# 千锋好程序员大数据面试宝典

# 第一部分：Java语言基础

###### 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。

1. **String 是最基本的数据类型吗? 【基础】**
   1. 答：不是。
2. **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;
3. **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);

}

}

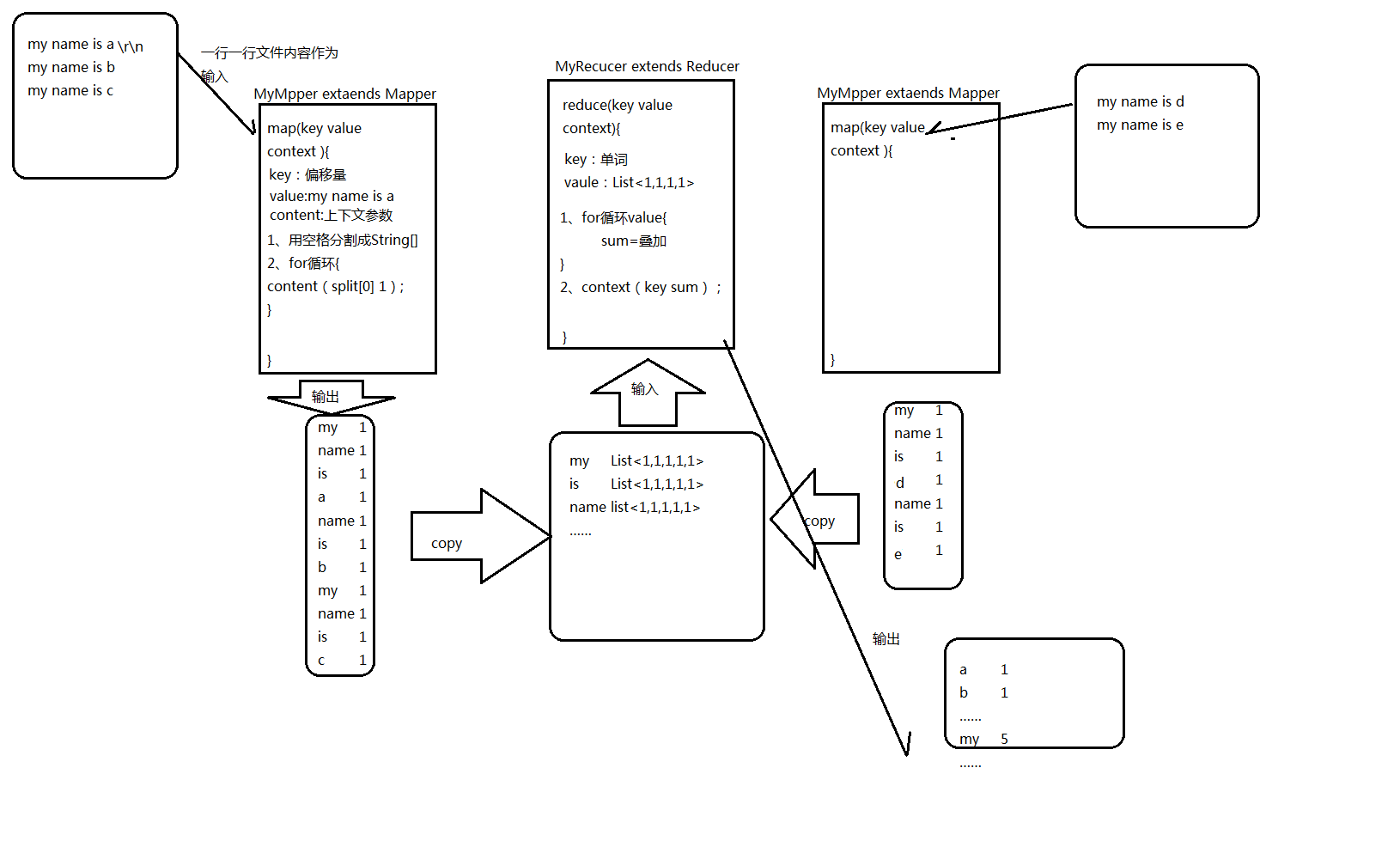
return ret;

* 1. }

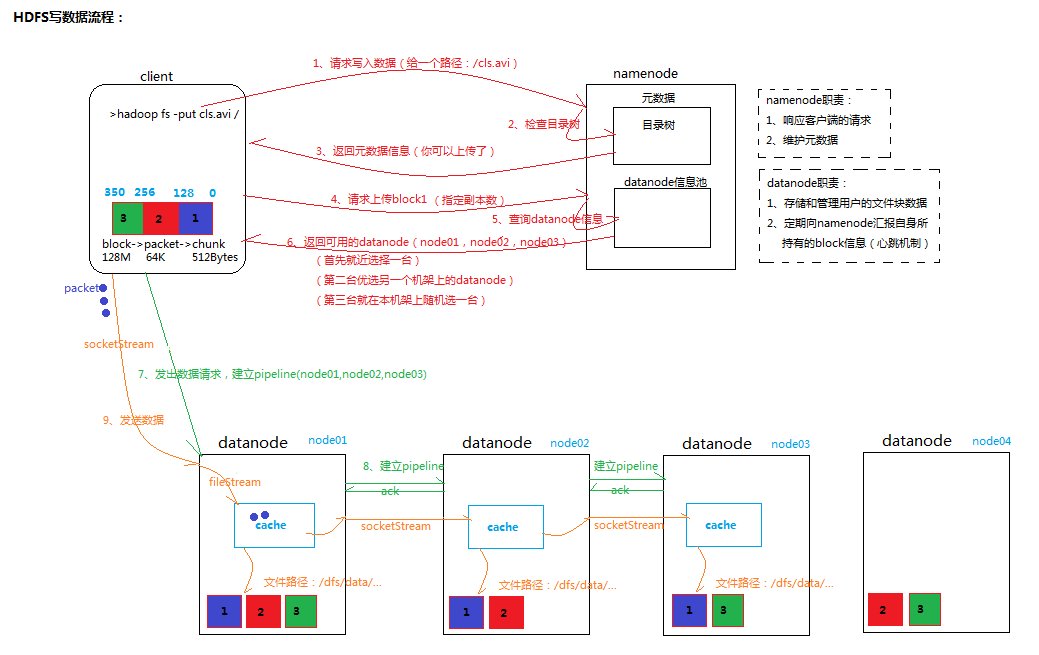
1. **字符串操作：如何实现字符串的反转及替换？【基础】**
   1. 答：可用字符串构造一StringBuffer 对象,然后调用StringBuffer 中的reverse方法即可实现字符串的反转,调用replace 方法即可实现字符串的替换。
2. **日期和时间：**
   1. 1)如何取得年月日、小时分秒？
   2. 2)如何取得从1970 年到现在的毫秒数？
   3. 3)如何取得某个日期是当月的最后一天？
   4. 4)如何格式化日期？【基础】
   5. 答：
   6. 1)创建java.util.Calendar 实例(Calendar.getInstance()),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

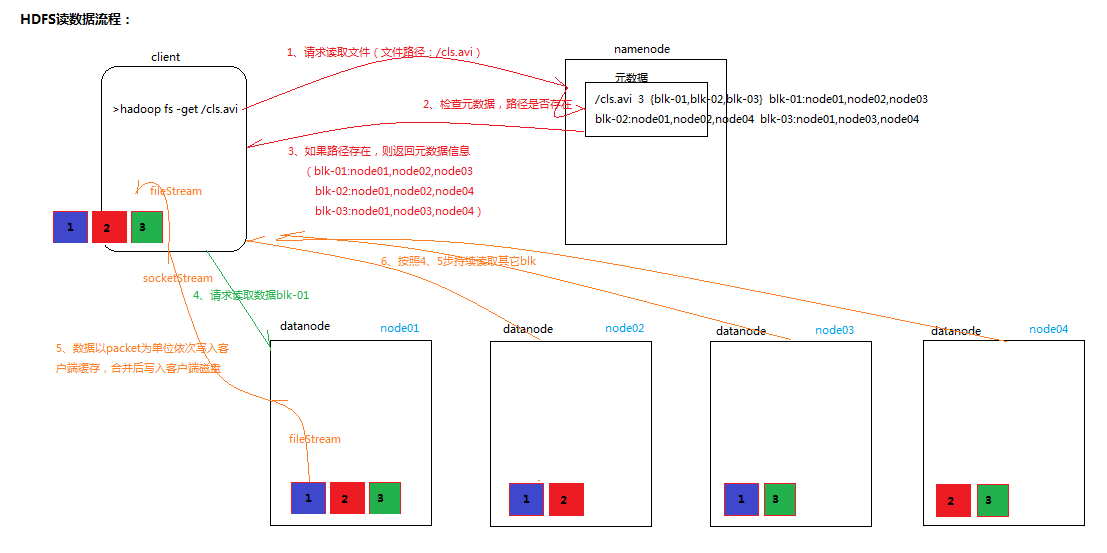
1. **通常情况下集群模式的瓶颈在哪?**
   1. 首先，瓶颈一般是指在整体中的关键限制因素，磁盘IO是指数据往磁盘读写，现在科技速度最快的属[固态硬盘](https://www.baidu.com/s?wd=%E5%9B%BA%E6%80%81%E7%A1%AC%E7%9B%98&tn=SE_PcZhidaonwhc_ngpagmjz&rsv_dl=gh_pc_zhidao%22%20%5Ct%20%22https://zhidao.baidu.com/question/_blank)了，读的速度很大有1G/秒左右，但是写入速度最快几百兆/秒，集群中数据在cpu和内存之间速度快的可以忽略，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。

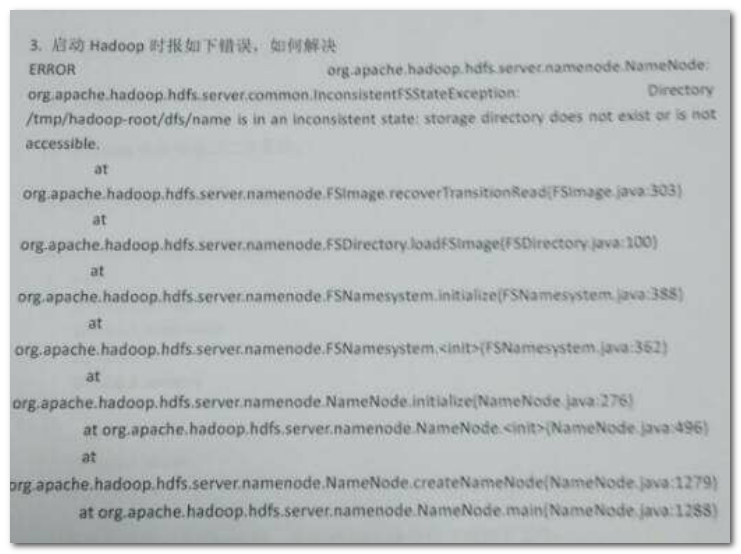
1. **举一个简单的例子说明MapReduce是怎么来运行的 ?**
   1. WordCount的例子
2. **用MapReduce来实现下面需求？现在有10个文件夹,每个文件夹都有1000000个url.现在让你找出top1000000url。**

