问地址。管理员通过该地址向RM发送管理命令等。
   14. 默认值：${yarn.resourcemanager.hostname}:8033
   15. ⑤ yarn.resourcemanager.webapp.address
   16. 参数解释：ResourceManager对外webui地址。用户可通过该地址在浏览器中查看集群各类信息。
   17. 默认值：${yarn.resourcemanager.hostname}:8088
   18. ⑥ yarn.resourcemanager.scheduler.class
   19. 参数解释：启用的资源调度器主类。目前可用的有FIFO、CapacityScheduler和FairScheduler。
   20. 默认值：
   21. org.apache.hadoop.yarn.server.resourcemanager.scheduler.capacity.CapacityScheduler
   22. ⑦yarn.resourcemanager.resource-tracker.client.thread-count
   23. 参数解释：处理来自NodeManager的RPC请求的Handler数目。
   24. 默认值：50
   25. ⑧ yarn.resourcemanager.scheduler.client.thread-count
   26. 参数解释：处理来自ApplicationMaster的RPC请求的Handler数目。
   27. 默认值：50
   28. ⑨ yarn.scheduler.minimum-allocation-mb/yarn.scheduler.maximum-allocation-mb
   29. 参数解释：单个可申请的最小/最大内存资源量。比如设置为1024和3072，则运行MapRedce作业时，每个Task最少可申请1024MB内存，最多可申请3072MB内存。
   30. 默认值：1024/8192
   31. ⑩ yarn.scheduler.minimum-allocation-vcores/yarn.scheduler.maximum-allocation-vcores
   32. 参数解释：单个可申请的最小/最大虚拟CPU个数。比如设置为1和4，则运行MapRedce作业时，每个Task最少可申请1个虚拟CPU，最多可申请4个虚拟CPU。什么是虚拟CPU，可阅读我的这篇文章：“YARN资源调度器剖析”。
   33. 默认值：1/32
   34. ⑪ yarn.resourcemanager.nodes.include-path/yarn.resourcemanager.nodes.exclude-path
   35. 参数解释：NodeManager黑白名单。如果发现若干个NodeManager存在问题，比如故障率很高，任务运行失败率高，则可以将之加入黑名单中。注意，这两个配置参数可以动态生效。（调用一个refresh命令即可）
   36. 默认值：“”
   37. ⑫ yarn.resourcemanager.nodemanagers.heartbeat-interval-ms
   38. 参数解释：NodeManager心跳间隔
   39. 默认值：1000（毫秒）
   40. 2）NodeManager相关配置参数
   41. ① yarn.nodemanager.resource.memory-mb
   42. 参数解释：NodeManager总的可用物理内存。注意，该参数是不可修改的，一旦设置，整个运行过程中不可动态修改。另外，该参数的默认值是8192MB，即使你的机器内存不够8192MB，YARN也会按照这些内存来使用（傻不傻？），因此，这个值通过一定要配置。不过，Apache已经正在尝试将该参数做成可动态修改的。
   43. 默认值：8192
   44. ② yarn.nodemanager.vmem-pmem-ratio
   45. 参数解释：每使用1MB物理内存，最多可用的虚拟内存数。
   46. 默认值：2.1
   47. ③ yarn.nodemanager.resource.cpu-vcores
   48. 参数解释：NodeManager总的可用虚拟CPU个数。
   49. 默认值：8
   50. ④ yarn.nodemanager.local-dirs
   51. 参数解释：中间结果存放位置，类似于1.0中的mapred.local.dir。注意，这个参数通常会配置多个目录，已分摊磁盘IO负载。
   52. 默认值：${hadoop.tmp.dir}/nm-local-dir
   53. ⑤ yarn.nodemanager.log-dirs
   54. 参数解释：日志存放地址（可配置多个目录）。
   55. 默认值：${yarn.log.dir}/userlogs
   56. ⑥ yarn.nodemanager.log.retain-seconds
   57. 参数解释：NodeManager上日志最多存放时间（不启用日志聚集功能时有效）。
   58. 默认值：10800（3小时）
   59. ⑦ yarn.nodemanager.aux-services
   60. 参数解释：NodeManager上运行的附属服务。需配置成mapreduce\_shuffle，才可运行MapReduce程序
   61. 默认值：“”
   62. mr参数：
   63. MapReduce重要配置参数
   64. 1） 资源相关参数
   65. //以下参数是在用户自己的mr应用程序中配置就可以生效
   66. ①mapreduce.map.memory.mb: 一个MapTask可使用的资源上限（单位:MB），默认为1024。如果MapTask实际使用的资源量超过该值，则会被强制杀死。
   67. ②mapreduce.reduce.memory.mb: 一个ReduceTask可使用的资源上限（单位:MB），默认为1024。如果ReduceTask实际使用的资源量超过该值，则会被强制杀死。
   68. ③mapreduce.map.java.opts: Map Task的JVM参数，你可以在此配置默认的javaheap size等参数,e.g.
   69. “-Xmx1024m-verbose:gc -Xloggc:/tmp/@taskid@.gc”（@taskid@会被Hadoop框架自动换为相应的taskid）,默认值: “”
   70. ④mapreduce.reduce.java.opts: Reduce Task的JVM参数，你可以在此配置默认的javaheap size等参数,e.g.
   71. “-Xmx1024m-verbose:gc -Xloggc:/tmp/@taskid@.gc”, 默认值:“”
   72. ⑤mapreduce.map.cpu.vcores: 每个Maptask可使用的最多cpucore数目, 默认值:1
   73. ⑥mapreduce.reduce.cpu.vcores: 每个Reducetask可使用的最多cpucore数目, 默认值:1
   74. //应该在yarn启动之前就配置在服务器的配置文件中才能生效
   75. ⑦yarn.scheduler.minimum-allocation-mb 1024 给应用程序container分配的最小内存
   76. ⑧yarn.scheduler.maximum-allocation-mb 8192 给应用程序container分配的最大内存
   77. ⑨yarn.scheduler.minimum-allocation-vcores 1
   78. ⑩yarn.scheduler.maximum-allocation-vcores 32
   79. ⑪yarn.nodemanager.resource.memory-mb 8192
   80. //shuffle性能优化的关键参数，应在yarn启动之前就配置好
   81. ⑫mapreduce.task.io.sort.mb 100 //shuffle的环形缓冲区大小，默认100m
   82. ⑬mapreduce.map.sort.spill.percent 0.8 //环形缓冲区溢出的阈值，默认80%
   83. 2） 容错相关参数
   84. ①mapreduce.map.maxattempts: 每个MapTask最大重试次数，一旦重试参数超过该值，则认为MapTask运行失败，默认值：4。
   85. ②mapreduce.reduce.maxattempts: 每个ReduceTask最大重试次数，一旦重试参数超过该值，则认为MapTask运行失败，默认值：4。
   86. ③mapreduce.map.failures.maxpercent: 当失败的MapTask失败比例超过该值为，整个作业则失败，默认值为0.如果你的应用程序允许丢弃部分输入数据，则该该值设为一个大于0的值，比如5，表示如果有低于5%的MapTask失败（如果一个MapTask重试次数超过mapreduce.map.maxattempts，则认为这个MapTask失败，其对应的输入数据将不会产生任何结果），整个作业扔认为成功。
   87. ④mapreduce.reduce.failures.maxpercent: 当失败的ReduceTask失败比例超过该值为，整个作业则失败，默认值为0.
   88. ⑤mapreduce.task.timeout:Task超时时间，经常需要设置的一个参数，该参数表达的意思为：如果一个task在一定时间内没有任何进入，即不会读取新的数据，也没有输出数据，则认为该task处于block状态，可能是卡住了，也许永远会卡主，为了防止因为用户程序永远block住不退出，则强制设置了一个该超时时间（单位毫秒），默认是300000。如果你的程序对每条输入数据的处理时间过长（比如会访问数据库，通过网络拉取数据等），建议将该参数调大，该参数过小常出现的错误提示是“AttemptID:attempt\_14267829456721\_123456\_m\_000224\_0Timed out after 300 secsContainer killed by the ApplicationMaster.”。
   89. 3） 本地运行mapreduce作业
   90. 设置以下几个参数:
   91. mapreduce.framework.name=local
   92. mapreduce.jobtracker.address=local
   93. fs.defaultFS=local
   94. 4） 效率和稳定性相关参数
   95. ①mapreduce.map.speculative: 是否为MapTask打开推测执行机制，默认为false
   96. ②mapreduce.reduce.speculative: 是否为ReduceTask打开推测执行机制，默认为false
   97. ③mapreduce.job.user.classpath.first &mapreduce.task.classpath.user.precedence：当同一个class同时出现在用户jar包和hadoopjar中时，优先使用哪个jar包中的class，默认为false，表示优先使用hadoopjar中的class。
   98. ④mapreduce.input.fileinputformat.split.minsize:FileInputFormat做切片时的最小切片大小，(5)mapreduce.input.fileinputformat.split.maxsize: FileInputFormat做切片时的最大切片大小
   99. (切片的默认大小就等于blocksize，即134217728)
   100. Hive参数：
   101. 1)在使用union all的时候，系统资源足够的情况下，为了加快hive处理速度，可以设置如下参数实现并发执行
   102. set mapred.job.priority=VERY\_HIGH;
   103. set hive.exec.parallel=true;
   104. 2)设置map reduce个数
   105. -- 设置map capacity
   106. set mapred.job.map.capacity=2000;
   107. set mapred.job.reduce.capacity=2000;
   108. -- 设置每个reduce的大小
   109. set hive.exec.reducers.bytes.per.reducer=500000000;
   110. -- 直接设置个数
   111. set mapred.reduce.tasks = 15;
   112. 3)设置任务名称
   113. -- 设置名称
   114. set mapred.job.name=my\_job\_{DATE};
   115. 4）Hive文件合并
   116. -- 设置文件合并
   117. set abaci.is.dag.job=false;
   118. set hive.merge.mapredfiles=true;
   119. set mapred.combine.input.format.local.only=false;
   120. set hive.merge.smallfiles.avgsize=100000000;
   121. -- 在map only的情况下，如上的参数如果没有生效，可以设置如下
   122. -- 在HQL的最外层增加distribute by rand()
   123. select \* from XXX distribute by rand()
   124. use namespace udw\_ns;
   125. set mapred.job.name=job\_name\_{DATE};
   126. set hive.mapred.mode=nonstrict;
   127. set mapred.reduce.tasks = 600;
   128. set hive.exec.dynamic.partition.mode=nonstrict;
   129. set hive.exec.dynamic.partition=true;
   130. set hive.exec.compress.output=true;
   131. set mapred.output.compress=true;
   132. set mapred.output.compression.codec=org.apache.hadoop.io.compress.LzoCodec;
   133. 5）dfs.block.size
   134. 决定HDFS文件block数量的多少(文件个数),它会间接的影响Job Tracker的调度和内存的占用(更影响内存的使用),
   135. 6）mapred.map.tasks.speculative.execution=true
   136. mapred.reduce.tasks.speculative.execution=true
   137. 这是两个推测式执行的配置项,默认是true
   138. 所谓的推测执行，就是当所有task都开始运行之后，Job Tracker会统计所有任务的平均进度，如果某个task所在的task node机器配
   139. 置比较低或者CPU load很高（原因很多），导致任务执行比总体任务的平均执行要慢，此时Job Tracker会启动一个新的任务
   140. （duplicate task），原有任务和新任务哪个先执行完就把另外一个kill掉，这也是我们经常在Job Tracker页面看到任务执行成功，但
   141. 是总有些任务被kill，就是这个原因。
   142. 7）mapred.child.java.opts
   143. 一般来说，都是reduce耗费内存比较大，这个选项是用来设置JVM堆的最大可用内存，但不要设置过大，如果超过2G(这是数字有
   144. 待考证)，就应该考虑一下优化程序。
   145. Input Split的大小，决定了一个Job拥有多少个map，默认64M每个Split，如果输入的数据量巨大，那么默认的64M的block会有特
   146. 别多Map Task，集群的网络传输会很大，给Job Tracker的调度、队列、内存都会带来很大压力。
   147. 8）mapred.min.split.size
   148. 这个配置决定了每个Input Split 的最小值,也间接决定了一个job的map数量
   149. HDFS块大小是在job写入时决定的,而分片的大小,是由三个元素决定的(在3各种去最大的那个)
   150. ① 输入的块数
   151. ② Mapred.min.split.size
   152. ③ Job.setNumMapTasks()
   153. 9）mapred.compress.map.output
   154. 压缩Map的输出，这样做有两个好处：
   155. a)压缩是在内存中进行，所以写入map本地磁盘的数据就会变小，大大减少了本地IO次数
   156. b) Reduce从每个map节点copy数据，也会明显降低网络传输的时间
   157. 注：数据序列化其实效果会更好，无论是磁盘IO还是数据大小，都会明显的降低。
   158. 10）io.sort.mb
   159. 以MB为单位，默认100M，这个值比较小
   160. map节点没运行完时，内存的数据过多，要将内存中的内容写入洗盘，这个设置就是设置内存缓冲的大小，在suffle之前
   161. 这个选项定义了map输出结果在内存里占用buffer的大小，当buffer达到某个阈值(后面那条配置)，会启动一个后台线程来对buffer
   162. 的内容进行排序，然后写入本地磁盘(一个spill文件)
   163. 根据map输出数据量的大小，可以适当的调整buffer的大小，注意是适当的调整，并不是越大越好，假设内存无限大，
   164. io.sort.mb=1024(1G), 和io.sort.mb=300 (300M)，前者未必比后者快：
   165. （1）1G的数据排序一次
   166. （2）排序3次，每次300MB
   167. 一定是后者快（归并排序）
   168. 11）io.sort.spill.percent
   169. 这个值就是上面提到的buffer的阈值，默认是0.8，既80%，当buffer中的数据达到这个阈值，后台线程会起来对buffer中已有的数
   170. 据进行排序，然后写入磁盘，此时map输出的数据继续往剩余的20% buffer写数据，如果buffer的剩余20%写满，排序还没结束，
   171. map task被block等待。
   172. 如果你确认map输出的数据基本有序，排序时间很短，可以将这个阈值适当调高，更理想的，如果你的map输出是有序的数据，那
   173. 么可以把buffer设的更大，阈值设置为1.
   174. 12）Io.sort.factor
   175. 同时打开的文件句柄的数量，默认是10
   176. 当一个map task执行完之后，本地磁盘上(mapred.local.dir)有若干个spill文件，map task最后做的一件事就是执行merge sort，
   177. 把这些spill文件合成一个文件（partition，combine阶段）。