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spark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//开启局部聚合
   8. //产生两个mr 1->map结果随机散布到reduce相同的key会分开,mr2->将mr1的结果进行总的汇总
   9. set hive.groupby.skewindata=false/true(建议开启)
   10. //开启列和分区裁剪 即不要扫描不需要的列和分区
   11. 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，一部分(20%)是用来存储数据的索引，一部分(80%)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

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。

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

1. **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
2. **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。
3. **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
4. **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

HIVE的优化

(并发执行/动态分区/推测执行/多group by合并/合并小文件/自定义mapreduce数量/使用自动索引/combinner聚合)

1. 通用设置

hive.optimize.cp=true：列裁剪

hive.optimize.prunner：分区裁剪

hive.limit.optimize.enable=true：优化LIMIT n语句

hive.limit.row.max.size=1000000：

hive.limit.optimize.limit.file=10：最大文件数

1. 本地模式(小任务)

job的输入数据大小必须小于参数：hive.exec.mode.local.auto.inputbytes.max(默认128MB)

job的map数必须小于参数：hive.exec.mode.local.auto.tasks.max(默认4)

job的reduce数必须为0或者1

hive.exec.mode.local.auto.inputbytes.max=134217728

hive.exec.mode.local.auto.tasks.max=4

hive.exec.mode.local.auto=true

hive.mapred.local.mem：本地模式启动的JVM内存大小

1. 并发执行

hive.exec.parallel=true ，默认为false

hive.exec.parallel.thread.number=8

1. Strict Mode：

hive.mapred.mode=true，严格模式不允许执行以下查询：

分区表上没有指定了分区

没有limit限制的order by语句

笛卡尔积：JOIN时没有ON语句

1. 动态分区

hive.exec.dynamic.partition.mode=strict：该模式下必须指定一个静态分区

hive.exec.max.dynamic.partitions=1000

hive.exec.max.dynamic.partitions.pernode=100：在每一个mapper/reducer节点允许创建的最大分区数

DATANODE：dfs.datanode.max.xceivers=8192：允许DATANODE打开多少个文件

1. 推测执行

mapred.map.tasks.speculative.execution=true

mapred.reduce.tasks.speculative.execution=true

hive.mapred.reduce.tasks.speculative.execution=true;

1. 多个group by合并

hive.multigroupby.singlemar=true：当多个GROUP BY语句有相同的分组列，则会优化为一个MR任务

1. 虚拟列

hive.exec.rowoffset：是否提供虚拟列

1. 分组

两个聚集函数不能有不同的DISTINCT列，以下表达式是错误的：

INSERT OVERWRITE TABLE pv\_gender\_agg SELECT pv\_users.gender, count(DISTINCT pv\_users.userid), count(DISTINCT pv\_users.ip) FROM pv\_users GROUP BY pv\_users.gender;

SELECT语句中只能有GROUP BY的列或者聚集函数。

1. Combiner聚合

hive.map.aggr=true;在map中会做部分聚集操作，效率更高但需要更多的内存。

hive.groupby.mapaggr.checkinterval：在Map端进行聚合操作的条目数目

数据倾斜

hive.groupby.skewindata=true：数据倾斜时负载均衡，当选项设定为true，生成的查询计划会有两个MRJob。

第一个MRJob 中，Map的输出结果集合会随机分布到Reduce中，每个Reduce做部分聚合操作，并输出结果，这样处理的结果是相同的GroupBy Key

有可能被分发到不同的Reduce中，从而达到负载均衡的目的；

第二个MRJob再根据预处理的数据结果按照GroupBy Key分布到Reduce中（这个过程可以保证相同的GroupBy Key被分布到同一个Reduce中），最后完成最终的聚合操作。

1. 排序

ORDER BY colName ASC/DESC

hive.mapred.mode=strict时需要跟limit子句

hive.mapred.mode=nonstrict时使用单个reduce完成排序

SORT BY colName ASC/DESC ：每个reduce内排序

DISTRIBUTE BY(子查询情况下使用 )：控制特定行应该到哪个reducer，并不保证reduce内数据的顺序

CLUSTER BY ：当SORT BY 、DISTRIBUTE BY使用相同的列时。

1. 合并小文件

hive.merg.mapfiles=true：合并map输出

hive.merge.mapredfiles=false：合并reduce输出

hive.merge.size.per.task=256\*1000\*1000：合并文件的大小

hive.mergejob.maponly=true：如果支持CombineHiveInputFormat则生成只有Map的任务执行merge

hive.merge.smallfiles.avgsize=16000000：文件的平均大小小于该值时，会启动一个MR任务执行merge。

1. 自定义map/reduce数目

减少map数目：

　　set mapred.max.split.size

　　set mapred.min.split.size

　　set mapred.min.split.size.per.node

　　set mapred.min.split.size.per.rack

set hive.input.format=org.apache.hadoop.hive.ql.io.CombineHiveInputFormat

增加map数目：

当input的文件都很大，任务逻辑复杂，map执行非常慢的时候，可以考虑增加Map数，来使得每个map处理的数据量减少，从而提高任务的执行效率。

假设有这样一个任务：

select data\_desc, count(1), count(distinct id),sum(case when …),sum(case when ...),sum(…) from a group by data\_desc

如果表a只有一个文件，大小为120M，但包含几千万的记录，如果用1个map去完成这个任务，肯定是比较耗时的，这种情况下，我们要考虑将这一个文件合理的拆分成多个，这样就可以用多个map任务去完成。

　　set mapred.reduce.tasks=10;

　　create table a\_1 as select \* from a distribute by rand(123);

这样会将a表的记录，随机的分散到包含10个文件的a\_1表中，再用a\_1代替上面sql中的a表，则会用10个map任务去完成。每个map任务处理大于12M（几百万记录）的数据，效率肯定会好很多。

reduce数目设置：

参数1：hive.exec.reducers.bytes.per.reducer=1G：每个reduce任务处理的数据量

参数2：hive.exec.reducers.max=999(0.95\*TaskTracker数)：每个任务最大的reduce数目

reducer数=min(参数2,总输入数据量/参数1)

set mapred.reduce.tasks：每个任务默认的reduce数目。典型为0.99\*reduce槽数，hive将其设置为-1，自动确定reduce数目。

1. 使用索引：

hive.optimize.index.filter：自动使用索引

hive.optimize.index.groupby：使用聚合索引优化GROUP BY操作

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

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

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

1. **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。

1. **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

1. **kafak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）

1. **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
2. **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。
3. **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
4. **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。

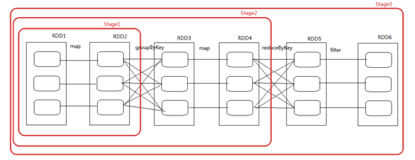
3、RDD之间的依赖关系。RDD的每次转换都会生成一个新的RDD，所以RDD之间就会形成类似于流水线一样的前后依赖关系。在部分分区数据丢失时，Spark可以通过这个依赖关系重新计算丢失的分区数据，而不是对RDD的所有分区进行重新计算。

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。

1. **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。

1. **stage怎么划分的**
   1. 
2. **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

1. **kakfadirect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。

1. **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

1. **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）

1. **哪些算子操作涉及到shuffle**

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

1. **spark怎么整合hive？**

Hive On Spark

1. **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

1. **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

1. **Spark的shuffle有几种方式**

shuffle方式共三种，分别是：

HashShuffle

SortShuffle

TungstenShuffle

1. **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。

1. **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。

1. **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。

1. **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

1. **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。

1. **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。

1. **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

1. **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

1. **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

1. **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

1. **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

1. **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

1. **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

1. **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

1. **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

1. **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

1. **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）

1. **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

1. **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

1. **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。

1. **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

1. **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

1. **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。

1. **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

1. **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。

1. **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。

1. **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。

1. **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()

}

}

1. **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

1. **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。

1. **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)

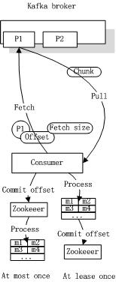
4） offsets.storage

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



1. **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。
2. **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。

1. **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。
2. **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。

1. **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

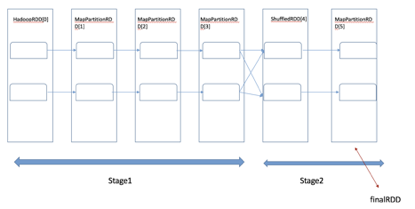
sc.stop()

}

}

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



1. **map和tuple 的区别**
2. 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；
3. 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；
4. Map中存放的时键值对；
5. Tuple中可以有很多不同类型的数据；
6. 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
7. **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。

1. **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

1. **RDD定义**

**(弹性分布式数据集)**

弹性体现在计算之上,代表的是spark可以做到在做分布式计算的时候可以容错,比如说某个节点挂了,或者是某个节点数据丢失了,可以用RDD的机制进行数据修复,这是计算层面得.

分布式也就是数据可以跨节点存储在不同的节点上,spark计算的时候也一样,代码可以运行在一个和多个节点之上,提升计算效率.

数据集在spark中可以通过读文件创建,或者通过编程的方式创建,或者通过一个数据集转换成另一个数据集,类似于hdfs的block.

1. **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。

1. **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。

1. **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。

1. **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”。

1. **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 \_)

1. **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

1. **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。

1. **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

1. **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。

1. **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])

1. **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）。

1. **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。

1. **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。

1. **flume获取数据是用什么方式**

flume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

1. **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。

1. **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()

}

1. **Yarn的运行原理(Cluster)**

1. 首先用户提交任务请求到Yarn（ResourceManager）

2. ResourceManager将去分配任务给NodeManager（启动APPMaster）

3. NodeManager将去启动ApplicationMaster（Driver）

4. 当APPMaster启动后创建对应的对象（DAG和Task）

5. 然后NodeManager进行分配资源到Container中，启动对应的Executor

6. 此时我们的APPMaster只负责和Container中的Executor进行交互发送任务和心跳检测

7. 任务的调度和资源的分配现在就属于ResourceManager负责进行操作

8. 当下一个任务进行提交也是上面相同操作，但是还是FIFO模式！

1. **spark源码内部调度（DAGScheduler和TaskScheduler）**

1. 首先找到一个Action算子（比如：foreach）

2. 会发现有sc.runJob，开始执行任务调度了（下面是经典实现）

3. 继续点击runJob（提交任务，要点击好几层才会发现内幕）

4. 找到dagScheduler.runJob 接下来就开始执行DAG内部实现

5. 我们点进runJob后，你会发现此时有一个方法放在眼前，submitJob是用来提交Job任务的，然后他返回Job提交信息这里会有一个阻塞线程存在，用来做提交等待的操作

6. 点入SubmitJob这个方法后，里面会首先获取分区数，判断是否有任务执行

7. 内部调用eventProcessLoop->new DAGSchedulerEventProcessLoop(this)->onReceive->doOnReceive(event)->JobSubmitted(到这里，就是真正的Stage划分开始了，内部会很复杂)

8. handleJobSubmitted进入内部，首先会创建一个ResultStage对象，用于进行切分Stage， getOrCreateShuffleMapStage此方法用于划分宽窄依赖（getShuffleDependencies）

9. 当所有的依赖区分完成后，将放入ActiveJob中，进行等待执行，然后调用getMissingParentStages方法进行stage划分，此方法内有一个嵌套方法visit方法，是划分stage的核心

10. 内部会有一个visit方法，他首先会去判断你的stage获取的时候，会不会有依赖关系，如果有依赖关系，那么将调用getOrCreateShuffleMapStage，进行RDD的依赖划分，然后在将划分好的RDD存入到我们的之前未准备好的集合中，那么此时就是stage划分一个阶段，然后如果没有shuffle发生，将循环调用waitingForVisit方法，进行在依赖的调用，直至调用所有的依赖完成，然后将所有的划分stage的操作存入missing集合中，放回成List集合，这样所有的依赖完成。

11. 在构建一个Job的时候 其实看得出是从最后一个RDD开始创建ResultStage，然后不停遍历自己的父RDD的依赖，并且查看是否之前持久化过（包括缓存，以及Checkpoint）若没有就会从父RDD中提取出它的父RDD并继续检查，一直到发现持久化过 或者 第一个RDD为止，最后拿到的这个RDD的计算结果后，从前往后一次计算直到产生ResultStage

12. 接下来开始提交所有的Stage，通过submitStage方法提交

13. submitMissingTasks方法进行Task内部划分

14. 内部方法中，会进行Task的划分（ShuffleMapTask和ResultTask），通过模式匹配方式进行匹配，内部实现Task的数量取决于你有多少个分区，我们的ShuffleMapTask是通过匹配ShuffleMapStage进行创建的，当然了，ShuffleMapTask内部也会进行Shuffle的写入操作，这些操作都是发送到Executor端执行的。

15. 如果匹配到了ResultStage的时候，创建ResultTask对象，然后拉取shuffleMap短的数据。

16. 当我们的Task都划分好了，那么将提交所有的Task任务，到TaskSet上面。

17. 此时DAG会调用taskScheduler.submitTasks方法将所有的TaskSet提交至Task Schedule上面。

18. 我们发现Task Schedule内部方法调用了父类方法submitTasks，那么需要找到对应的父类方法。

19. 下面就是开始调度所有的Task任务，那么开始调度之前，实现我们的本地化级别，并且判断Task是否有失败情况，然后进行重试，接下来将所有的Task提交是Run方法中，进行执行所有Task。

20. 找对应的方法recomputeLocality，实现本地化级别划分。

21. computeValidLocalityLevels本地化级别

22. PROCESS\_LOCAL, NODE\_LOCAL, NO\_PREF, RACK\_LOCAL, ANY 五个本地化级别

23. 最后我们通过haslaunchtask提交所有Task任务，然后executor实现LaunchTask进行反序列化，任务正常执行。

1. **Spark Streaming处理kafka产生数据积压问题怎么解决？**

Spark可以通过设置反压机制

sparkConf.set("spark.streaming.backpressure.enabled", "true")

，还可以通过kafka的限流机制来处理数据积压问题，

如果kafka客户端是认证的，那么可以使用userId和clientId两种认证方式。如果没有认证只能使用clientId限流

开启反压的缘由：一个批次的数据应该在一个批次内处理完，即batch process time应该接近于batch Duration，如果batch处理时间总是比batch间隔时间长，就会不断增加调度延迟时间而且数据也会在内存里堆积，进而增加系统不稳定性；另一方面，如果batch处理时间总是远远小于batch间隔时间，则集群资源利用率不高，也是一种资源浪费。

控制批处理时间的关键在batch接收的数据量和业务逻辑处理复杂度，往往前者起了决定性作用，反压机制就可以动态控制batch接收消息速率，来适配集群处理能力。

以下两种场景需要启用反压，可以有效防止应用程序过载：

1、首次启动Streaming应用，kafka保留了大量未消费历史消息，并且auto.offset.reset=latest，可以防止第一个batch接收大量消息、处理时间过长和内存溢出

2、防止kafka producer突然生产大量消息，一个batch接收到大量数据，导致batch之间接收到的数据倾斜

1. **spark Streaming 怎么处理冷热数据join的问题，比如我实时处理的数据需要join一些静态数据，广播太大了，应该怎么处理 ？**

两种解决方案

第一种存储数据库的方式

第二种批流分离方式（也就是解耦，业务线分离，实时接收数据，轻量计算，结果在join）

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

· spark.streaming.concurrentJobs=10：提高Job并发数，读过源码的话会发现，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。

· spark.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。

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

· 在应用级别配置重试  
spark.yarn.maxAppAttempts=5  
spark.yarn.am.attemptFailuresValidityInterval=1h

1. **mapPartition和map执行流程上有什么区别（完善上面的区别）**

1）map的输入变换函数是应用于RDD中每个元素，而mapPartitions的输入函数是应用于每个分区。

假设一个rdd有10个元素，分成3个分区。如果使用map方法，map中的输入函数会被调用10次；而使用mapPartitions

方法的话，其输入函数会只会被调用3次，每个分区调用1次。

2）从输入函数（myfuncPerElement、myfuncPerPartition）层面来看，map是推模式，数据被推到myfuncPerElement

中；mapPartitons是拉模式，myfuncPerPartition通过迭代子从分区中拉数据

3）大数据集情况下的资源初始化开销和批处理处理，如果在myfuncPerPartition和myfuncPerElement中都要初始化一个

耗时的资源，然后使用，比如数据库连接。在上面的例子中，myfuncPerPartition只需初始化3个资源（3个分区每个1次），

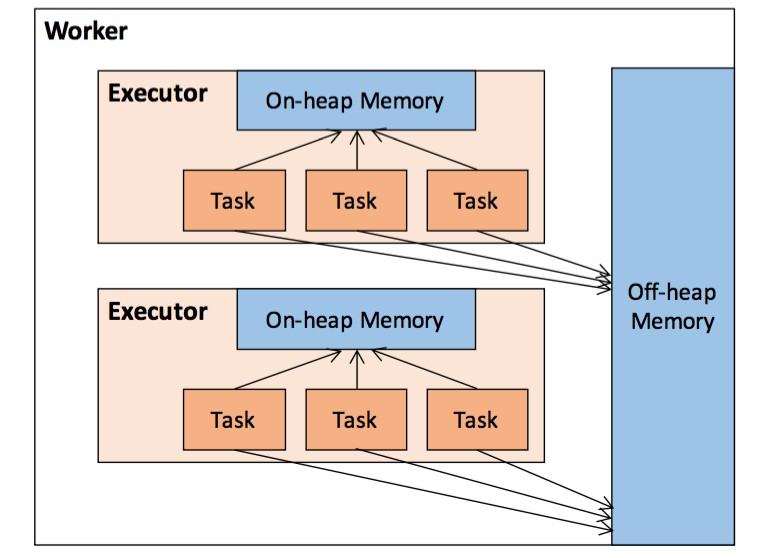
而myfuncPerElement要初始化10次（10个元素每个1次），显然在大数据集情况下（数据集中元素个数远大于分区数），

mapPartitons的开销要小很多，也便于进行批处理操作。

1. **Spark内存管理模型？**

## 堆内和堆外内存规划

作为一个 JVM 进程，Executor 的内存管理建立在 JVM 的内存管理之上，Spark 对 JVM 的堆内（On-heap）空间进行了更为详细的分配，以充分利用内存。同时，Spark 引入了堆外（Off-heap）内存，使之可以直接在工作节点的系统内存中开辟空间，进一步优化了内存的使用.堆内和堆外内存示意图如下：



**堆内内存**

堆内内存的大小，由 Spark 应用程序启动时的 –executor-memory 或 spark.executor.memory 参数配置。Executor 内运行的并发任务共享 JVM 堆内内存，这些任务在缓存 RDD 数据和广播（Broadcast）数据时占用的内存被规划为存储（Storage）内存，而这些任务在执行 Shuffle 时占用的内存被规划为执行（Execution）内存，剩余的部分不做特殊规划，那些 Spark 内部的对象实例，或者用户定义的 Spark 应用程序中的对象实例，均占用剩余的空间。不同的管理模式下，这三部分占用的空间大小各不相同（下面第 2 小节会进行介绍）。

Spark 对堆内内存的管理是一种逻辑上的"规划式"的管理，因为对象实例占用内存的申请和释放都由 JVM 完成，Spark 只能在申请后和释放前**记录**这些内存，我们来看其具体流程：

**申请内存**：

1. Spark 在代码中 new 一个对象实例
2. JVM 从堆内内存分配空间，创建对象并返回对象引用
3. Spark 保存该对象的引用，记录该对象占用的内存

**释放内存**：

1. Spark 记录该对象释放的内存，删除该对象的引用
2. 等待 JVM 的垃圾回收机制释放该对象占用的堆内内存

我们知道，JVM 的对象可以以序列化的方式存储，序列化的过程是将对象转换为二进制字节流，本质上可以理解为将非连续空间的链式存储转化为连续空间或块存储，在访问时则需要进行序列化的逆过程——反序列化，将字节流转化为对象，序列化的方式可以节省存储空间，但增加了存储和读取时候的计算开销。

对于 Spark 中序列化的对象，由于是字节流的形式，其占用的内存大小可直接计算，而对于非序列化的对象，其占用的内存是通过周期性地采样近似估算而得，即并不是每次新增的数据项都会计算一次占用的内存大小，这种方法降低了时间开销但是有可能误差较大，导致某一时刻的实际内存有可能远远超出预期[2]。此外，在被 Spark 标记为释放的对象实例，很有可能在实际上并没有被 JVM 回收，导致实际可用的内存小于 Spark 记录的可用内存。所以 Spark 并不能准确记录实际可用的堆内内存，从而也就无法完全避免内存溢出（OOM, Out of Memory）的异常。

虽然不能精准控制堆内内存的申请和释放，但 Spark 通过对存储内存和执行内存各自独立的规划管理，可以决定是否要在存储内存里缓存新的 RDD，以及是否为新的任务分配执行内存，在一定程度上可以提升内存的利用率，减少异常的出现。