   178. 执行merge sort的时候，每次同时打开多少个spill文件，就是由io.sort.factor决定的。打开的文件越多，不一定merge sort就越
   179. 快，也要根据数据情况适当的调整。
   180. 注：merge排序的结果是两个文件，一个是index，另一个是数据文件，index文件记录了每个不同的key在数据文件中的偏移量（即partition）。
   181. 在map节点上，如果发现map所在的子节点的机器io比较重，原因可能是io.sort.factor这个设置的比较小，io.sort.factor设置小的
   182. 话，如果spill文件比较多，merge成一个文件要很多轮读取操作，这样就提升了io的负载。io.sort.mb小了，也会增加io的负载。
   183. 如果设置了执行combine的话，combine只是在merge的时候，增加了一步操作，不会改变merge的流程，所以combine不会减少
   184. 或者增加文件个数。另外有个min.num.spills.for.combine的参数，表示执行一个merge操作时，如果输入文件数小于这个数字，就
   185. 不调用combiner。如果设置了combiner，在写spill文件的时候也会调用，这样加上merge时候的调用，就会执行两次combine。
   186. 提高Reduce的执行效率，除了在Hadoop框架方面的优化，重点还是在代码逻辑上的优化.比如：对Reduce接受到的value可能有重
   187. 复的，此时如果用Java的Set或者STL的Set来达到去重的目的，那么这个程序不是扩展良好的(non-scalable)，受到数据量的限制，
   188. 当数据膨胀，内存势必会溢出
   189. 13）mapred.reduce.parallel.copies
   190. Reduce copy数据的线程数量，默认值是5
   191. Reduce到每个完成的Map Task 拷贝数据（通过RPC调用），默认同时启动5个线程到map节点取数据。这个配置还是很关键的，
   192. 如果你的map输出数据很大，有时候会发现map早就100%了，reduce却在缓慢的变化，那就是copy数据太慢了，比如5个线程
   193. copy 10G的数据，确实会很慢，这时就要调整这个参数，但是调整的太大，容易造成集群拥堵，所以 Job tuning的同时，也是个权
   194. 衡的过程，要熟悉所用的数据！
   195. mapred.job.shuffle.input.buffer.percent
   196. 当指定了JVM的堆内存最大值以后，上面这个配置项就是Reduce用来存放从Map节点取过来的数据所用的内存占堆内存的比例，默
   197. 认是0.7，既70%，通常这个比例是够了，但是对于大数据的情况，这个比例还是小了一些，0.8-0.9之间比较合适。（前提是你的
   198. reduce函数不会疯狂的吃掉内存）
   199. 14）mapred.job.shuffle.merge.percent(默认值0.66)
   200. mapred.inmem.merge.threshold(默认值1000)
   201. 第一个指的是从Map节点取数据过来，放到内存，当达到这个阈值之后，后台启动线程（通常是Linux native process）把内存中的
   202. 数据merge sort，写到reduce节点的本地磁盘；
   203. 第二个指的是从map节点取过来的文件个数，当达到这个个数之后，也进行merger sort，然后写到reduce节点的本地磁盘；这两
   204. 个配置项第一个优先判断，其次才判断第二个thresh-hold。
   205. 从实际经验来看，mapred.job.shuffle.merge.percent默认值偏小，完全可以设置到0.8左右；第二个默认值1000，完全取决于
   206. map输出数据的大小，如果map输出的数据很大，默认值1000反倒不好，应该小一些，如果map输出的数据不大（light
   207. weight），可以设置2000或者以上。
   208. 15）mapred.reduce.slowstart.completed.maps （map完成多少百分比时，开始shuffle）
   209. 当map运行慢，reduce运行很快时，如果不设置mapred.reduce.slowstart.completed.maps会使job的shuffle时间变的很长，
   210. map运行完很早就开始了reduce，导致reduce的slot一直处于被占用状态。mapred.reduce.slowstart.completed.maps 这个值是
   211. 和“运行完的map数除以总map数”做判断的，当后者大于等于设定的值时，开始reduce的shuffle。所以当map比reduce的执行
   212. 时间多很多时，可以调整这个值（0.75,0.80,0.85及以上）
   213. 下面从流程里描述一下各个参数的作用：
   214. 当map task开始运算，并产生中间数据时，其产生的中间结果并非直接就简单的写入磁盘。这中间的过程比较复杂，并且利用到了
   215. 内存buffer来进行已经产生的部分结果的缓存，并在内存buffer中进行一些预排序来优化整个map的性能。每一个map都会对应存
   216. 在一个内存buffer（MapOutputBuffer），map会将已经产生的部分结果先写入到该buffer中，这个buffer默认是100MB大小，但
   217. 是这个大小是可以根据job提交时的参数设定来调整的，该参数即为：io.sort.mb。当map的产生数据非常大时，并且把io.sort.mb
   218. 调大，那么map在整个计算过程中spill的次数就势必会降低，map task对磁盘的操作就会变少，如果map tasks的瓶颈在磁盘上，
   219. 这样调整就会大大提高map的计算性能。
   220. map在运行过程中，不停的向该buffer中写入已有的计算结果，但是该buffer并不一定能将全部的map输出缓存下来，当map输出
   221. 超出一定阈值（比如100M），那么map就必须将该buffer中的数据写入到磁盘中去，这个过程在mapreduce中叫做spill。map并
   222. 不是要等到将该buffer全部写满时才进行spill，因为如果全部写满了再去写spill，势必会造成map的计算部分等待buffer释放空间的
   223. 情况。所以，map其实是当buffer被写满到一定程度（比如80%）时，就开始进行spill。这个阈值也是由一个job的配置参数来控
   224. 制，即io.sort.spill.percent，默认为0.80或80%。这个参数同样也是影响spill频繁程度，进而影响map task运行周期对磁盘的读写
   225. 频率的。但非特殊情况下，通常不需要人为的调整。调整io.sort.mb对用户来说更加方便。
   226. 当map task的计算部分全部完成后，如果map有输出，就会生成一个或者多个spill文件，这些文件就是map的输出结果。map在正
   227. 常退出之前，需要将这些spill合并（merge）成一个，所以map在结束之前还有一个merge的过程。merge的过程中，有一个参数
   228. 可以调整这个过程的行为，该参数为：io.sort.factor。该参数默认为10。它表示当merge spill文件时，最多能有多少并行的stream
   229. 向merge文件中写入。比如如果map产生的数据非常的大，产生的spill文件大于10，而io.sort.factor使用的是默认的10，那么当
   230. map计算完成做merge时，就没有办法一次将所有的spill文件merge成一个，而是会分多次，每次最多10个stream。这也就是说，
   231. 当map的中间结果非常大，调大io.sort.factor，有利于减少merge次数，进而减少map对磁盘的读写频率，有可能达到优化作业的
   232. 目的。
   233. 当job指定了combiner的时候，我们都知道map介绍后会在map端根据combiner定义的函数将map结果进行合并。运行combiner
   234. 函数的时机有可能会是merge完成之前，或者之后，这个时机可以由一个参数控制，即min.num.spill.for.combine（default 3），
   235. 当job中设定了combiner，并且spill数最少有3个的时候，那么combiner函数就会在merge产生结果文件之前运行。通过这样的方
   236. 式，就可以在spill非常多需要merge，并且很多数据需要做conbine的时候，减少写入到磁盘文件的数据数量，同样是为了减少对磁
   237. 盘的读写频率，有可能达到优化作业的目的。
   238. 减少中间结果读写进出磁盘的方法不止这些，还有就是压缩。也就是说map的中间，无论是spill的时候，还是最后merge产生的结
   239. 果文件，都是可以压缩的。压缩的好处在于，通过压缩减少写入读出磁盘的数据量。对中间结果非常大，磁盘速度成为map执行瓶
   240. 颈的job，尤其有用。控制map中间结果是否使用压缩的参数为：mapred.compress.map.output(true/false)。将这个参数设置为
   241. true时，那么map在写中间结果时，就会将数据压缩后再写入磁盘，读结果时也会采用先解压后读取数据。这样做的后果就是：写
   242. 入磁盘的中间结果数据量会变少，但是cpu会消耗一些用来压缩和解压。所以这种方式通常适合job中间结果非常大，瓶颈不在
   243. cpu，而是在磁盘的读写的情况。说的直白一些就是用cpu换IO。根据观察，通常大部分的作业cpu都不是瓶颈，除非运算逻辑异常
   244. 复杂。所以对中间结果采用压缩通常来说是有收益的。
   245. 当采用map中间结果压缩的情况下，用户还可以选择压缩时采用哪种压缩格式进行压缩，现在hadoop支持的压缩格式有：
   246. GzipCodec，LzoCodec，BZip2Codec，LzmaCodec等压缩格式。通常来说，想要达到比较平衡的cpu和磁盘压缩比，LzoCodec
   247. 比较适合。但也要取决于job的具体情况。用户若想要自行选择中间结果的压缩算法，可以设置配置参数：
   248. mapred.map.output.compression.codec=org.apache.hadoop.io.compress.DefaultCodec或者其他用户自行选择的压缩方式。
3. **hbase里的热点问题遇到过么？怎么解决的？**
   1. 一、出现热点问题原因
   2. 1、hbase的中的数据是按照字典序排序的，当大量连续的rowkey集中写在个别的region，各个region之间数据分布不均衡；
   3. 2、创建表时没有提前预分区，创建的表默认只有一个region，大量的数据写入当前region；
   4. 3、创建表已经提前预分区，但是设计的rowkey没有规律可循，设计的rowkey应该由regionNo+messageId组成。
   5. 二、如何解决热点问题?
   6. 设计可以让数据分布均匀的rowkey，与nosql数据库们一样，rowkey是用来检索记录的主键。访问hbase table中的行，rowkey?可以是任意字符串(最大长度 是 64KB，实际应用中长度一般为 10-100bytes)，在hbase内部，rowkey保存为字节数组，存储时，数据按照rowkey的字典序排序存储。
   7. 1、第一种设计rowkey方式：随机数+messageId，如果想让最近的数据快速get到，可以将时间戳加上这种设计的rowkey可以解决热点问题，但是要建立关联表，比如将rowkey保存到数据库或者nosql数据库中，因为前面的regionNo是随机的，不知道对应数据在hbase的rowkey是多少；同一批数据，因为这个regionNo是随机的，所以要到多个region中get数据，不能使用startkey和endkey去get数据。
   8. 2、第二种设计rowkey的方式：通过messageId映射regionNo，这样既可以让数据均匀分布到各个region中，同时可以根据startkey和endkey可以get到同一批数据messageId映射regionNo，使用一致性hash算法解决
4. **hadoop运行原理**
   1. 包括HDFS和Mapreduce两部分。
   2. 1）HDFS自动保存多个副本，移动计算。缺点是小文件存取占用namenode内存，写入只支持追加，不能随机修改。
   3. 它存储的逻辑空间称为block，文件的权限类似linux。整体架构分三种节点，NN,SNN,DN
   4. NN 负责读写操作保存metadata(Ownership Permission blockinfo)
   5. SNN 负责辅助NN合并fsimage和edits，减少nn启动时间
   6. DN 负责存数据，每个数据（文件）分割成若干block，每个block默认3个副本。启动后像NN发送心跳保持联系
   7. NN保存的metadata在hdfs启动后加载到计算机内存，除block位置信息的metadata保存在OS文件系统中的fsimage文件中，对metadata的操作日志保存在OS文件系统中的edits文件中。block位置信息是hdfs启动后由DN上报NN再加载到内存的。
   8. HDFS的安全模式：直到NN完全加载完metadata之前的这段时间。期间不能写入文件，DN检查各个block完整性，并修复。
   9. 2）MapReduce
   10. 离线计算框架，过程分为split map shuffle reduce四个过程
   11. 架构节点有：Jobtracker TaskTracker
   12. Split将文件分割，传输到mapper，mapper接收KV形式的数据，经过处理，再传到shuffle过程。
   13. Shuffle先进行HashPartition或者自定义的partition，会有数据倾斜和reduce的负载均衡问题；再进行排序，默认按字典排序；为减少mapper输出数据，再根据key进行合并，相同key的数据value会被合并；最后分组形成（key,value{}）形式的数据，输出到下一阶段
   14. Reduce输入的数据就变成了，key+迭代器形式的数据，再进行处理
5. **MapReduce原理**
   1. 逻辑上：
   2. 1、split
   3. 2、map
   4. 3、shuffle
   5. 4、reduce
   6. 四个过程
   7. 物理上：
   8. JobTracker节点：JobTracker创建每一个Task（即MapTask和ReduceTask）
   9. 并将它们分发到各个TaskTracker服务中去执行。负责调度Job的每一个子任务task运行于TaskTracker上。
   10. TaskTracker节点：运行在多个节点上的slaver服务。TaskTracker主动与JobTracker通信，接收作业，并负责直接执行每一个任务。TaskTracker都需要运行在HDFS的DataNode上
6. **hdfs存储机制**
   1. 1） client端发送写文件请求，namenode检查文件是否存在，如果已存在，直接返回错误信息，否则，发送给client一些可用namenode节点2） client将文件分块，并行存储到不同节点上datanode上，发送完成后，client同时发送信息给namenode和datanode3） namenode收到的client信息后，发送确信信息给datanode4） datanode同时收到namenode和datanode的确认信息后，提交写操作。
7. **用mr设计一个分组排重计数算法**
   1. 输入文件格式:二级域名,一级频道,二级频道,访问ip地址,访问者id
   2. 需求:按照二级域名,一级频道,二级频道分组,计算pageview数,计算独立ip数和独立访问者id数
8. **hadoop中combiner的作用**
   1. 当map生成的数据过大时，带宽就成了瓶颈，怎样精简压缩传给Reduce的数据，有不影响最终的结果呢。有一种方法就是使用Combiner，Combiner号称本地的Reduce，Reduce最终的输入，是Combiner的输出
9. **HIVE的优化**
   1. 底层是MapReduce，所以又可以说是MapReduce优化。
   2. 1. 小文件都合并成大文件
   3. 2. Reducer数量在代码中介于节点数\*reduceTask的最大数量的0.95倍到1.75倍
   4. 3. 写一个UDF函数，在建表的时候制定好分区
   5. 4. 配置文件中，打开在map端的合并
10. **Hbase优化**
    1. 1. 表设计
    2. 建表时就分区，rowkey设置定长（64字节），CF2到3个
    3. Max Versio，Time to live，Compact&Split
    4. 2. 写表
    5. 多Htable并发写
    6. Htable参数设置，手动flush，降低IO
    7. WriteBuffer
    8. 批量写
    9. 多线程并发写
    10. 3. 读表
    11. 多Htable并发读
    12. Htable参数设置
    13. 批量读
    14. 释放资源
    15. 缓存查询结果
11. **MapReduce优化**
    1. 1.任务调度
    2. I/O 方面：Hadoop 会尽量将 Map 任务分配给 InputSplit 所在的机器，以减少网
    3. 络 I/O 的消耗。
    4. 2.数据预处理与 InputSplit 的大小
    5. 合理地设置 block块大小是很重要的调节方式。除此之外，也可以通过合理地
    6. 设置 Map 任务的数量来调节 Map 任务的数据输入。
    7. 3. Map 和 Reduce 任务的数量
    8. 当 Reduce 任务的数量是任务槽的 1.75 倍时，执行速度快的机器可以获得更多的 Reduce 任务，因此可以使负载更加均衡，以提高任务的处理速度。
    9. 4. Combine 函数
    10. MapReduce 框架运行用户写的 combine 函数用于本地合并，这会大大减少网
    11. 络 I/O 操作的消耗
12. **你自定义的UDAF函数都实现了哪些功能？**
    1. 关于UDAF:
    2. ①UDAF是Hive中用户自定义的聚集函数。如：Hive中内置的UDAF聚集函数包括sum和count。
    3. ②聚合多行数据，然后返回单一值。
    4. ③Hive自带的绝大多数聚合函数，大多数情况下都能满足业务需求，不能满足业务的是极个别的情形。
    6. 使用UDAF函数实现了：
    7. ①类似于要合并相同的field对应的值，如：{"a","b","c","a"}，若结果集中存在，则不用处理，不存在，则加入。该需求，可以使用hive自带的collect\_set内置函数，该函数用于计算去重后的元素的数组。
    8. ②有A、B、C三列，按A列进行聚合，求出C列聚合后的最小值和最大值各自对应的B列值。这个需求用hql和内建函数也可完成，但是比较繁琐，会解析成几个MR进行执行，如果自定义UDAF便可只利用一个MR完成任务。