### 堆外内存

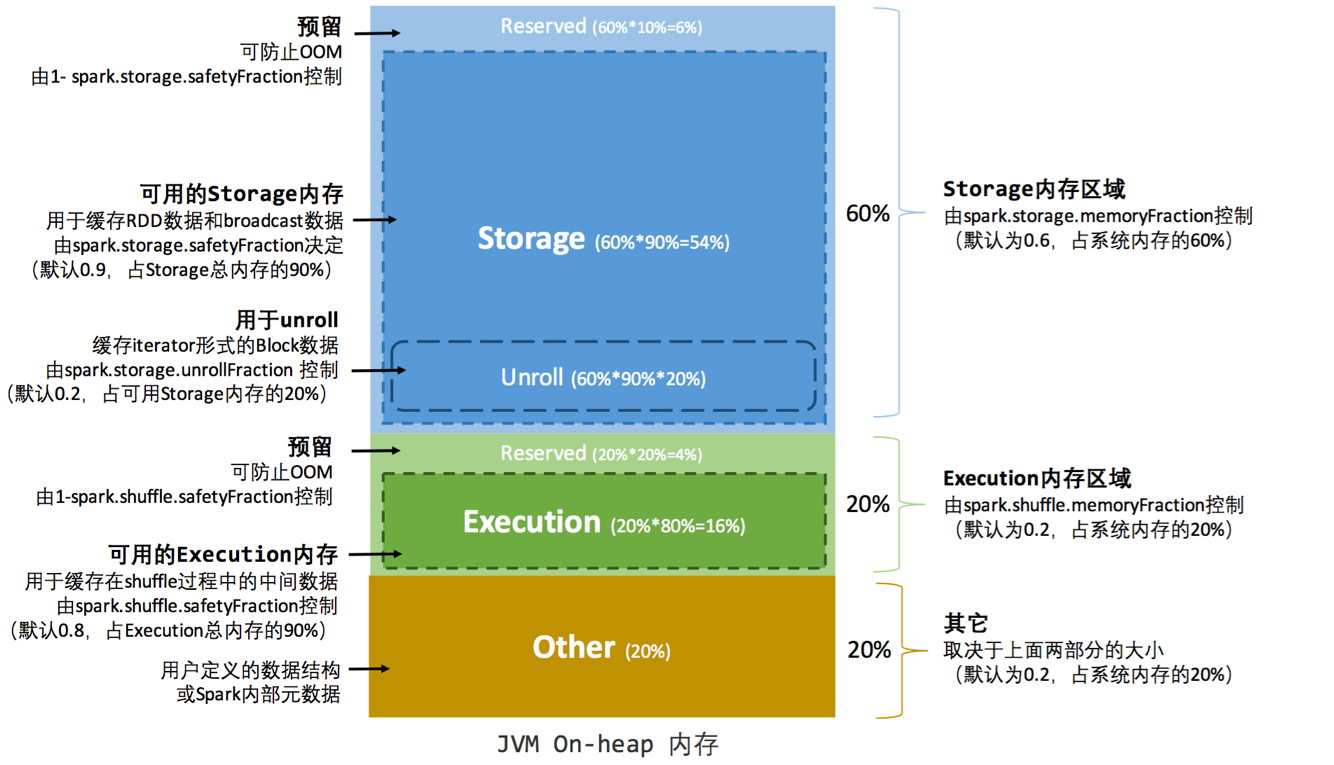
为了进一步优化内存的使用以及提高 Shuffle 时排序的效率，Spark 引入了堆外（Off-heap）内存，使之可以直接在工作节点的系统内存中开辟空间，存储经过序列化的二进制数据。利用 JDK Unsafe API（从 Spark 2.0 开始，在管理堆外的存储内存时不再基于 Tachyon，而是与堆外的执行内存一样，基于 JDK Unsafe API 实现[3]），Spark 可以直接操作系统堆外内存，减少了不必要的内存开销，以及频繁的 GC 扫描和回收，提升了处理性能。堆外内存可以被精确地申请和释放，而且序列化的数据占用的空间可以被精确计算，所以相比堆内内存来说降低了管理的难度，也降低了误差。

在默认情况下堆外内存并不启用，可通过配置 spark.memory.offHeap.enabled 参数启用，并由 spark.memory.offHeap.size 参数设定堆外空间的大小。除了没有 other 空间，堆外内存与堆内内存的划分方式相同，所有运行中的并发任务共享存储内存和执行内存。

### 静态内存管理？

在 Spark 最初采用的静态内存管理机制下，存储内存、执行内存和其他内存的大小在 Spark 应用程序运行期间均为固定的，但用户可以应用程序启动前进行配置，堆内内存的分配如图 2 所示：

**图 2 . 静态内存管理图示——堆内**



可以看到，可用的堆内内存的大小需要按照下面的方式计算：

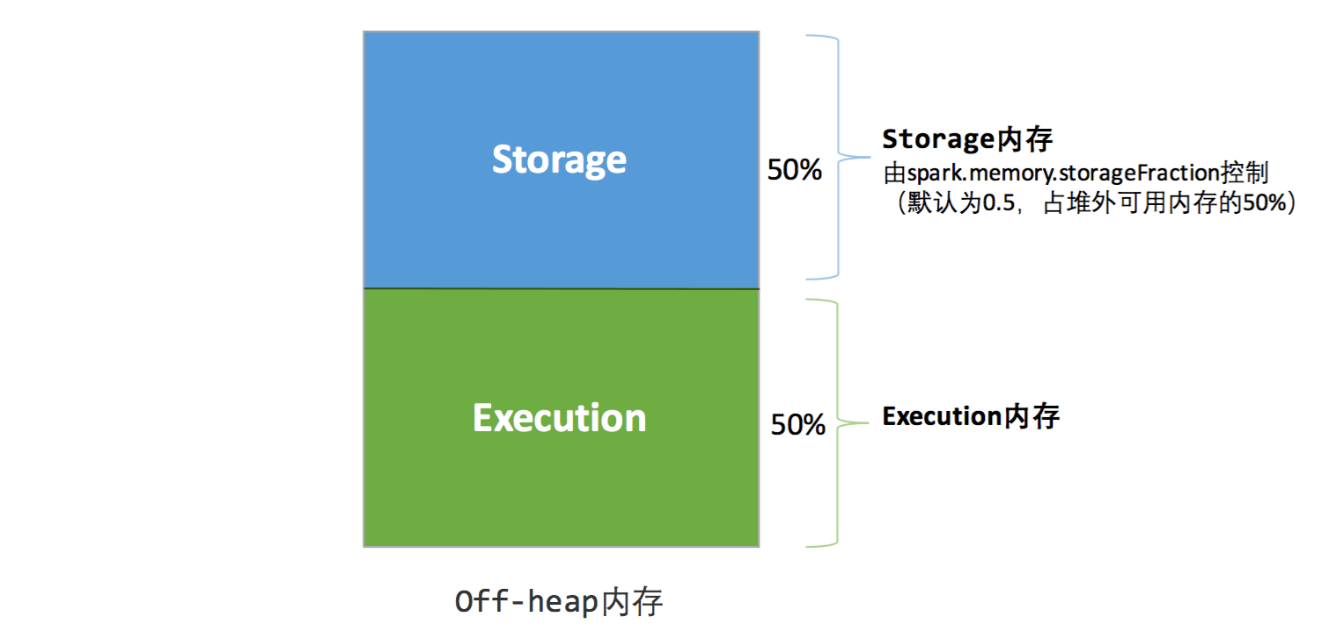
**可用堆内内存空间：**

|  |  |
| --- | --- |
| 1  2 | 可用的存储内存 = systemMaxMemory \* spark.storage.memoryFraction \* spark.storage.safetyFraction  可用的执行内存 = systemMaxMemory \* spark.shuffle.memoryFraction \* spark.shuffle.safetyFraction |

其中 systemMaxMemory 取决于当前 JVM 堆内内存的大小，最后可用的执行内存或者存储内存要在此基础上与各自的 memoryFraction 参数和 safetyFraction 参数相乘得出。上述计算公式中的两个 safetyFraction 参数，其意义在于在逻辑上预留出 1-safetyFraction 这么一块保险区域，降低因实际内存超出当前预设范围而导致 OOM 的风险（上文提到，对于非序列化对象的内存采样估算会产生误差）。值得注意的是，这个预留的保险区域仅仅是一种逻辑上的规划，在具体使用时 Spark 并没有区别对待，和"其它内存"一样交给了 JVM 去管理。

堆外的空间分配较为简单，只有存储内存和执行内存，如图 3 所示。可用的执行内存和存储内存占用的空间大小直接由参数 spark.memory.storageFraction 决定，由于堆外内存占用的空间可以被精确计算，所以无需再设定保险区域。

**图 3 . 静态内存管理图示——堆外**

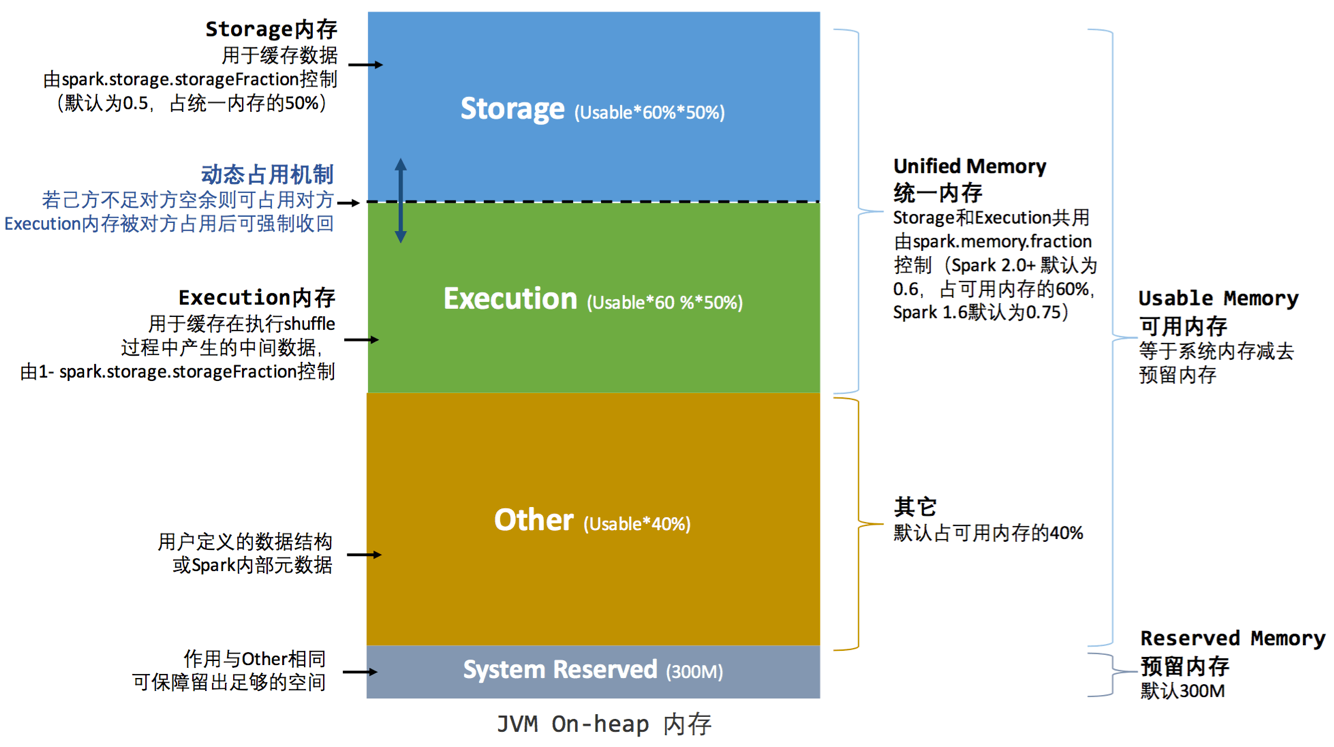


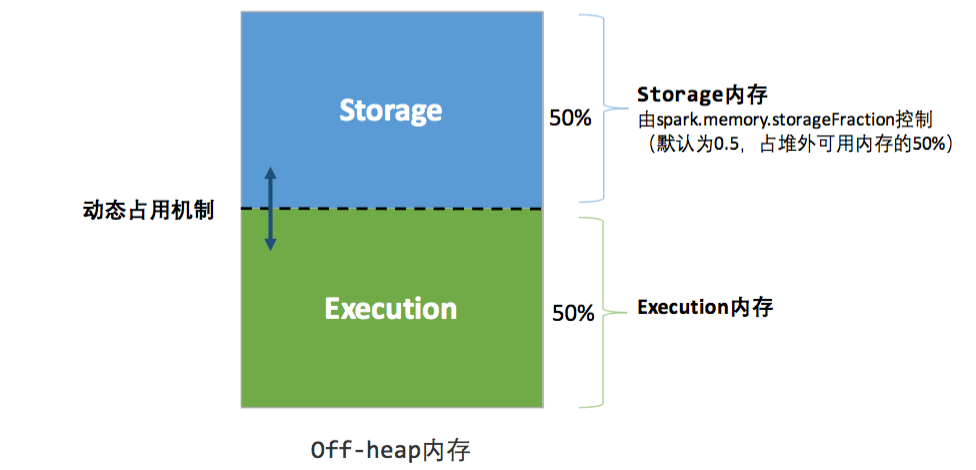
静态内存管理机制实现起来较为简单，但如果用户不熟悉 Spark 的存储机制，或没有根据具体的数据规模和计算任务或做相应的配置，很容易造成"一半海水，一半火焰"的局面，即存储内存和执行内存中的一方剩余大量的空间，而另一方却早早被占满，不得不淘汰或移出旧的内容以存储新的内容。由于新的内存管理机制的出现，这种方式目前已经很少有开发者使用，出于兼容旧版本的应用程序的目的，Spark 仍然保留了它的实现。

### 统一内存管理？

Spark 1.6 之后引入的统一内存管理机制，与静态内存管理的区别在于存储内存和执行内存共享同一块空间，可以动态占用对方的空闲区域，如图 4 和图 5 所

**图 4 . 统一内存管理图示——堆内**

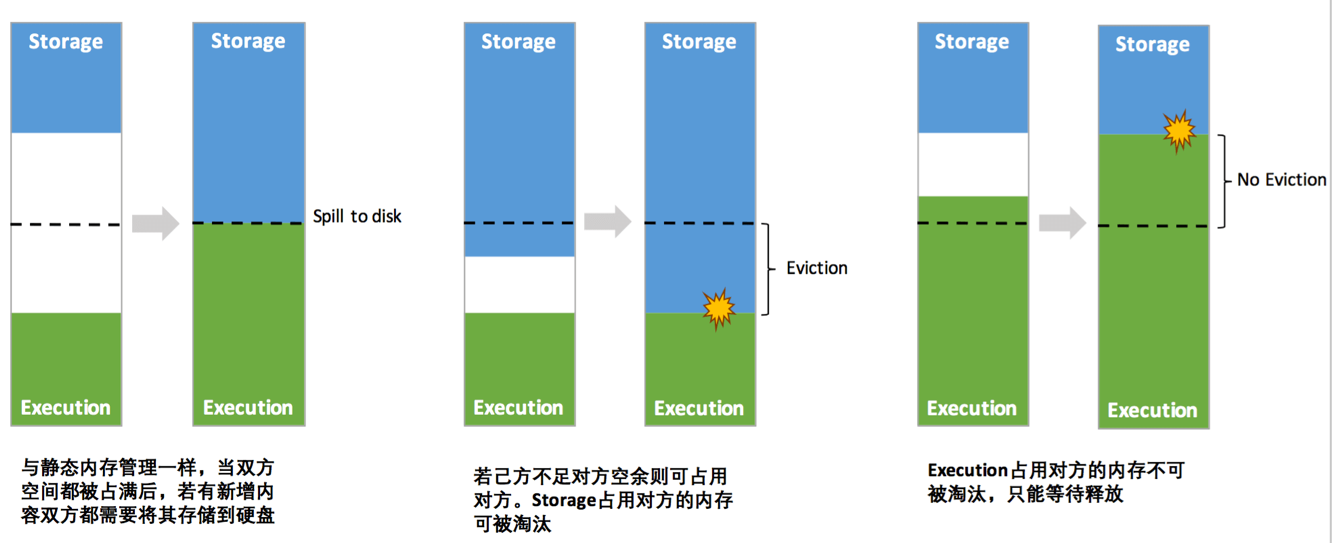
**图 5 . 统一内存管理图示——堆外**



其中最重要的优化在于动态占用机制，其规则如下：

1. 设定基本的存储内存和执行内存区域（spark.storage.storageFraction 参数），该设定确定了双方各自拥有的空间的范围
2. 双方的空间都不足时，则存储到硬盘；若己方空间不足而对方空余时，可借用对方的空间;（存储空间不足是指不足以放下一个完整的 Block）
3. 执行内存的空间被对方占用后，可让对方将占用的部分转存到硬盘，然后"归还"借用的空间
4. 存储内存的空间被对方占用后，无法让对方"归还"，因为需要考虑 Shuffle 过程中的很多因素，实现起来较为复杂[4

图 6 . 动态占用机制图示



凭借统一内存管理机制，Spark 在一定程度上提高了堆内和堆外内存资源的利用率，降低了开发者维护 Spark 内存的难度，但并不意味着开发者可以高枕无忧。譬如，所以如果存储内存的空间太大或者说缓存的数据过多，反而会导致频繁的全量垃圾回收，降低任务执行时的性能，因为缓存的 RDD 数据通常都是长期驻留内存的 [5] 。所以要想充分发挥 Spark 的性能，需要开发者进一步了解存储内存和执行内存各自的管理方式和实现原理。

1. **Spark 2.x与 1.6版本的区别**
2. Spark Core/SQL 在内存和CPU使用方面进一步优化Spark引擎性能，支持SQL 2003标准，支持子查询，对常用的SQL操作和DataFrame,性能有2-10倍的提升
3. sparksession Spark2.0 中引入了 SparkSession 的概念，它为用户提供了一个统一的切入点来使用 Spark 的各项功能，统一了旧的SQLContext与HiveContext。用户不但可以使用 DataFrame 和Dataset 的各种 API，学习 Spark2 的难度也会大大降低。
4. 统一 DataFrames 和 Datasets 的 API 它们都是提供给用户使用，包括各类操作接口的 API，1.3 版本引入 DataFrame，1.6版本引入Dataset，在 spark 2.0 中，把 dataframes 当作是一种特殊的 datasets，dataframes = datasets[row]，把两者统一为datasets。
5. **Hive 和 Spark SQL的区别？**
6. SparkSQL和Hive On Spark都是在Spark上实现SQL的解决方案。Spark早先有Shark项目用来实现SQL层，不过后来推翻重做了，就变成了SparkSQL。这是Spark官方Databricks的项目，Spark项目本身主推的SQL实现。Hive On Spark比SparkSQL稍晚。Hive原本是没有很好支持MapReduce之外的引擎的，而Hive On Tez项目让Hive得以支持和Spark近似的Planning结构（非MapReduce的DAG）。所以在此基础上，Cloudera主导启动了Hive On Spark。
7. 历史上存在的原理，以前都是使用hive来构建数据仓库，所以存在大量对hive所管理的数据查询的需求。而hive、shark、sparlSQL都可以进行hive的数据查询。shark是使用了hive的sql语法解析器和优化器，修改了执行器，使之物理执行过程是跑在spark上；而sparkSQL是使用了自身的语法解析器、优化器和执行器，同时sparkSQL还扩展了接口，不单单支持hive数据的查询，可以进行多种数据源的数据查询。
8. **Kafka如何动态扩容**

1.在新的物理机上安装kafka程序

2.修改config/server.properties文件里的broker.id必须在集群中唯一，修改其他必要的配置项，其中zookeeper.connect配置项，写上kafka集群现在使用的zookeeper集群的地址。

3.启动kafka就可以加入到集群中了。

注意

新加入的机器只能对新产生的topic起作用，对已有的topic在没有做处理前，是不会承担任何任务的，所以不会分担集群的压力

1. **如何保证kafka数据不丢失不重复**

|  |
| --- |
| kafka动态维护了一个同步状态的副本的集合（a set of In-Sync Replicas），简称ISR。  在这个集合中的节点都是和leader保持高度一致的，任何一条消息只有被这个集合中的每个节点读取并追加到日志中，才会向外部通知说“这个消息已经被提交”。  只有当消息被所有的副本加入到日志中时，才算是“committed”，只有committed的消息才会发送给consumer，这样就不用担心一旦leader down掉了消息会丢失。  消息从leader复制到follower, 我们可以通过决定producer是否等待消息被提交的通知(ack)来区分同步复制和异步复制。 同步发送： 发出消息后，必须阻塞等待收到通知后，才发送下一条消息。  特点：同步发送模式虽然吞吐量小，但是发一条收到确认后再发下一条，既能保证不丢失消息，又能保证顺序。 异步发送： 一直往缓冲区写，然后一把写到队列中去。  特点：异步的可能会丢失数据。 |

|  |
| --- |
| 如何保证有序： 如果有一个发送失败了，后面的就不能继续发了，不然重发的那个肯定乱序了。  生产者在收到发送成功的反馈之前，不能发下一条数据，但我感觉生产者是一个流，阻塞生产者感觉业务上不可行，怎么会因为一条消息发出去没收到反馈，就阻塞生产者。 |

|  |
| --- |
| 解决策略： 1.异步方式缓冲区满了，就阻塞在那，等着缓冲区可用，不能清空缓冲区。  2.发送消息之后回调函数，发送成功就发送下一条，发送失败就记在日志中，等着定时脚本来扫描。  注意：发送失败可能并不真的发送失败，只是没收到反馈，定时脚本可能会重发。 |

|  |
| --- |
| ack： ack确认机制设置为0，表示不等待响应，不等待borker的确认信息，最小延迟，producer无法知道消息是否发生成功，消息可能丢失，但具有最大吞吐量。  ack确认机制设置为-1，也就是让消息写入leader和所有的副本，ISR列表中的所有replica都返回确认消息。  ack确认机制设置为1，leader已经接收了数据的确认信息，replica异步拉取消息，比较折中。  ack确认机制设置为2，表示producer写partition leader和其他一个follower成功的时候，broker就返回成功，无论其他的partition follower是否写成功。  ack确认机制设置为 "all" 即所有副本都同步到数据时send方法才返回, 以此来完全判断数据是否发送成功, 理论上来讲数据不会丢失。    min.insync.replicas=1  意思是至少有1个replica返回成功，否则product异常 |