## 第三部分：Spark

#### ElasticSearch如何避免脑裂?

修改集群中每个节点的配置文件（elasticsearch.yml）参数 discovery.zen.minimum\_master\_nodes，这个参数决定了主节点选择过程中最少需要多少个 master 节点，默认配置是1。

一个基本原则是这里需要设置成 N/2+1，N 是集群中节点的数量。

修改集群中每个节点的配置文件（elasticsearch.yml）参数 discovery.zen.ping.timeout，默认值是3，决定节点之间网络通信的等待时间。

修改集群中每个节点的配置文件（elasticsearch.yml）参数 discovery.zen.ping.unicast.hosts，把集群中可能成为主节点的机器节点都配置到这个参数中。

#### ElasticSearch中:match和term区别?

term是代表完全匹配，也就是精确查询，搜索前不会再对搜索词进行分词，所以搜索词必须是文档分词集合中的一个

match查询会先对搜索词进行分词,分词完毕后再逐个对分词结果进行匹配，因此相比于term的精确搜索，match是分词匹配搜索,match搜索还有两个相似功能的变种，一个是match\_phrase，一个是multi\_match

#### spark开发分两个方面？哪两个方面呢？

* 1. 离线数据分析和实时数据分析

#### spark和Mapreduce快？ 为什么快呢？ 快在哪里呢？

* 1. 内存迭代、RDD设计、算子的设计。

#### Spark Sql为什么比Hive快呢？

* 1. Spark Sql最终内部调用的是Spark RDD，而Hive则是调用的MapReduce

#### RDD的数据结构是怎么样的？

1、一组分片（Partition），即数据集的基本组成单位。对于RDD来说，每个分片都会被一个计算任务处理，并决定并行计算的粒度。用户可以在创建RDD时指定RDD的分片个数，如果没有指定，那么就会采用默认值。默认值就是程序所分配到的CPU Core的数目。

2、一个计算每个分区的函数。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之间就会形成类似于流水线一样的前后依赖关系。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。

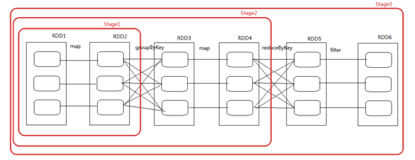
4、一个Partitioner，即RDD的分片函数。当前Spark中实现了两种类型的分片函数，一个是基于哈希的HashPartitioner，另外一个是基于范围的RangePartitioner。只有对于于key-value的RDD，才会有Partitioner，非key-value的RDD的Parititioner的值是None。Partitioner函数不但决定了RDD本身的分片数量，也决定了parent RDD Shuffle输出时的分片数量。

5、一个列表，存储存取每个Partition的优先位置（preferred location）。对于一个HDFS文件来说，这个列表保存的就是每个Partition所在的块的位置。按照“移动数据不如移动计算”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。

#### Spark工作流程

提交任务：用户提交一个任务。 入口是从sc开始的。 sc会去创建一个taskScheduler。根据不同的提交模式， 会根据相应的taskchedulerImpl进行任务调度。 同时会去创建Scheduler和DAGScheduler。DAGScheduler 会根据RDD的宽依赖或者窄依赖，进行阶段的划分。划分好后放入taskset中，交给taskscheduler 。 appclient会到master上注册。首先会去判断数据本地化，尽量选最好的本地化模式去执行。 打散 Executor选择相应的Executor去执行。ExecutorRunner会去创建CoarseGrainerExecutorBackend进程。 通过线程池的方式去执行任务。 反向： Executor向 SchedulerBackend反向注册 Spark On Yarn模式下。 driver负责计算调度。appmaster 负责资源的申请。

#### stage怎么划分的

* 1. 