====Kafka消息保证生产的信息不丢失和重复消费问题===================

### Kafka到底会不会丢数据(data loss)?

通常不会，但有些情况下的确有可能会发生。  
下面的参数配置及Best practice列表可以较好地保证数据的持久性(当然是trade-off 即权衡利弊，牺牲了吞吐量)。  
笔者会在该列表之后对列表中的每一项进行讨论，有兴趣的同学可以看下后面的分析。

没有银弹（彻底消除软件危机，带来软件研发的变革的神奇武器）。

如果想要高吞吐量就要能容忍偶尔的失败（比如重发漏发无顺序保证）。

|  |
| --- |
| block.on.buffer.full = true **acks = all** retries = MAX\_VALUE max.in.flight.requests.per.connection = 1 使用KafkaProducer.send(record, callback) callback逻辑中显式关闭producer：close(0)  unclean.leader.election.enable=false **replication.factor = 3  min.insync.replicas = 2** **replication.factor > min.insync.replicas** **enable.auto.commit=false**  消息处理完成之后再提交位移 |

### 

|  |
| --- |
| ****产生数据丢失的两种情况：**** 1）同步模式：有3种状态保证消息被安全生产，但是在配置为1（只保证写入leader成功）的话，如果刚好leader partition挂了，数据就会丢失。 2）异步模式：当缓冲区满了，如果配置为0（还没有收到确认的情况下，缓冲池一满，就清空缓冲池里的消息），数据就会被立即丢弃掉。 |

### 

|  |
| --- |
| ****在数据生产时避免数据丢失的方法：**** 1）同步模式：确认机制设置为-1，也就是让消息写入leader和所有的副本。 2）异步模式：如果消息发出去了，但还没有收到确认的时候，缓冲池满了，在配置文件中设置成不限制阻塞超时的时间，也就说让生产端一直阻塞，这样也能保证数据不会丢失。 在数据消费时，避免数据丢失的方法：如果使用了storm，要开启storm的ackfail机制。  如果没有使用storm，确认数据被完成处理之后，再更新offset值。低级API中需要手动控制offset值。 |

|  |
| --- |
| 消息队列的问题都要从源头找问题，就是生产者是否有问题。  讨论一种情况：  如果数据发送成功，但是接受response的时候丢失了，机器重启之后就会重发。    **重发 解决：**  消费端增加去重表就能解决，但是如果生产者丢失了数据，问题就很麻烦了。  **重复消费 解决：** （1）去重：将消息的唯一标识保存到外部介质中，每次消费处理时判断是否处理过。 （2）不管：大数据场景中，报表系统或者日志信息丢失几条都无所谓，不会影响最终的统计分析结果。 |

给出列表之后，我们从两个方面来探讨一下数据为什么会丢失：  
  
  
1. Producer端

|  |  |
| --- | --- |
| 问题1：丢失消息  目前比较新版本的Kafka正式替换了Scala版本的old producer，使用了由Java重写的producer。新版本的producer采用异步发送机制。          KafkaProducer.send(ProducerRecord)方法仅仅是把这条消息放入一个缓存中(即RecordAccumulator，本质上使用了队列来缓存记录)，同时后台的IO线程会不断扫描该缓存区，将满足条件的消息封装到某个batch中然后发送出去。显然，这个过程中就有一个数据丢失的窗口：若IO线程发送之前client端挂掉了，累积在accumulator中的数据的确有可能会丢失。  问题2：消息乱序    假设客户端代码依次执行下面的语句将两条消息发到相同的分区 producer.send(record1); producer.send(record2);  如果此时由于某些原因(比如瞬时的网络抖动)导致record1没有成功发送。  同时Kafka又配置了重试机制和max.in.flight.requests.per.connection大于1(默认值是5，本来就是大于1的)。  那么重试record1成功后，record1在分区中就在record2之后，从而造成消息的乱序。  很多某些要求强顺序保证的场景是不允许出现这种情况的。  发送之后重发就会丢失顺序。 **解决：**  **鉴于producer的这两个问题，我们应该如何规避呢？？**    对于消息丢失的问题，很容易想到的一个方案就是：  既然异步发送有可能丢失数据， 我改成同步发送总可以。比如这样：producer.send(record).get(); 这样当然是可以的，但是性能会很差，不建议这样使用。    **因此特意总结了一份配置列表：**  个人认为该配置清单应该能够比较好地规避producer端数据丢失情况的发生。  特此说明一下：软件配置的很多决策都是trade-off（平衡，权衡，按实际情况权衡一下，得到最终决策）。  下面的配置也不例外： 应用了这些配置，你可能会发现你的producer/consumer 吞吐量会下降，这是正常的，因为你换取了更高的数据安全性。   |  | | --- | | block.on.buffer.full = true  尽管该参数在0.9.0.0已经被标记为“deprecated”（不赞成，反对的意思），但鉴于它的含义非常直观。  所以这里还是显式设置它为true，使得producer将一直等待缓冲区直至其变为可用。否则如果producer生产速度过快耗尽了缓冲区，producer将抛出异常。缓冲区满了就阻塞在那，不要抛异常，也不要丢失数据。  acks=all  很好理解，所有follower都响应了才认为消息提交成功，即"committed"。  retries = MAX  无限重试，直到你意识到出现了问题  max.in.flight.requests.per.connection = 1  限制客户端在单个连接上能够发送的未响应请求的个数。设置此值是1表示kafka broker在响应请求之前client不能再向同一个broker发送请求。注意：设置此参数是为了避免消息乱序 使用KafkaProducer.send(record, callback)而不是send(record)方法   自定义回调逻辑处理消息发送失败，比如记录在日志中，用定时脚本扫描重处理。  callback  逻辑中最好显式关闭producer：close(0)  注意：设置此参数是为了避免消息乱序（仅仅因为一条消息发送没收到反馈就关闭生产者，感觉代价很大）。  unclean.leader.election.enable=false  关闭unclean leader选举，即不允许非ISR中的副本被选举为leader，以避免数据丢失。  replication.factor >= 3  这个完全是个人建议了，参考了Hadoop及业界通用的三备份原则  min.insync.replicas > 1  消息至少要被写入到这么多副本才算成功，也是提升数据持久性的一个参数。与acks配合使用。  保证  replication.factor > min.insync.replicas  如果两者相等，当一个副本挂掉了分区也就没法正常工作了。  通常设置  replication.factor = min.insync.replicas + 1  即可。 | |

### 2. Consumer端

|  |
| --- |
| consumer端丢失消息的情形比较简单：  如果在消息处理完成前就提交了offset，那么就有可能造成数据的丢失。  由于Kafka consumer默认是自动提交位移的，所以在后台提交位移前一定要保证消息被正常处理了，因此不建议采用很重的处理逻辑，如果处理耗时很长，则建议把逻辑放到另一个线程中去做。  为了避免数据丢失，现给出两点建议： enable.auto.commit=false  关闭自动提交位移，在消息被完整处理之后再手动提交位移。 |

### 总结1：

|  |
| --- |
| 消息的完整性和系统的吞吐量是互斥的，为了确保消息不丢失就必然会损失系统的吞吐量  producer：  1、ack设置-1  2、设置副本同步成功的最小同步个数为副本数-1  3、加大重试次数  4、同步发送  5、对于单条数据过大，要设置可接收的单条数据的大小  6、对于异步发送，通过回调函数来感知丢消息  7、配置不允许非ISR集合中的副本当leader  8、客户端缓冲区满了也可能会丢消息；或者异步情况下消息在客户端缓冲区还未发送，客户端就宕机  9、block.on.buffer.full = true  consumer：  1、enable.auto.commit=false  关闭自动提交位移      同一分区消息乱序：  假设a,b两条消息，a先发送后由于发送失败重试，这时顺序就会在b的消息后面，可以设置max.in.flight.requests.per.connection=1来避免    max.in.flight.requests.per.connection：限制客户端在单个连接上能够发送的未响应请求的个数。设置此值是1表示kafka broker在响应请求之前client不能再向同一个broker发送请求，但吞吐量会下降 |

### 总结2：

|  |
| --- |
| 生产者如何保证数据的不丢失  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。   ****同步模式：**** ack机制能够保证数据的不丢失，如果ack设置为0，风险很大，一般不建议设置为0  producer.type=sync  request.required.acks=1  //只leader    # 当所有的follower都同步消息成功后发送ack  request.required.acks=-1   ****异步模式：**** 通过buffer来进行控制数据的发送，有两个值来进行控制，时间阈值与消息的数量阈值，如果buffer满了数据还没有发送出去，如果设置的是立即清理模式，风险很大，一定要设置为阻塞模式。    结论：  **生产者如何保证数据的不丢失**：  producer有丢数据的可能，但是可以通过配置保证消息的不丢失。  producer.type=async  异步模式  request.required.acks=1  需要ack校验机制  queue.buffering.max.ms=5000  队列最大缓存时间大一些5秒  queue.buffering.max.messages=10000  队列最大缓存消息数量大些10000  queue.enqueue.timeout.ms = -1  进入队列超时时间设置为永不超时  batch.num.messages=200 每一批处理消息的数量设置小些200  **消费者如何保证数据的不丢失**  设置不自动提交，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。 |

1. **redis的缓存有哪些问题？一致性？击穿？雪崩等是如何解决的？**

什么叫击穿？

请求数据顺序，首先会从redis缓存中拿数据，若未拿到则查数据库，再写到redis。如果是请求一条根本不存在的数据时，则会去直接访问数据库，但是数据库也没有，所以它也没把数据写入redis缓存。所以每次这种请求都会直接访问数据库。如果请求的数量太大的话，这种绕过redis直接访问数据库情况则成为击穿。

解决办法：

1、在查询数据库的时候若未查询到数据，依然在redis中保存一条记录，并把value设置为null或者其他便于区别的值，可以再给该条数据设置一个过期时间，那么以后再有请求过来访问时就不会绕过redis直接去查库。

2、在从redis中获取数据前增加查询校验，对所有可能存在的查询参数缓存到hash集合中，若某个请求在该集合中存在，则可以访问rides，若不存在则直接丢弃该请求。

什么叫雪崩？

每个key（即数据）如果设置了失效时间的话，如果大量key同时过期的时候，或者说因为某种原因redis中的数据突然大批量丢失，这些key又大量地去请求这些key时，因为redis里面没有这些数据，就会大量的请求就会大量涌向数据库，就会导致数据库处理不过来，导致“雪崩”。

解决办法：

1、设置均匀的过期时间，避免某段时间出现大批量数据失效

2、在缓存失效后，通过加锁或者队列来控制读数据库写缓存的线程数量。比如对某个key只允许一个线程查询数据和写缓存，其他线程等待。

3、做二级缓存，一级缓存未查询到数据，则从二级缓存查询

**89. Kafka的分区策略**

#### 1. Range  Startegy（根据范围消费）

**range策略针对于每个topic，各个topic之间分配时没有任何关联**，分配步骤如下：

1. topic下的所有有效分区平铺，例如P0, P1, P2, P3... ...
2. 消费者按照字典排序，例如C0, C1, C2
3. 分区数除以消费者数，得到n
4. 分区数对消费者数取余，得到m
5. 消费者集合中，前m个消费者能够分配到n+1个分区，而剩余的消费者只能分配到n个分区。

所以对于某个topic来说：  
如果有5个分区（P0, P1, P2, P3, P4），且订阅这个topic的消费者组有2个消费者（C0, C1）。那么P0, P1, P2将被C0消费，P3, P4将被C1消费。  
如果有4个分区（P0, P1, P2, P3），且订阅这个topic的消费者组有2个消费者（C0, C1）。那么P0, P1将被C0消费，P2, P3将被C1消费。

#### 2. RoundRobin strategy（轮询的消费策略）

**roundrobin策略针对于全局所有的topic和消费者**，分配步骤如下：

1. 消费者按照字典排序，例如C0, C1, C2... ...，并构造环形迭代器。
2. topic名称按照字典排序，并得到每个topic的所有分区，从而得到所有分区集合。
3. 遍历第2步所有分区集合，同时轮询消费者。
4. 如果轮询到的消费者订阅的topic不包括当前遍历的分区所属topic，则跳过；否则分配给当前消费者，并继续第3步。

所以对于某个topic来说：  
如果有5个分区（P0, P1, P2, P3, P4），且订阅这个topic的消费者组有2个消费者（C0, C1）。那么P0, P2, P4将被C0消费，P1, P3将被C1消费。

**90.Elasticsearch中的两种查询操作query和filter区别**

**ES中的查询操作分为2种：查询（query）和过滤（filter）。**

**查询即是之前提到的query查询，它（查询）默认会计算每个返回文档的得分，然后根据得分排序。**

**而过滤（filter）只会筛选出符合的文档，并不计算得分，且它可以缓存文档。所以，单从性能考虑，过滤比查询更快。**

**所以我们在组合使用这两种查询操作的时候，我们优先考虑先过滤后查询的组合。**

**就是说：过滤适合在大范围筛选数据，而查询则适合精确匹配数据。所以，一般应用时，应先使用过滤操作过滤数据，然后使用查询匹配数据。**

# Kafka

## 1.1 (//TODO)类似问题： flume中向kafka拉取 数据 数据丢失怎么办

## 1.2 (//TODO)实时流式计算框架,几个人,多长时间,细节问题,包括讲flume ,kafka ,storm 的各个的组件组成,你负责那一块,如果需要你搭建你可以完成么?

## 1.3(//TODO)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）

用下面命令可以查询到topic:DynamicRange broker:SparkMaster:9092的offset的最小值：  
bin/IMG_256kafka-run-class.sh kafka.tools.GetOffsetShell --broker-list slave6:9092 -topic videoplay --time -2  
输出  
DynamicRange:0:1288  
查询offset的最大值：  
bin/IMG_257kafka-run-class.sh kafka.tools.GetOffsetShell --broker-list slave6:9092 -topic videoplay --time -1  
输出  
DynamicRange:0:7885  
从上面的输出可以看出topic:DynamicRange只有一个partition:0 offset范围为:[1288,7885]

## 1.4(★★)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

## 1.5 (★★)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

## 1.6(★)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。

## 1.7(★★)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

类似问题：介绍一下kafka kafka的API写过吗？

## 1.8(★★)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

## 1.9(★★★)kafka+spark-streaming结合丢数据怎么解决？

这篇文章:http://www.csdn.net/article/2015-06-21/2825011

http://group.jobbole.com/15559/

spark streaming从1.2开始提供了数据的零丢失，想享受这个特性，需要满足如下条件：

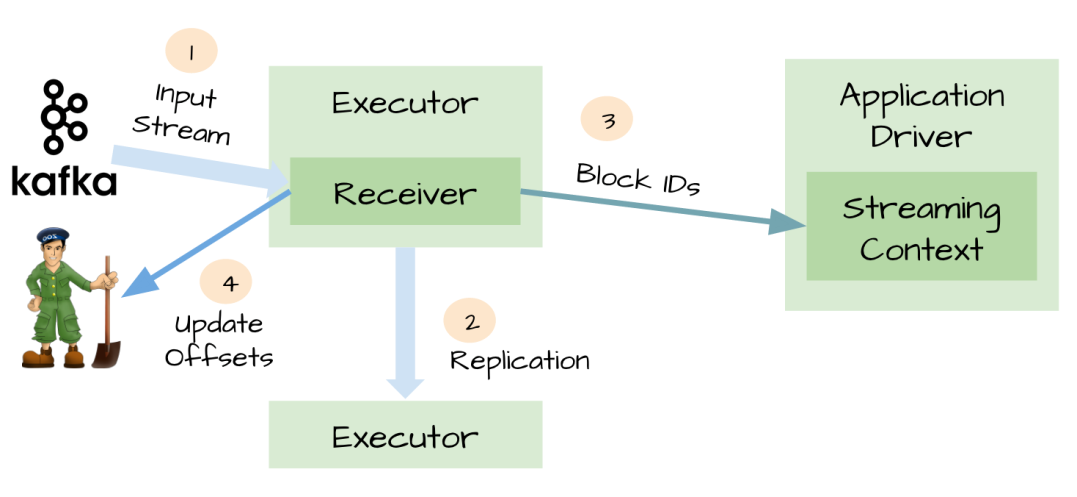
数据输入需要可靠的sources和可靠的receivers

应用metadata必须通过应用driver checkpoint

WAL（write ahead log）

可靠的sources和receivers

spark streaming可以通过多种方式作为数据sources（包括kafka），输入数据通过receivers接收，通过replication存储于spark中（为了faultolerance，默认复制到两个spark executors），如果数据复制完成，receivers可以知道（例如kafka中更新offsets到zookeeper中）。这样当receivers在接收数据过程中crash掉，不会有数据丢失，receivers没有复制的数据，当receiver恢复后重新接收。

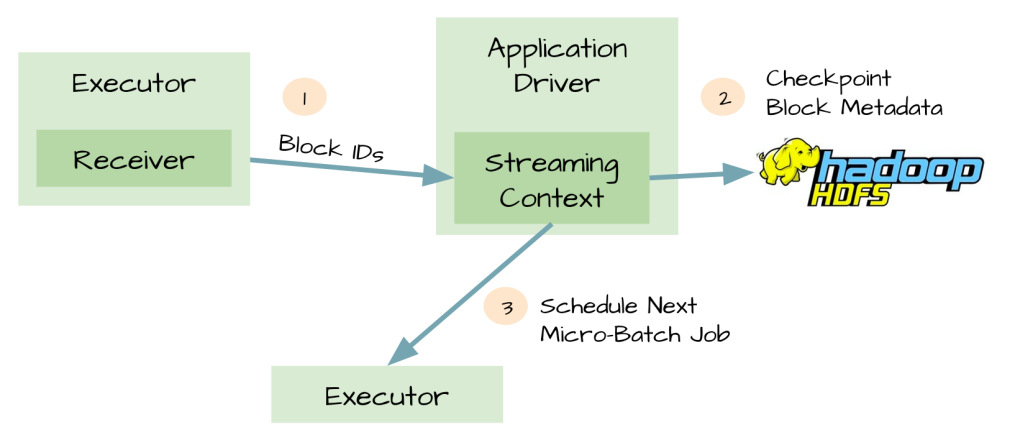
[](https://raw.githubusercontent.com/jacksu/utils4s/master/spark-knowledge/images/spark-streaming-kafka/spark-reliable-source-reliable-receiver.png)

metadata checkpoint

可靠的sources和receivers，可以使数据在receivers失败后恢复，然而在driver失败后恢复是比较复杂的，一种方法是通过checkpoint metadata到HDFS或者S3。metadata包括：

configuration

code

一些排队等待处理但没有完成的RDD（仅仅是metadata，而不是data）  
[](https://raw.githubusercontent.com/jacksu/utils4s/master/spark-knowledge/images/spark-streaming-kafka/spark-metadata-checkpointing.png)