#### 宽依赖窄依赖是什么？

宽依赖指的是多个子RDD的Partition会依赖同一个父RDD的Partition

窄依赖指的是每一个父RDD的Partition最多被子RDD的一个Partition使用

#### kafkadirect是什么? 为什么要用这个，有什么优点？和其他的有什么区别。

1）Direct的方式是会直接操作kafka底层的元数据信息，这样如果计算失败了，可以把数据重新读一下，重新处理。即数据一定会被处理。拉数据，是RDD在执行的时候直接去拉数据。

2）由于直接操作的是kafka，kafka就相当于你底层的文件系统。这个时候能保证严格的事务一致性，即一定会被处理，而且只会被处理一次。而Receiver的方式则不能保证，因为Receiver和ZK中的数据可能不同步，Spark Streaming可能会重复消费数据，这个调优可以解决，但显然没有Direct方便。而Direct api直接是操作kafka的，spark streaming自己负责追踪消费这个数据的偏移量或者offset，并且自己保存到checkpoint，所以它的数据一定是同步的，一定不会被重复。即使重启也不会重复，因为checkpoint了，但是程序升级的时候，不能读取原先的checkpoint，面对升级checkpoint无效这个问题，怎么解决呢?升级的时候读取我指定的备份就可以了，即手动的指定checkpoint也是可以的，这就再次完美的确保了事务性，有且仅有一次的事务机制。那么怎么手动checkpoint呢？构建SparkStreaming的时候，有getorCreate这个api，它就会获取checkpoint的内容，具体指定下这个checkpoint在哪就好了。而如果从checkpoint恢复后，如果数据累积太多处理不过来，怎么办?1限速2增强机器的处理能力3放到数据缓冲池中。

3）由于底层是直接读数据，没有所谓的Receiver，直接是周期性(Batch Intervel)的查询kafka，处理数据的时候，我们会使用基于kafka原生的Consumer api来获取kafka中特定范围(offset范围)中的数据。这个时候，Direct Api访问kafka带来的一个显而易见的性能上的好处就是，如果你要读取多个partition，Spark也会创建RDD的partition，这个时候RDD的partition和kafka的partition是一致的。而Receiver的方式，这2个partition是没任何关系的。这个优势是你的RDD，其实本质上讲在底层读取kafka的时候，kafka的partition就相当于原先hdfs上的一个block。这就符合了数据本地性。RDD和kafka数据都在这边。所以读数据的地方，处理数据的地方和驱动数据处理的程序都在同样的机器上，这样就可以极大的提高性能。不足之处是由于RDD和kafka的patition是一对一的，想提高并行度就会比较麻烦。提高并行度还是repartition，即重新分区，因为产生shuffle，很耗时。这个问题，以后也许新版本可以自由配置比例，不是一对一。因为提高并行度，可以更好的利用集群的计算资源，这是很有意义的。

4）不需要开启wal机制，从数据零丢失的角度来看，极大的提升了效率，还至少能节省一倍的磁盘空间。从kafka获取数据，比从hdfs获取数据，因为zero copy的方式，速度肯定更快。

#### spark读取数据，是几个Partition呢？ hdfs几个block 就有几个 Partition？

默认的是一个block块会对应生成一个Partition，也可以自定义Partition的数量

#### spark on yarn的两种模式? client 模式？ 和cluster模式？

cluster模式：Driver程序在YARN中运行，应用的运行结果不能在客户端显示，所以最好运行那些将结果最终保存在外部存储介质（如HDFS、Redis、Mysql）而非stdout输出的应用程序，客户端的终端显示的仅是作为YARN的job的简单运行状况。

client模式：Driver运行在Client上，应用程序运行结果会在客户端显示，所有适合运行结果有输出的应用程序（如spark-shell）

#### spark怎么整合hive？

Hive On Spark

#### SparkSql如何处理结构化数据和非结构化数据？

结构化数据：Json转化为DataFrame、通过注册表操作sql的方式

非结构化数据：通过反射推断方式、构建一个Schema

#### 对于Spark你觉得他对于现有大数据的现状的优势和劣势在哪里？

优势：内存级运算

劣势：对性能要求较高

#### 哪些算子操作涉及到shuffle

distinct、groupByKey、reduceByKey、aggregateByKey、join、cogroup、repartition

#### Spark的shuffle有几种方式

shuffle方式共三种，分别是：

HashShuffle

SortShuffle

TungstenShuffle

#### 简述MR的shuffle和Spark的shuffle过程?

MR：首先MR的shuffle，主要是基于磁盘计算，如果数据量过大的话，那么磁盘io就会产生过大，那么此时性能会很低，计算起来速度很慢，并且MR的shuffle计算默认是需要进行分组排序，那么此时数据量很大，那么进行分组排序的时候，每个数据都要分到相同的分区，并且还要排序，资源大大消耗，毫无效率可言。

Spark：spark计算主要是基于内存，当内存写满，才会写到磁盘，这样速度很快，并且sparkshuffle的操作可以不进行排序操作，这里可以设置，利用hashshuffle，和consolidation机制，而且shuffle计算可以迭代计算，通过这种设置，可以大大提高性能，并且缩短计算时间。

#### RANK、DENSE\_RANK以及ROW\_NUMBER的区别

它们都是对分过组的数据排序加序号；

其中,ROW\_NUMBER() 是没有重复值的排序(即使两条记录相同，序号也不重复的)，不会有同名次。

DENSE\_RANK() 是连续的排序，两个第二名仍然跟着第三名。

RANK()       是跳跃排序，两个第二名下来就是第四名。

#### stage,task和job的区别与划分方式

Job:一个由多个任务组成的并行计算，当你需要执行一个 rdd 的 action 的时候，会生成一个 job。

Stage:每个 Job 被拆分成更小的被称作 stage（阶段） 的 task（任务） 组，stage 彼此之间是相互依赖的,各个 stage 会按照执行顺序依次执行。

Task:一个将要被发送到 Executor 中的工作单元。是stage的一个任务执行单元，一般来说，一个 rdd 有多少个 partition，就会有多少个 task，因为每一个 task 只是处理一个 partition 上的数据。

#### flatmap和map的区别

map：对集合中每个元素进行操作。

flatMap：对集合中每个元素进行操作然后再扁平化

#### foreach 和map的区别

两个方法的共同点在于：都是用于遍历集合对象，并对每一项执行指定的方法。而两者的差异在于：foreach无返回值（准确说返回void）,map返回集合对象。foreach用于遍历集合，而map用于映射（转换）集合到另一个集合。

#### foreachRDD 用过吗？

foreachRDD：DStream是抽象类，它把连续的数据流拆成很多的小RDD数据块， 这叫做“微批次”， spark的流式处理， 都是“微批次处理”。 foreachRDD作用于由DStream创建的RDD,经常用来把每一个RDD的数据写出到外部文件系统，比如文件，数据库等。

#### Spark调优的几种方法？

Spark调优可以从以下几个方面着手：

性能调优

JVM调优

数据倾斜调优

Shuffle调优

RDD算子调优

序列化调优

广播变量调优等

#### Spark-Streaming获取kafka数据的两种方式,并简要介绍他们的优缺点?

Receiver方式；当一个任务从driver发送到executor执行的时候，这时候，将数据拉取到executor中做操作，但是如果数据太大的话，这时候不能全放在内存中，receiver通过WAL，设置了本地存储，他会存放本地，保证数据不丢失，然后使用Kafka高级API通过zk来维护偏移量，保证数据的衔接性，其实可以说，receiver数据在zk获取的，这种方式效率低，而且极易容易出现数据丢失

Direct 方式； 他使用Kafka底层Api 并且消费者直接连接kafka的分区上，因为createDirectStream创建的DirectKafkaInputDStream每个batch所对应的RDD的分区与kafka分区一一对应，但是需要自己维护偏移量，迭代计算，即用即取即丢，不会给内存造成太大的压力，这样效率很高

#### 数据倾斜解决方案

数据倾斜的发生一般都是一个key对应的数据过大，而导致task执行过慢，或者内存溢出，OOM，一般发生在shuffle的时候，比如reducebykey，countbykey，groupbykey，容易产生数据倾斜；

如何解决数据倾斜，首先看log日志信息，因为log日志报错时候会提示在那些行，然后就是去检查发生shuffle的地方，这些地方比较容易发生数据倾斜；

1.第一个方案就是聚合源数据，我们的数据一般来自于hive 表，那么在生成hive表的时候对数据进行聚合，按照key进行分组，将key对应的所有values以另一种格式存储，比如拼接成一个字符串这样的话，可以省略groupbykey和reducebykey的操作，那么没有这样操作的话，也就不用shuffle了，没shuffle的话不可能出现数据倾斜，如果不能完美拼接，但是能少量拼接也能减少key对应的数据量，这样也可以提高性能

2.第二种方案过滤导致倾斜的key 这种方案就是说如果业务允许或者沟通过后能理解的话，我们可以把大的key进行过滤，这样可以轻松解决问题

3.第三种方案提高shuffle操作reduce并行度（reduceByKey（new.. , 1000）） 通过提高reduce端的task执行数量，来分担数据压力，也就是说将task执行数量提高，性能也会相应提高，这样方式如果在运行中确实解决了数据倾斜问题最好了，但是如果出现之前运行时候OOM了，加大了reduce端task的数量，可以运行了，但是执行时间相当的长，那么就放弃这第三种的方案，换别的方案

4.第四种方案利用双重聚合 用于groupBykey和reduceBykey，比较试用与join，但是通常不用这样做，也就是说首先第一轮对key进行打散，将原来一样的key变成不一样的key前面加前缀，相当于将一样的key分了多个组，然后进行局部聚合，接着除掉每个key的前缀，然后在进行全局的聚合，进行两次聚合，避免数据倾斜问题

5.第五种方案将reduce join 转换成map join，如果两个rdd进行join，有一个表比较小的话，可以将小的表通过broadcast广播出去，这样每个节点的blockmanager中都有一份，这样的话根本不会发生shuffle，那么也就肯定不会存在数据倾斜的问题了，如果join中有数据倾斜的情况，第一时间考虑这样方式；但是如果两个表都很大，那么就不可以broadcast了（内存不足），还有就是用map join来代替reduce join，也就是说牺牲一点点内存，是可以接收的；

6.第六种方案sample抽样分解聚合 也就是说将倾斜的key单拉出来，然后用一个RDD进行打乱join

7.第七种方案使用随机数和扩容表进行join 也就是说通过flatmap进行扩容，然后在将随机数打入进去，在进行join，这样的话不能根本的解决数据倾斜，但是我可以有效的缓解数据倾斜的问题，也会提高性能

#### dataframe和dataset区别？

其实dataset就是dataframe的升级版，相当于dataframe是dataset的子集，主要区别在于，在spark2.0以后的dataset添加的编码器，在dataframe中他不是面向对象的编程思维，而在dataset中变成面向对象编程，同时dataset相当于dataframe和rdd的整合版，操作更加灵活

#### 说一说yarn-client模式的运行原理?

通过spark-submit shell脚本提交任务到yarn上，首先通过jvm启动一个线程执行脚本任务，然后启动driver，启动完成后driver会向RM申请启动applicationmaster，此时RM接收到请求后会通知NM启动一个applicationmaster（ExecutorLauncher），applicationmaster启动后向RM申请对应的executor，RM通知其他的NM启动executor进程，此时executor启动后，会向driver反向注册自己，driver接收到注册信息，就知道我的任务要在那个NM的executor启动了，然后开始执行相应的spark代码作业，一行一行执行，而这种模式driver只负责划分stage和task和各种的资源调度，applicationmaster负责申请executor

#### Spark读取数据生成RDD分区默认多少？

testFile算子的参数minPartitions的默认值为defaultMinPartitions，该方法的实现代码为math.min(defaultParallelism, 2)，其中defaultParallelism与CPU的核数有关系，也就是说默认分区数量是取cpu的核数和2的最小值

#### 你们的项目都做了那些优化?

spark性能调优：

解决性能调优的最简单的就是分配更多的资源；如果在资源固定的情况下再去考虑我们的代码和程序

我们可以在spark-submit shell脚本中 设置

num-executor

driver-memory

executor-memory

executor-core

这些数量可以适当的调节，根据自己的内存大小调节

有一个原则，你能使用的资源有多大，就尽量调节到最大的大小（executor的数量，几十个到上百个不等，executor内存；executor cup core）

调节executor中的core，要适中，根据自己的task数量来分配相符合的core

设置并行度：spark.default.parallelism 参数

设置公用的RDD持久化（将RDD放入缓存中/磁盘）但是如果数据过大的话，可能会导致OOM内存溢出，这时候我们要使用序例化，也就是将RDD每个partition的数据序例化成大的字节数组，大大减少内存空间的占用，缺点就是需要反序例化；如果序例化后还出现OOM，那么我们就要往磁盘上存储；如果还是OOM，那么只能内存加上磁盘然后在序例化；

还可以使用广播变量，广播出去，广播变量初始的时候，在driver中有一份数据，当task在运行时，想要用广播出去的变量数据，此时会在自己本地executor中对应的executorManage中尝试获取变量副本，如果没有的话，executorManage会去driver中通过网络拉去一份，下一次task在用的时候就有副本了，executorManage也有可能会去其他executor上拉去副本，但是前提是离得很近的情况下；

spark内部也有自己的性能优化，kryo可以对我们的数据进行序例化优化，他序例化的内存占用是java的十分之一的大小，可以大大提高性能，可以在算子函数中使用到的外部变量使用，减少网络传输的开销和占用内存的消耗，提高性能，也可以在持久化RDD优化内存的占用和消耗，减少task执行是创建对象内存占满，导致频繁的GC，还有就是在shuffle的时候，可以提高网络传输的消耗，提高性能；

要使用它的话就就要注册自己自定义的类

数据本地化有三种情况，

第一种是最好的，就是数据本地化，这种可以直接在内存中拉去数据（PROESS\_LOCAL）

然后使用跨节点的方式，通过网络传输拉去数据

最差的就是跨机架的模式，第三种是最消耗IO的模式，因为跨机架网络传输消耗太大

如何调节 spark.locality.wait 默认是3秒 观察日志，适量调节

spark.locality.wait.process

spark.locality.wait.node

spark.locality.wait.rack

#### JVM性能调优

内存不足时候会导致minor gc频繁，导致spark停止工作

频繁进行gc的时候，可能有些年轻代里面的对象应该被回收，但是因为内存性能不足，导致传入老年代里面，老年代里面如果内存溢满，那么就会进行full gc 也就是全局清理，那么full gc是很消耗时间，少则十几秒，多则几十分钟，这就导致spark性能作业大大降低。

那么如何调优呢？

降低cache操作的内存占比大不了用persist操作，现在一部分数据写入磁盘，或者序例化，配合kryo使用，减少RDD缓存的内存占用，减低cache操作的内存占用，对应的算子函数的内存占比就提升上去了，可以减少频繁的gc操作，简单说就是让task执行算子函数的时候，有更多的内存可以使用

spark.storage.memoryFraction,0.6 cache的默认占用内存是60%，可以根据自己的情况来调节

executor堆外内存的设置，如果我们出现shuffle output file not found 的错误那么我们就要调节一下堆外内存，--conf spark.yarn.executor.memoryOverhead=2048（基于yarn的提交模式）

这个参数不是在sparkContext的代码中调节，而是在spark-submit脚本中添加，这个参数调节上去以后，会避免oom内存溢出的问题

偶尔也有可能会出现连接等待超时，因为executor跨界点拉取数据的时候，可能另一个executor正在jvm垃圾回收（所有线程停止），导致连接失败会报 （sjkdfds-3231sdf-232df）not found，这时候我们要调节连接参数，要在spark-submit脚本中调节

spark.core.connection.ack.wait.timeout=300 调节这个值比较大的时候，通常可以避免某某文件lost掉了

#### shuffle性能调优：

设置map输出文件的合并设置参数 .set("spark.shuffle.consolidateFile","true")，默认是不开启的，设置true开启，注意一点，就是无法并发执行

设置map端的内存缓冲区大小，和reduce端内存的大小，这个主要针对文件太大，导致性能效率低，但是调优不是特别明显

map端参数：.set("spark.shuffle.file.buffer","64k")

reduce端参数：.set("spark.shuffle.memoryFraction","0.3")

这两个参数根据我们观察日志的读写文件的多少来调节，适量调节，如果是stand-load模式 观察4040的页面，如果是yarn模式直接进入yarn UIlog日志

当我们在shuffle过程中，要进行排序可以选择sortshuffleManager（默认的）

"spark.shuffle.manager","sort"

如果我们不需要排序的话，也可以用hashshufflemanager模式，然后要实现consolidate机制

"spark.shuffle.manager","hash"

在spark 1.5版本以后出现一个新的模式，就是钨丝Manager（tungsten-sort）这种模式内部有自己的内存机制，可以有效的防止OOm内存溢出

"spark.shuffle.manager","tungsten-sort"

还用一种是我们想用sortshuffleManager模式，但是不想让他排序，那么就要设置reduce合并文件参数

"spark.shuffle.sort.bypassMergeThreshold","550"(默认数量是200，如果不想进行排序的话就要调大文件合并数量)

Mappartitions方法，表示计算一个分区的数据，就是将一个分区内的数据，可以提高执行性能，但是也有缺点，如果将task内的数据全部放入Mappartition进行计算，那么数据量过大的话就会造成内存溢出，OOM了

coalesce算子可以缩减partition数量

设置并行度，spark.defafult.parallelism,100，但是我们用sparkSQL的话，这样设置就会失效；这是怎么办？那我们要用repartition这个算子，进行自定义分区

reduceByKey也可以对性能提升（在本地进行combiner操作），主要用类似于单词统计这样的模式

#### troubleshooting：

reduce端缓冲（buffer）大小避免OOM，当map端的task输出量特别大，那么我们适当的减小，以性能换执行方式

spark.reducer.maxSizwInFlight,48m 默认就是48m

spark作业中 常见的报错情况（shuffle file not found）也就是说，你上一个进程正在gc 但是下一个拉去的时候，拉去数据没拉去到，就会报这样一个异常

解决办法：调节参数 spark.shuffle.io.maxRetries 3 默认三次 这个参数就是说默认尝试拉去失败3次

spark.shuffle.io.retryWait 5s 这个参数是每次重新拉去时间的间隔，默认是5秒 这两个参数都可以适当的增大 避免出现这样错

#### kafka在高并发的情况下,如何避免消息丢失和消息重复?

消息丢失解决方案:

首先对kafka进行限速， 其次启用重试机制，重试间隔时间设置长一些，最后Kafka设置acks=all，即需要相应的所有处于ISR的分区都确认收到该消息后，才算发送成功

消息重复解决方案:

消息可以使用唯一id标识

生产者（ack=all 代表至少成功发送一次)

消费者 （offset手动提交，业务逻辑成功处理后，提交offset）

落表（主键或者唯一索引的方式，避免重复数据）

业务逻辑处理（选择唯一主键存储到Redis或者mongdb中，先查询是否存在，若存在则不处理；若不存在，先插入Redis或Mongdb,再进行业务逻辑处理）

#### kafka怎么保证数据消费一次且仅消费一次

幂等producer：保证发送单个分区的消息只会发送一次，不会出现重复消息

事务(transaction)：保证原子性地写入到多个分区，即写入到多个分区的消息要么全部成功，要么全部回滚

流处理EOS：流处理本质上可看成是“读取-处理-写入”的管道。此EOS保证整个过程的操作是原子性。注意，这只适用于Kafka Streams

#### kafka保证数据一致性和可靠性

数据一致性保证

一致性定义：若某条消息对client可见，那么即使Leader挂了，在新Leader上数据依然可以被读到

HW-HighWaterMark: client可以从Leader读到的最大msg offset，即对外可见的最大offset， HW=max(replica.offset)

对于Leader新收到的msg，client不能立刻消费，Leader会等待该消息被所有ISR中的replica同步后，更新HW，此时该消息才能被client消费，这样就保证了如果Leader fail，该消息仍然可以从新选举的Leader中获取。

对于来自内部Broker的读取请求，没有HW的限制。同时，Follower也会维护一份自己的HW，Folloer.HW = min(Leader.HW, Follower.offset)

数据可靠性保证

当Producer向Leader发送数据时，可以通过acks参数设置数据可靠性的级别

0: 不论写入是否成功，server不需要给Producer发送Response，如果发生异常，server会终止连接，触发Producer更新meta数据；

1: Leader写入成功后即发送Response，此种情况如果Leader fail，会丢失数据

-1: 等待所有ISR接收到消息后再给Producer发送Response，这是最强保证

#### kafka到spark streaming怎么保证数据完整性，怎么保证数据不重复消费？

保证数据不丢失（at-least）

spark RDD内部机制可以保证数据at-least语义。

Receiver方式

开启WAL（预写日志），将从kafka中接受到的数据写入到日志文件中，所有数据从失败中可恢复。

Direct方式

依靠checkpoint机制来保证。

保证数据不重复（exactly-once）

要保证数据不重复，即Exactly once语义。

- 幂等操作：重复执行不会产生问题，不需要做额外的工作即可保证数据不重复。

- 业务代码添加事务操作

就是说针对每个partition的数据，产生一个uniqueId，只有这个partition的所有数据被完全消费，则算成功，否则算失效，要回滚。下次重复执行这个uniqueId时，如果已经被执行成功，则skip掉。

#### kafka的消费者高阶和低阶API有什么区别

kafka 提供了两套 consumer API：The high-level Consumer API和 The SimpleConsumer API

其中 high-level consumer API 提供了一个从 kafka 消费数据的高层抽象，而 SimpleConsumer API 则需要开发人员更多地关注细节。

The high-level consumer API

high-level consumer API 提供了 consumer group 的语义，一个消息只能被 group 内的一个 consumer 所消费，且 consumer 消费消息时不关注 offset，最后一个 offset 由 zookeeper 保存。

使用 high-level consumer API 可以是多线程的应用，应当注意：

如果消费线程大于 patition 数量，则有些线程将收不到消息

如果 patition 数量大于线程数，则有些线程多收到多个 patition 的消息

如果一个线程消费多个 patition，则无法保证你收到的消息的顺序，而一个 patition 内的消息是有序的

The SimpleConsumer API

如果你想要对 patition 有更多的控制权，那就应该使用 SimpleConsumer API，比如：

多次读取一个消息

只消费一个 patition 中的部分消息

使用事务来保证一个消息仅被消费一次但是使用此 API 时，partition、offset、broker、leader 等对你不再透明，需要自己去管理。你需要做大量的额外工作：

必须在应用程序中跟踪 offset，从而确定下一条应该消费哪条消息

应用程序需要通过程序获知每个 Partition 的 leader 是谁

需要处理 leader 的变更

#### kafka的exactly-once

幂等producer：保证发送单个分区的消息只会发送一次，不会出现重复消息

事务(transaction)：保证原子性地写入到多个分区，即写入到多个分区的消息要么全部成功，要么全部回滚

流处理EOS：流处理本质上可看成是“读取-处理-写入”的管道。此EOS保证整个过程的操作是原子性。注意，这只适用于Kafka Streams

#### 如何保证从Kafka获取数据不丢失?

1.生产者数据的不丢失

kafka的ack机制：在kafka发送数据的时候，每次发送消息都会有一个确认反馈机制，确保消息正常的能够被收到。

2.消费者数据的不丢失

通过offset commit 来保证数据的不丢失，kafka自己记录了每次消费的offset数值，下次继续消费的时候，接着上次的offset进行消费即可。

#### spark实时作业宕掉，kafka指定的topic数据堆积怎么办？

应对措施：

①spark.streaming.concurrentJobs=10：提高Job并发数，从源码中可以察觉到，这个参数其实是指定了一个线程池的核心线程数而已，没有指定时，默认为1。

②spark.streaming.kafka.maxRatePerPartition=2000：设置每秒每个分区最大获取日志数，控制处理数据量，保证数据均匀处理。

③spark.streaming.kafka.maxRetries=50：获取topic分区leaders及其最新offsets时，调大重试次数。

④在应用级别配置重试

spark.yarn.maxAppAttempts=5

# 尝试失败有效间隔时间设置

spark.yarn.am.attemptFailuresValidityInterval=1h

此处需要【注意】：

spark.yarn.maxAppAttempts值不能超过hadoop集群中yarn.resourcemanager.am.max-attempts

#### produce向kafka中发送数据产生的offset 怎么算（给你传入几条大小的消息 求offset是多少）

什么是offset

offset是consumer position，Topic的每个Partition都有各自的offset.

消费者需要自己保留一个offset，从kafka 获取消息时，只拉去当前offset 以后的消息。Kafka 的scala/java 版的client 已经实现了这部分的逻辑，将offset 保存到zookeeper 上

1） auto.offset.reset

如果Kafka没有开启Consumer，只有Producer生产了数据到Kafka中，此后开启Consumer。在这种场景下，将auto.offset.reset设置为largest，那么Consumer会读取不到之前Produce的消息，只有新Produce的消息才会被Consumer消费

2） auto.commit.enable（例如true，表示offset自动提交到Zookeeper）

3） auto.commit.interval.ms(例如60000,每隔1分钟offset提交到Zookeeper)

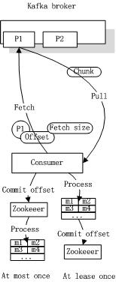
4） offsets.storage

Select where offsets should be stored (zookeeper or kafka).默认是Zookeeper

5） 基于offset的重复读

6） Kafka的可靠性保证(消息消费和Offset提交的时机决定了At most once和At least once语义)

Kafka默认实现了At least once语义



#### spark作业有多少个？

Spark Application在遇到action算子时，SparkContext会生成Job；也就是Job的个数由Action算子的个数决定。

#### 项目中当前batch获取前面的batch的算子是哪个？

updateStateByKey(func) 根据于key的前置状态和key的新值，对key进行更新，返回一个新状态的Dstream。

1）Spark Streaming的updateStateByKey可以DStream中的数据进行按key做reduce操作，然后对各个批次的数据进行累加。

2）updateStateByKey 解释:

以DStream中的数据进行按key做reduce操作，然后对各个批次的数据进行累加

在有新的数据信息进入或更新时，可以让用户保持想要的任何状。使用这个功能需要完成两步：

① 定义状态：可以是任意数据类型

② 定义状态更新函数：用一个函数指定如何使用先前的状态，从输入流中的新值更新状态。

对于有状态操作，要不断的把当前和历史的时间切片的RDD累加计算，随着时间的流失，计算的数据规模会变得越来越大

3）要思考的是如果数据量很大的时候，或者对性能的要求极为苛刻的情况下，可以考虑将数据放在Redis或者tachyon或者ignite上

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

#### 100个分片，我想聚合成两个分片，用哪个算子?

coalesce算子，主要就是用于在filter操作之后，针对每个partition的数据量各不相同的情况，来压缩partition的数量。减少partition的数量，而且让每个partition的数据量都尽量均匀紧凑。从而便于后面的task进行计算操作，在某种程度上，能够一定程度的提升性能。

#### spark对接hbase

1）Spark读取HBase中的数据

import org.apache.hadoop.hbase.{HBaseConfiguration, HTableDescriptor, TableName}

import org.apache.hadoop.hbase.client.HBaseAdmin

import org.apache.hadoop.hbase.mapreduce.TableInputFormat

import org.apache.spark.\_

import org.apache.hadoop.hbase.util.Bytes

/\*\*

\*

\* 从hbase读取数据转化成RDD

\*/

object SparkReadHBase {

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

val sparkConf = new SparkConf().setAppName("HBaseTest").setMaster("local")

val sc = new SparkContext(sparkConf)

val tablename = "account"

val conf = HBaseConfiguration.create()

//设置zooKeeper集群地址，也可以通过将hbase-site.xml导入classpath，但是建议在程序里这样设置

conf.set("hbase.zookeeper.quorum","node02,node03,node04")

//设置zookeeper连接端口，默认2181

conf.set("hbase.zookeeper.property.clientPort", "2181")

conf.set(TableInputFormat.INPUT\_TABLE, tablename)

// 如果表不存在则创建表

val admin = new HBaseAdmin(conf)

if (!admin.isTableAvailable(tablename)) {

val tableDesc = new HTableDescriptor(TableName.valueOf(tablename))

admin.createTable(tableDesc)

}

//读取数据并转化成rdd

val hBaseRDD = sc.newAPIHadoopRDD(conf, classOf[TableInputFormat],

classOf[org.apache.hadoop.hbase.io.ImmutableBytesWritable],

classOf[org.apache.hadoop.hbase.client.Result])

val count = hBaseRDD.count()

println(count)

hBaseRDD.foreach{case (\_,result) =>{

//获取行键

val key = Bytes.toString(result.getRow)

//通过列族和列名获取列

val name = Bytes.toString(result.getValue("cf".getBytes,"name".getBytes))

val age = Bytes.toInt(result.getValue("cf".getBytes,"age".getBytes))

println("Row key:"+key+" Name:"+name+" Age:"+age)

}}

sc.stop()

admin.close()

}

}

2）Spark写HBase

import org.apache.hadoop.hbase.HBaseConfiguration

import org.apache.hadoop.hbase.client.Put

import org.apache.hadoop.hbase.io.ImmutableBytesWritable

import org.apache.hadoop.hbase.mapred.TableOutputFormat

import org.apache.hadoop.hbase.util.Bytes

import org.apache.hadoop.mapred.JobConf

import org.apache.spark.{SparkConf, SparkContext}

import org.apache.spark.rdd.RDD.rddToPairRDDFunctions

/\*\*

\*

\* 使用saveAsHadoopDataset写入数据

\*/

object SparkWriteHBaseOne {

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

val sparkConf = new SparkConf().setAppName("HBaseTest").setMaster("local")

val sc = new SparkContext(sparkConf)

val conf = HBaseConfiguration.create()

//设置zooKeeper集群地址，也可以通过将hbase-site.xml导入classpath，但是建议在程序里这样设置

conf.set("hbase.zookeeper.quorum","node02,node03,node04")

//设置zookeeper连接端口，默认2181

conf.set("hbase.zookeeper.property.clientPort", "2181")

val tablename = "account"

//初始化jobconf，TableOutputFormat必须是org.apache.hadoop.hbase.mapred包下的！

val jobConf = new JobConf(conf)

jobConf.setOutputFormat(classOf[TableOutputFormat])

jobConf.set(TableOutputFormat.OUTPUT\_TABLE, tablename)

val indataRDD = sc.makeRDD(Array("1,jack,15","2,Lily,16","3,mike,16"))

val rdd = indataRDD.map(\_.split(',')).map{arr=>{

/\*一个Put对象就是一行记录，在构造方法中指定主键

\* 所有插入的数据必须用org.apache.hadoop.hbase.util.Bytes.toBytes方法转换

\* Put.add方法接收三个参数：列族，列名，数据

\*/

val put = new Put(Bytes.toBytes(arr(0).toInt))

put.add(Bytes.toBytes("cf"),Bytes.toBytes("name"),Bytes.toBytes(arr(1)))

put.add(Bytes.toBytes("cf"),Bytes.toBytes("age"),Bytes.toBytes(arr(2).toInt))

//转化成RDD[(ImmutableBytesWritable,Put)]类型才能调用saveAsHadoopDataset

(new ImmutableBytesWritable, put)

}}

rdd.saveAsHadoopDataset(jobConf)

sc.stop()

}

}

#### stage 的划分

1）spark划分stage的整体思路是：从后往前推，遇到宽依赖就断开，划分为一个stage；遇到窄依赖就将这个RDD加入该stage中。

2）在spark中，Task的类型分为2种：ShuffleMapTask和ResultTask；简单来说，DAG的最后一个阶段会为每个结果的partition生成一个ResultTask，即每个Stage里面的Task的数量是由该Stage中最后一个RDD的Partition的数量所决定的！

4）而其余所有阶段都会生成ShuffleMapTask；之所以称之为ShuffleMapTask是因为它需要将自己的计算结果通过shuffle到下一个stage中。

4）总结

map,filter为窄依赖，

groupbykey为款依赖

遇到一个宽依赖就分一个stage

#### spark submint 参数设直以及提交任务提交流程

spark提交任务常见的两种模式:

1）local[k]:本地使用k个worker线程运行saprk程序.这种模式适合小批量数据在本地调试代码用.(若使用本地的文件,需要在前面加上:file://)

2）spark on yarn模式:

    (1)yarn-client模式: 以client模式连接到yarn集群,该方式driver是在client上运行的;

    (2)yarn-cluster模式:以cluster模式连接到yarn集群,该方式driver运行在worker节点上.

    (3)对于应用场景来说,Yarn-Cluster适合生产环境，Yarn-Client适合交互和调试。

3）提交任务时的几个重要参数:



4）几个重要的参数说明:

①executor\_cores\*num\_executors

     表示的是能够并行执行Task的数目不宜太小或太大！一般不超过总队列 cores 的 25%，比如队列总 cores    400，最大不要超过100，最小不建议低于40，除非日志量很小。

②executor\_cores

     不宜为1！否则 work 进程中线程数过少，一般 2~4 为宜。

③executor\_memory

     一般 6~10g 为宜，最大不超过20G，否则会导致GC代价过高，或资源浪费严重。

④driver-memory

     driver 不做任何计算和存储，只是下发任务与yarn资源管理器和task交互，除非你是 spark-shell，否则一般 1-2g

     增加每个executor的内存量，增加了内存量以后，对性能的提升，有三点：

⑤如果需要对RDD进行cache，那么更多的内存，就可以缓存更多的数据，将更少的数据写入磁盘，

     甚至不写入磁盘。减少了磁盘IO。

⑥对于shuffle操作，reduce端，会需要内存来存放拉取的数据并进行聚合。如果内存不够，也会写入磁盘。

     如果给executor分配更多内存以后，就有更少的数据，需要写入磁盘，甚至不需要写入磁盘。减少了磁盘IO，提升了性能。

⑦对于task的执行,可能会创建很多对象.如果内存比较小,可能会频繁导致JVM堆内存满了,然后频繁GC,垃圾回收 ,minor GC和full GC.（速度很慢）.内存加大以后，带来更少的GC，垃圾回收，避免了速度变慢，性能提升。

#### 项目中遇到的问题及解决

1）软件项目开发合同的订立，合同需要对将来几个月甚至几年需要做的事情有个明确的定义说明，限定好工作范围）工作内容）承担的责任）项目总费用，每个阶段支付的费用都需要有明确的说明甚至付款条件等都需要一清二楚，很多东西都没讲明白是将来合作不愉快的导火索，这些都需要白纸黑字写清楚，其实从合同上也能看出甲乙双方的水平在什么层次上的。

2）软件开发过程中，往往会发生客户不按时支付费用的事情，因为软件开发不只是脑力活儿，也是强度非常大的体力活儿，难免会遇到不能按时交付软件的可能性，除非遇上非常有经验的能相对准确评估工作量）工期的管理人员，参考历史的开发经验）再按自身团队的开发技术能力）协调工作效率，计划出一个合理的工期计划来，因为整个公司都需要考虑到资金安全）开发风险，需要有一定的水平能说服客户及时付款，至少可以支付大部分款项的人，在开发软件项目的过程中往往会发生需要垫资几十万的事情，其间需要做好防备工作需要防止资金链断裂了。

3）软件开发人员中途离职也是家常便饭，相对规范的公司，一年也大概也会有10%的人员流动性，若薪资待遇也不怎么样）公司管理也不规范，开发人员也学不到知识）业务也不稳定的，那估计有50%的流动性也是很正常的事情，连微软）Google都会有开发人员离职现象，更何况一个不知名的公司，人员离职是很正常的现象，但是人员离职了就得需要有后备开发人员，公司管理人员需要在最短的时间内招聘到合适的人员，这也需要必备的技能。

4）现在已经不是单枪匹马就能搞定中型软件系统的年代了，一个软件项目开发过程中往往需要N多人参与，客户对软件项目的质量要求，功能要求也越来越高，不只是需要把程序写好，还需要有各种配套文档，测试都需要跟上，所以这些人的协调工作）及时沟通也是很大的问题，若一个项目经理的沟通能力有问题也很容易遇到很多没必要的麻烦，也会使得项目进展会很不顺利的局面，甚至到有敌对力量产生的程度，一个公司，一个项目最怕的是内耗，我们国家其实很多东西也都浪费在内耗上了，若没几千年封建王朝的内耗，我们应该会发展的比美国强大很多。

5）合理的安排工作计划）有目的有计划的做事情，很不容易，项目里需要完成的工作NN多，需要协调的人员NN多，需要设计实现的功能NN多，做一个软件项目并没有学习编程那么轻松愉快，更不项打网络游戏一样输了还可以从头再来，软件项目开发是不允许输了再来的，输了就需要按合同进行经济赔偿）又要丢人）又容易吃官司）还无法在这个圈子里继续生存了，至少会口碑很差了。

6）进度的把控比制定工作计划更难，我们可以制定个计划2012年开发好操作系统）2013年开发好数据库）2014年开发好编译器）开发环境，看上去很美，其实更本没那个能力实现这个计划，计划计划难免会有变化，计划目标需要不断地调整，但是调整得太大，那说明这个计划有问题不符合实际甚至是有些空洞的计划瞎搞搞而已，开发项目过程中需要分工合理，有一定的稳定性，例如今天让你ASP开发，明天PHP开发，后天C#开发，大后面又是JAVA开发，估计没几个开发人员不会被折磨疯了，工作分配也是一个道理，需要有一定的稳定性。及时的验收确认好工作安排也是需要有水平的，若开会问大家任务完成了吗？大部分都会说“快好了”，快好了可以理解为，已经完成了10%？已经完成了90%？但是剩下10%是技术难题，超级复杂的功能，那其实这并不是完成了90%，虽然开发人员理解为90%，但是可能10%都不到而已。

7）高效的会议，解决问题需要有效率，特别紧急时需要有站立式会议，项目紧急时也需要安排每天的会议，会议不适合超过20-30分钟，甚至10分钟内开好会议是最理想的，例如我们10个人参加会议，会议开了1天，那其实是超级浪费生命，如何高效的指挥大家，如何开一个高效的会议，责任明确的，能解决问题的会议是需要有一些水平的，若以前参与过牛B管理人员主持的会议，那很容易有经验了，参考别人的好处多多。

8）软件在开发测试阶段往往会有客户的需求变更，甚至有可能会有大面积的需求变更，每变更一次需求，客户会觉得这个是简单的变更，开发人员会说是超级复杂的需求变更甚至会说前面的工作都白做了，这时候需要有超级强的沟通能力，一方面尽量阻止客户发生没必要的变更，甚至彻底想清楚了再变更，每次变更都有文档记录，好向客户追加软件开发费用，其实这个除了大客户）实际强的客户外，想追加费用是难于上天的事情。只能是跟客户处理好关系）下次客户还能找你就不错了，客户的钱也不是飘来的，预算也是有限的，所以若不想把客户得罪了，还只能按着客户的变更来）顶多是把事情都讲清楚，这部分变更带来了多少工作量等等，至少按合同支付费用时，能有个协商的筹码对吧。

9）采用成熟的软件组件也会大大的促进软件项目的开发进度，这次我们工作流自己开发了一套B/S的，在网页上拖拖拉拉就可以设定好工作流的，自己也比较满意的效果，但是现在想想有接近足足开发了5个月，这个开发成本算 开发人员的工资 + 公司的房租）办公费用 +相应的管理费用 + 测试成本 ，远远超过了6万以上的成本，只是这个钱没一次性拿出，而是每个月一点点的往外付出而已。而且还花费了5个月时间，还不能确保没任何错误，其实到真正稳定好用，至少要烧掉10万了。若从项目开始开发就用合理的价格购买了一套，不用5个月时间自己开发，而是用1个月时间学会怎么用，然后剩下的4个月时间放在核心的业务系统的开发上，项目会相对来说更轻松）更顺利一些，毕竟战线就缩短了很多了，可以集中优势兵力重点突破。兵力分散乃大忌也。

10）项目经理的带头作用是不可低估的，若碰上一个天天吃喝嫖赌）天天游手好闲的项目经理，那这个项目的最后的结局就是等着赔款就可以了，其他人员看到项目是这样的人没几个SB会拼命干了，大家顶多装装样子，混混日子找找那里有更好的前途了，这里就是不是久留之地的念头没几天就产生了，我自己曾经就遇到过这样的情况，我没到半年就跑路了，公司没两年就关门大吉了，因为这样的领导不是真正干事的，顶多就是转了空子碰到到了狗屎运而已长久不来。

11）技术疑难为题外包，项目过程中遇到了一些WCF配置相关的疑难问题，前后解决了10多个问题，还是无法顺利搞定信息加密传输）电子证书SSL安全配置等等，甚至两台电脑之间的TCP方式通讯上也遇到了问题，由于手上有300多个付费用户，而且他们都是开发人员，所以把这个信息一发布，马上就有专家响应，人家2个小时就搞定问题了，支付了500元辛苦费，钱虽然少也是个心意，我也把问题搞定了，我的付费客户也从我这里赚到辛苦钱了，2个小时若都能赚500元，而且是自己擅长的事情，我想也足够可以了，有时候选择花钱办事比花时间办事更爽。

12）系统架构重构上也花费蛮多时间，由于客户是要求在分布式环境里运行系统，开发时又往往是单机上开发调试，又没充足的时间慢慢勾画）慢慢设计，工作安排往往是排得满满的，系统的架构有时候需要进行一些调整，若刚开始开发时就架构不明确）思路不严谨，到项目的中后期，整个项目就会大乱，更本经不起系统架构的重构，当然这里的架构架构重构更多的是小调整，若真的是大调整那说明刚开始的架构就是非常失败的，项目由于不是1个人开发的，若是一个人开发项目那还好说，想怎么调整就调整，现在是多个人开发项目，虽然不能比喻是航空母舰，至少像个护卫舰，想怎么拐弯就怎么拐弯不是那么容易的。

13）代码规范，代码质量检查，由于项目不是一杆子买卖，往往还担负着后期的维护，甚至部分运行工作，若项目的代码质量不好后期会有无穷无尽的痛苦，把一些问题扼杀在摇篮里，总比把问题培养大了，再去消灭得麻烦）头大，所以项目的中后期一定要安排严格的代码质量检查工作，可以找个工作效率非常高，做事情又相对仔细认真的人，来个地毯式轰炸，从头到尾把扫一眼，很多有SQL注入漏洞）有重复功能的代码）命名不合理的代码等等还是会被发现很多，毕竟项目开发中参与的人多，人多了就很容易啥鸟都有了。

14）成熟的数据库设计套路，其实数据库设计也是一门学问，看起来简单，真正想设计好也需要有硬功夫，也需要手艺精湛）技艺高超的。数据库基本上还是目前开发各种管理系统必不可少的组成部分，甚至现在还是稳定的管理信息系统的基石，所以数据库设计是否合理）至少30-40%的项目是否顺利稳定的分量是有的。

1. 成熟的功能设计套路）函数命名套路）窗体命名）变量命名等等，也会大大的减少项目的开发周期，项目前期需要把例子程序都写好，适当的进行一些培训工作，然后让大家模仿例子程序就可以了，例子程序不适合写得超级复杂，功能超级强大，只要能把主要核心思想都表明了就可以了，最好还是拿个投影讲一下比较好，这样大家的印象也会更深刻一些。

#### 项目的数据怎么跑的 ，测试过程出现问题怎么查看

1）大数据分析其实没有那么玄乎，对于测试来说你可以这样分解一下。首先大数据分析=一个计算器，只不过可能是一个性能功能很强大的计算器；如果把计算器视作黑盒，那么你可以沿用构造输入，验证输出的模式来制作测试用例。但输入部分如何构造海量数据，如何随机插值这就需要你自己研究了。其次如果我们关注计算器本身，那么涉及到的一般就是性能测试。所以目前好像并没有在大数据分析上分化出一个独立的测试分支。

2）一般来说大数据领域并没有说对应传统软件开发那样的测试岗位，或者相关的职能。如果说框架测试，或者部分分布式存储的读写测试，其实一般的大数据平台开发工程师来胜任，来做框架的初期选型调研，或者一些分布式文件系统的读写性能测试等等。总的来说，这个模块相对较少相关的需求，即使有，偏集群运维的数据开发工程师也是可以做的，所以应该是没有专门的测试岗位的。

3）由于数据是随机生成的，所以如果代码有明显的漏洞，很容易就拍出来(尤其是一些细节上的问题)，当然也有代码在随机数据的情况下表现的非常好，但是会被构造的数据卡掉，可以尝试构造极端的数据来进行测试。

4）总的来说对拍对的代码不一定就是正确的，遇到错误时最好还是先再理一遍自己的思路，跟着自己的代码走一遍，确认思路没有错之后再使用对拍。

5）数据的建立不是短时间内就能完成的，它需要一个长期的数据积累、沉淀、分析等形成一个庞大的数据库，它是一个很大的概念。

#### flatMap 算子怎么压平的

1）flatMap：对集合中每个元素进行操作然后再扁平化。

2）理解扁平化可以举个简单例子

val arr=sc.parallelize(Array(("A",1),("B",2),("C",3)))

arr.flatmap(x=>(x.\_1+x.\_2)).foreach(println)

输出结果为：

A

1

B

2

C

3

1. 所以flatMap扁平话意思大概就是先用了一次map之后对全部数据再一次map。

#### reduceBykey 能否替换掉 groupBykey

不能。用法不同。

1）reduceByKey用于对每个key对应的多个value进行merge操作，最重要的是它能够在本地先进行merge操作，并且merge操作可以通过函数自定义。

2）groupByKey也是对每个key进行操作，但只生成一个sequence。需要特别注意“Note”中的话，它告诉我们：如果需要对sequence进行aggregation操作（注意，groupByKey本身不能自定义操作函数），那么，选择reduceByKey/aggregateByKey更好。这是因为groupByKey不能自定义函数，我们需要先用groupByKey生成RDD，然后才能对此RDD通过map进行自定义函数操作。

#### 手写 wordCount scala 分 几个stage

object WordCount

{

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

val conf=new SparkConf().setAppName("WordCount")

//创建SparkContext对象

val sc=new SparkContext(conf)

//TODO WordCount的主要流程,saveAsTextFile这个Action才开始提交任务

sc.textFile(args(0)).flatMap(\_.split(" ")).map((\_,1)).reduceByKey(\_+\_).saveAsTextFile(args(1))

//释放资源

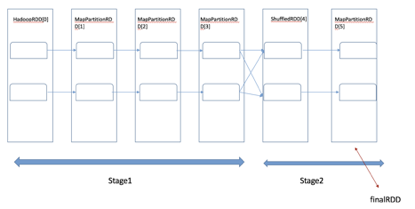
sc.stop()

}

}

主要是从HDFS读取文件后进行单词切割,然后进行计数,如果不懂RDD算子可以看RDD详解

WordCount的各个算子



#### map和tuple 的区别

1. 默认情况下Map构造的是不可变的集合，里面的内容不可修改，一旦修改就变成新的Map，原有的Map内容保持不变；
2. Map的实例是调用工厂方法模式apply来构造Map实例，而需要主要的是Map是接口，在apply中使用了具体的实现；
3. Map中存放的时键值对；
4. Tuple中可以有很多不同类型的数据；
5. Tuple的另外一个非常重要的使用是作为函数的返回值，在Tuple中返回若干个值

#### spark on yarn模式下作业提交处理流程

Spark在yarn集群上的部署方式分为两种，yarn client（driver运行在客户端）和yarn cluster（driver运行在master上）

yarn cluster（driver运行在master上）

(1) Spark Yarn Client向YARN中提交应用程序，包括Application Master程序、启动Application Master的命令、需要在Executor中运行的程序等；

(2) Resource manager收到请求后，在其中一个node manager中为应用程序分配一个container，要求它在container中启动应用程序的Application Master，Application master初始化sparkContext以及创建DAG Scheduler和Task Scheduler。

(3) Application master根据sparkContext中的配置，向resource manager申请container，同时，Application master向Resource manager注册，这样用户可通过Resource manager查看应用程序的运行状态

(4) Resource manager 在集群中寻找符合条件的node manager，在node manager启动container，要求container启动executor，

(5) Executor启动后向Application master注册，并接收Application master分配的task

(6) 应用程序运行完成后，Application Master向Resource Manager申请注销并关闭自己。

yarn client（driver运行在客户端）

(1) Spark Yarn Client向YARN的Resource Manager申请启动Application Master。同时在SparkContent初始化中将创建DAG Scheduler和TASK Scheduler等

(2) ResourceManager收到请求后，在集群中选择一个NodeManager，为该应用程序分配第一个Container，要求它在这个Container中启动应用程序的ApplicationMaster，与YARN-Cluster区别的是在该ApplicationMaster不运行SparkContext，只与SparkContext进行联系进行资源的分派

(3) Client中的SparkContext初始化完毕后，与Application Master建立通讯，向Resource Manager注册，根据任务信息向Resource Manager申请资源(Container)

(4) 当application master申请到资源后，便与node manager通信，要求它启动container

(5) Container启动后向driver中的sparkContext注册，并申请task

(6) 应用程序运行完成后，Client的SparkContext向ResourceManager申请注销并关闭自己。

#### spark sql用的多吗？什么业务场景用的？

在不涉及算法的业务场景中几乎都可以用到，比如用户纬度、区域纬度分析等

#### Spark的通信机制？

①概述：Spark的远程进程通信（RPC）是通过Akka类库来实现的，Akka使用scala语言开发，基于Actor并发模型实现，Akka具有高可靠、高性能、可扩展等特点。

②具体通信流程：

a)前提：首先启动Master进程，然后启动所有的Worker进程。

b)Worker启动后，在preStart方法中与Master建立连接，向Master发送注册信息，将Worker的信息通过case class封装起来发送给Master。

c)Master接收到Worker的注册消息后将其通过集合保存起来，然后向Worker反馈注册成功的消息。

d)Worker会定期向Master发送心跳包，领受新的计算任务。

e)Master会定期清理超时的Worker。

#### RDD的属性？

①一组分区（Partition）

a)是RDD的基本组成单位。针对于RDD来说，每个Partition都会被一个计算任务（Task）处理，其决定了并行计算的粒度。

b)可以在创建RDD的时候指定RDD分区的个数，如果没有指定，那么就会使用默认值。默认值是CPU Core的数目。

②一个计算每个分区的函数

a)Spark中RDD的计算是以分区为单位的，每个RDD都会实现compute函数以达到这个目的。

b)compute函数会对迭代器进行复合，不需要保存每次计算的结果。

③RDD之间的依赖关系

a)RDD的每次转换都会生成一个新的RDD,RDD之间就会形成类似流水线一样的前后依赖关系。

b)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，而不是对所有RDD进行重新计算。

④一个Partitioner，即RDD的分片函数。

a)当前Spark中实现了两种类型的分片函数，一个是基于HashPartitioner，另外一个是基于范围的RangePartitioner。

b)只有对于key-value的RDD，才会有Partitioner，非key-value的RDD的Partitioner的值是None。

c)partitioner函数不但决定了RDD本身的分片数，也决定了Parent RDD Shuffle输出的分片数量。

⑤一个列表，存储存取每个Partition的优先位置（preferred location）

a)对于HDFS而言，该列表保存的就是每个partition所在的块的位置。

b)按照“移动数据不如移动计算”的理念，spark在进行计算任务调度的时候，会尽可能将计算任务分配到其所要处理数据块的存储位置。

#### Spark SQL的原理？

①对读入的SQL语句进行解析

a)分辨出SQL语句中哪些是关键词（如：select,from,where）、哪些是表达式、哪些是Projection、哪些是Data Source等。

b)判断SQL语句是否规范

②将SQL语句和数据库的数据字典进行绑定

a)数据字典：列、表、视图等等。

b)若相关的Projection、DataSoruce等都是存在的话，就表示该SQL语句是可以执行的。

③数据库选择最优的执行计划

a)数据库会提供几个执行计划，这些计划都会运行统计数据。

b)数据库会从上述各种执行计划中选择一个最优计划。

④执行计划

①按照Operation(操作)~>DataSource(数据源)~>Result的次序来执行。

②在执行过程中有时候甚至不需要读取物理表就可以返回结果，比如重新运行刚运行过的SQL语句，可直接从数据库的缓冲池中返回结果。

#### scala的高阶函数？

关于函数：将可以做为参数传递到方法中的表达式称之为"函数"。

高阶函数：接受函数字面量作为参数的函数。

Scala中的高阶函数包括：作为值的函数、匿名函数、闭包和柯里化等等。

作为值的函数：将函数赋值给变量的函数。

匿名函数：没有将函数赋值给变量的函数。

闭包：是一个函数，该函数的返回值依赖于声明在函数外部的一个或是多个变量。

柯里化：将原来接受两个参数的方法变成接受一个参数方法的过程。

#### 为什么说scala是函数式编程？

Scala作为支持函数式编程的语言，可以将函数作为对象即所谓“函数是一等公民”。

#### 说一下scala的下划线有哪些作用？

1）用于替换Java的等价语法

①导入通配符。

如：import java.util.\_

②类成员的默认初始值

如：var s:String = \_

③可变长参数

如：printArgs(params: String\*)

④类型通配符

如：printArgs2(params: List[\_])

2）模式匹配

①默认匹配

如：case \_ => println("匹配默认")

②匹配集合元素

如： //匹配以0开头，长度任意的列表

case List(0, \_\*) => println("找到了哦...")

3）Scala特有的语法

①访问Tuple中的元素

如：val t = (4, "独孤求败", true)

println(t.\_2,t.\_1, t.\_3)

②简写函数字面量

如果函数的参数在函数体内只出现一次，则可以使用下划线代替

如： val f1 = (\_: Int) + (\_: Int)

//等价于

val f2 = (x: Int, y: Int) => x + y

list.foreach(println(\_))

//等价于

list.foreach(e => println(e))

list.filter(\_ > 0)

//等价于

list.filter(x => x > 0)

③定义一元操作符

利用下划线可以定义左置操作符

如：scala中的负数就是使用左置操作符实现的

-2

等价于

2.unary\_-

④定义赋值操作符

通过下划线精准地控制赋值过程

如： m.name = "Foo..." //等价于: m.name\_=("Foo...")

⑤定义部分应用函数

为某个函数只提供部分参数进行调用，返回结果为一个新的函数，即为：部分应用函数（partially applied function）。

如：val b = sum(1, \_: Int, 3)

⑥将方法转换成函数

在scala中，利用下划线可以将方法转换成函数

如： List(3,4,5).foreach(println \_)

#### Scala的隐式转换？

关于隐式转换：

①通过隐式转换，可以在编写程序时不用书写那些冗长的、过于细节的代码，让编译器在编译时自动推导出这些信息来。

②使用隐式转换可以极大地减少代码量。

使用方式：

①将方法或是变量标识为隐式的（implicit）

②将方法的参数列表标记为隐式的（implicit）

③将类标记为隐式的（implicit）

隐式转换的时机：

①当方法中的参数类型与目标类型不一致时

②对象调用类中不存在的方法或是成员时，编译器会自动将对象进行隐式转换

#### 说一下RPC框架？

Hadoop的RPC框架介绍：

①RPC采用客户端、服务器模式。

请求程序就是一个客户端，而服务提供程序就是一个服务器。

②RPC是分布式计算中Client/Server模型的一个应用实例。

③RPC具有如下特点：

a)透明性

b)高性能

c)可控性

④使用Hadoop的RPC分成4个步骤：

a)定义RPC协议

b)实现RPC协议

c)构造并启动RPC Server

d)构造RPC Client并发送RPC请求

Spark的RPC框架介绍：

Spark 1.6之前，Spark的RPC是基于Akka来实现的。Akka是一个基于Scala语言的异步消息框架。

Spark 1.6之后，Spark借鉴Akka的设计自己实现了一个基于Netty的RPC框架。

#### Spark是怎么优化的？

从以下几个方面进行优化：

1）开发调优

①避免创建重复的RDD

②尽可能复用同一个RDD

③对多次使用的RDD进行持久化

④尽量避免使用shuffle类算子

⑤使用map-side预聚合的shuffle操作

⑥使用高性能的算子

⑦广播大变量

⑧使用Kryo优化序列化性能

⑨优化数据结构

⑩提高并行度

⑪数据本地化

2）资源调优

①资源参数调优

②Java虚拟机垃圾回收调优

3）数据倾斜调优

①使用Hive ETL预处理数据

②过滤少数导致倾斜的key

③提高shuffle操作的并行度

④两阶段聚合（局部聚合+全局聚合）

⑤将reduce join转为map join

⑥采样倾斜key并分拆join操作

⑦使用随机前缀和扩容RDD进行join

⑧多种方案组合使用

4）各种Shuffle调优

#### sortByKey这个算子是全局排序吗？

是全局排序。

排序的内幕：

①在sortByKey之前将数据使用partitioner根据数据范围来分。

②使得p1分区所有的数据小于p2，p2分区所有的数据小于p3，以此类推。（p1~pn是分区标识）

③然后，使用sortByKey算子针对每一个Partition进行排序，这样全局的数据就被排序了。

#### 如何用Spark的算子实现取topN操作?

关于spark算子实现topN:

①利用分布式计算的优势，在各个分区生成一个本地的topN。

②Spark提供了topN的算子, 使用takeOrdered()，如下所示：

val topNResult = RDD的实例.takeOrdered(N)(implicit Ordering[T])

#### map与mapPartitions的区别：

同：map与mapPartitions都属于Transformation（转换）算子。

异：

①本质：

a) map是对rdd中的每一个元素进行操作

b) mapPartitions则是对rdd中的每个分区的迭代器进行操作

②RDD中的每个分区数据量不大的情形：

a) map操作性能低下

比如一个partition中有1万条数据，那么在分析每个分区时，function要执行和计算1万次。

b) mapPartitions性能较高

使用mapPartitions操作之后，一个task仅仅会执行一次function，function一次接收所有的partition数据。只要执行一次就可以了，性能比较高。

③RDD中的每个分区数据量超大的情形，比如一个Partition有100万条数据。

a)map能正常执行完

b)mapPartitions一次传入一个function后，可能一下子内存不够用，造成OOM（内存溢出）。

#### foreach和foreachPartition的区别

同： foreach和foreachPartition都属于行动（Action）算子。

异：

用法：

①foreach每次处理RDD中的一条数据。

②foreachPartition每次处理RDD中每个分区的迭代器中的数据。

#### groupByKey,reduceByKey和combineByKey的区别

1）groupByKey：

①用于对每个key进行操作，将结果生成一个sequence。

②groupByKey本身不能自定义函数。

③会将所有键值对进行移动，不会进行局部merge。

④会导致集群节点之间的开销很大，导致传输延时。

2）reduceByKey：

①用于对每个key对应的多个value进行merge操作。

②该算子能在本地先进行merge操作。

③merge操作可以通过函数进行自定义。

3）combineByKey：

①是一个比较底层的算子。

②reduceByKey底层就使用了combineByKey。

③该算子用法为：

combineByKey(createCombiner,mergeValue,mergeCombiners,partitioner,mapSideCombine)

a)参数1之createCombiner，第一次遇到key时会创建一个组合器函数（V=>C）。如：（x:Int）=>(List(x),1)

b)参数2之mergeValue，再次遇到相同key时，进行值的合并（C,V）=>C。如：(ele:(List[String],Int),x:String) =>(List[String],Int)

c)参数3之mergeCombiners，合并组合器函数（将C类型值两两合并成一个C类型）。

d)参数4之partitioner，使用已有或是自定义的分区函数（默认是HashPartitioner）。

e)参数5之mapSideCombine，是否在map端进行Combine，默认是true。

#### flume获取数据是用什么方式

fume获取数据有如下方式：

①从网络端口收集数据

②从指定的目录下获取所有文件中的数据

③从kafka消息队列中获取数据

④结合flume-ng-sql-source插件，从传统的关系型数据库中采集数据

#### spark sql中缓存方式有哪几种？registerTempTable是action类型的么,发生不发生缓存

spark sql中的缓存方式：

方式1：可以通过SQLConext实例.cacheTable("表名")缓存一张临时表

方式2：可以通过DataFrame实例.cache()缓存一张虚拟表

registerTempTable不是action类型的算子，不发生缓存。

#### 用spark core,不用spark sql怎么写入数据库

使用：RDD实例.foreachPartition(Iterator实例)

说明： ①每循环一次，用于分析每一个Partition中的数据。

②每个Partition中的数据置于了一个Iterator中。

③循环遍历Iterator，取出其中的数据，存入到RDBMS（如：mysql）中去。

代码片段参考：

//将结果保存到mysql中

df.foreachPartition(itr => {

//获得连接的实例

classOf[Driver]

val conn:Connection = DriverManager.getConnection("jdbc:mysql://JANSON01:3306/spark?useUnicode=true&characterEncoding=UTF-8","root","88888888")

//将当前分区中的数据保存到表中

val statement: PreparedStatement= conn.prepareStatement("insert into tb\_person values(?,?,?)")

//将当前分区中的数据保存到表中

insertToDB(itr, statement)

//释放资源

releaseResource(conn, statement)

})

//资源释放

spark.stop()

}