这样当driver失败时，可以通过metadata checkpoint，重构应用程序并知道执行到那个地方。

数据可能丢失的场景

可靠的sources和receivers，以及metadata checkpoint也不可以保证数据的不丢失，例如：

两个executor得到计算数据，并保存在他们的内存中

receivers知道数据已经输入

executors开始计算数据

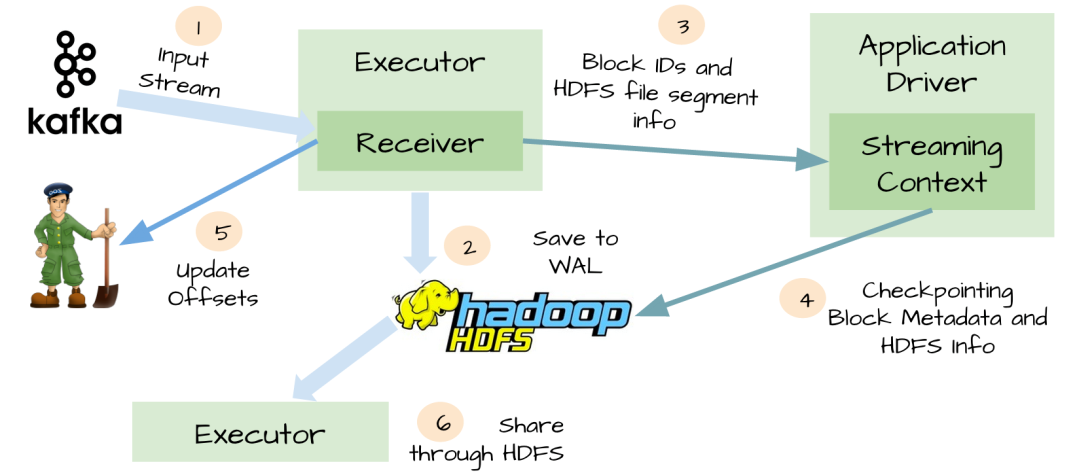
driver突然失败

driver失败，那么executors都会被kill掉

因为executor被kill掉，那么他们内存中得数据都会丢失，但是这些数据不再被处理

executor中的数据不可恢复

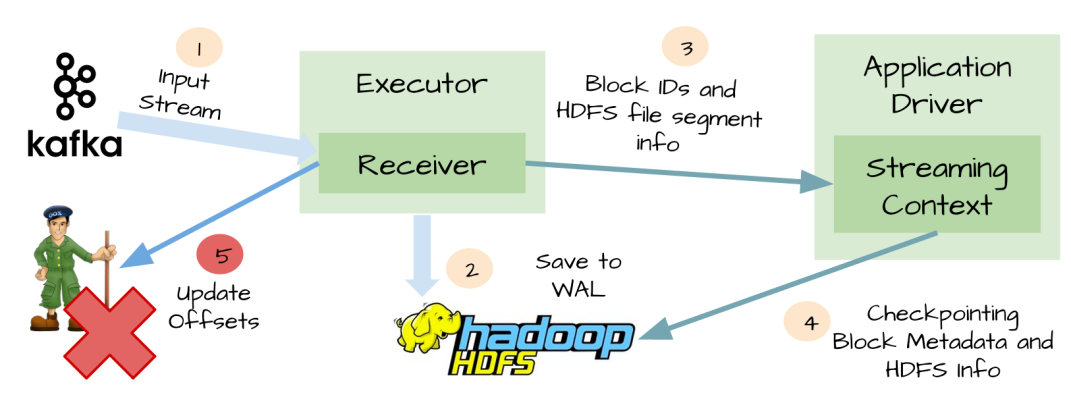
WAL

为了避免上面情景的出现，spark streaming 1.2引入了WAL。所有接收的数据通过receivers写入HDFS或者S3中checkpoint目录，这样当driver失败后，executor中数据丢失后，可以通过checkpoint恢复。  
[](https://raw.githubusercontent.com/jacksu/utils4s/master/spark-knowledge/images/spark-streaming-kafka/spark-wal.png)

At-Least-Once

尽管WAL可以保证数据零丢失，但是不能保证exactly-once，例如下面场景：

Receivers接收完数据并保存到HDFS或S3

在更新offset前，receivers失败了  
[](https://raw.githubusercontent.com/jacksu/utils4s/master/spark-knowledge/images/spark-streaming-kafka/spark-wall-at-least-once-delivery.png)

Spark Streaming以为数据接收成功，但是Kafka以为数据没有接收成功，因为offset没有更新到zookeeper

随后receiver恢复了

从WAL可以读取的数据重新消费一次，因为使用的kafka High-Level消费API，从zookeeper中保存的offsets开始消费

WAL的缺点

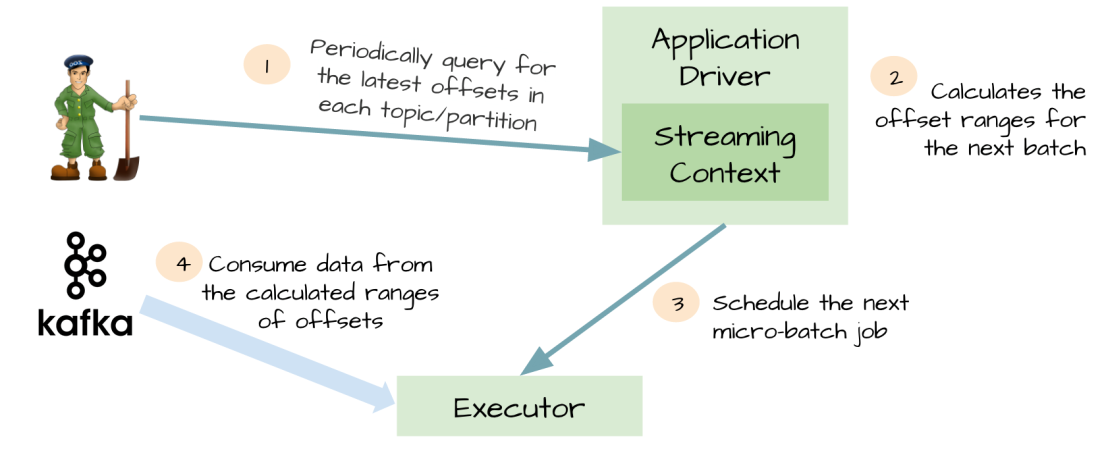
通过上面描述，WAL有两个缺点：

降低了receivers的性能，因为数据还要存储到HDFS等分布式文件系统

对于一些resources，可能存在重复的数据，比如Kafka，在Kafka中存在一份数据，在Spark Streaming也存在一份（以WAL的形式存储在Hadoop API兼容的文件系统中）

Kafka direct API

为了WAL的性能损失和exactly-once，spark streaming1.3中使用Kafka direct API。非常巧妙，Spark driver计算下个batch的offsets，指导executor消费对应的topics和partitions。消费Kafka消息，就像消费文件系统文件一样。

[](https://raw.githubusercontent.com/jacksu/utils4s/master/spark-knowledge/images/spark-streaming-kafka/spark-kafka-direct-api.png)

不再需要kafka receivers，executor直接通过Kafka API消费数据

WAL不再需要，如果从失败恢复，可以重新消费

exactly-once得到了保证，不会再从WAL中重复读取数据

总结

主要说的是spark streaming通过各种方式来保证数据不丢失，并保证exactly-once，每个版本都是spark streaming越来越稳定，越来越向生产环境使用发展。

## 1.10(★★)kafka中存储目录data/dir.....topic1和topic2怎么存储的，存储结构，data.....目录下有多少个分区，每个分区的存储格式是什么样的？

1、topic是按照“主题名-分区”存储的

2、分区个数由配置文件决定

3、每个分区下最重要的两个文件是0000000000.log和000000.index，0000000.log以默认1G大小回滚。

## 1.11(★)Kafka怎么丢失数据的?

原因1：强行kill线程，导致消费后的数据，offset没有提交。

原因2：设置offset为自动提交，关闭kafka时，如果在close之前，调用 consumer.unsubscribe() 则有可能部分offset没提交，下次重启会重复消费。

原因3（重复消费最常见的原因）：消费后的数据，当offset还没有提交时，partition就断开连接。比如，通常会遇到消费的数据，处理很耗时，导致超过了Kafka的session timeout时间（0.10.x版本默认是30秒），那么就会re-blance重平衡，此时有一定几率offset没提交，会导致重平衡后重复消费。

记录offset和恢复offset的方案

offset记录方案：

每次消费时更新每个topic+partition位置的offset在内存中，

Map<key, value>，key=topic+’-‘+partition，value=offset

当调用关闭consumer线程时，把上面Map的offset数据记录到 文件中\*（分布式集群可能要记录到redis中）。

下一次启动consumer，需要读取上一次的offset信息，方法是 以当前的topic+partition为key，从上次的Map中去寻找offset。

然后使用consumer.seek()方法指定到上次的offset位置

## 1.12(★★)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）

## 1.13(★★)kafka的partition影响spark的并发度如何解决？

可以在处理数据之前通过 repartition 或 coalease 对数据进行重分区,这种方法的好处是，对同一类型的数据，先后顺序是不会乱的，因为同一类型的数据经过重分区最后还是会分发到同一个分区里面的。

但是这个方法的使用前提是数据重分区+后续处理的时间比没有重分区直接处理数据的时间要短，否则重分区的开销过大导致总的处理时间过长那就没意义了。

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

限制每个批次读取进来的最大数据量，来平稳的读取积压的数据

--conf spark.streaming.kafka.maxRatePerPartition=10000

## 1.15(★★)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。

## 1.16(★★★)kakfadirect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。

## 1.17(★)kafka收集数据的原理？

Kafka存储策略

1）kafka以topic来进行消息管理，每个topic包含多个partition，每个partition对应一个逻辑log，有多个segment组成。

2）每个segment中存储多条消息（见下图），消息id由其逻辑位置决定，即从消息id可直接定位到消息的存储位置，避免id到位置的额外映射。

3）每个part在内存中对应一个index，记录每个segment中的第一条消息偏移。

4）发布者发到某个topic的消息会被均匀的分布到多个partition上（或根据用户指定的路由规则进行分布），broker收到发布消息往对应partition的最后一个segment上添加该消息，当某个segment上的消息条数达到配置值或消息发布时间超过阈值时，segment上的消息会被flush到磁盘，只有flush到磁盘上的消息订阅者才能订阅到，segment达到一定的大小后将不会再往该segment写数据，broker会创建新的segment。

## 1.18 ★kafka用的什么模式,区别

生产消费者模式与发布订阅模式

生产消费者模式，指的是由生产者将数据源源不断推送到消息中心，由不同的消费者从消息中心取出数据做自己的处理，在同一类别下，所有消费者拿到的都是同样的数据；

订阅发布模式，本质上也是一种生产消费者模式，不同的是，由订阅者首先向消息中心指定自己对哪些数据感兴趣，发布者推送的数据经过消息中心后，每个订阅者拿到的仅仅是自己感兴趣的一组数据。

这两种模式是使用消息中间件时最常用的，用于功能解耦和分布式系统间的消息通信。

## 1.19(★)kafka在消费者消费的数据格式有哪些?

.index文件、.log文件、.timeindex文件index件和log文件组成segment，segment文件的命名规则是，partion全局的第一个segment从0开始，后续每个segment文件名为上一个全局partion的最大offset(偏移message数)。数值最大为64位long大小，19位数字字符长度，没有数字用0填充。log.segment.bytes参数配置了一个log文件的大小，文件大小超过这个值就会生成新的文件

## 1.20(★)kafka有了解过吗为什么他的吞吐量大,速度快

kafka主要使用了以下几个方式实现了超高的吞吐率

顺序读写

kafka的消息是不断追加到文件中的，这个特性使kafka可以充分利用磁盘的顺序读写性能

顺序读写不需要硬盘磁头的寻道时间，只需很少的扇区旋转时间，所以速度远快于随机读写

零拷贝

先简单了解下文件系统的操作流程，例如一个程序要把文件内容发送到网络

这个程序是工作在用户空间，文件和网络socket属于硬件资源，两者之间有一个内核空间

在操作系统内部，整个过程为：

在Linux kernel2.2 之后出现了一种叫做"零拷贝(zero-copy)"系统调用机制，就是跳过“用户缓冲区”的拷贝，建立一个磁盘空间和内存的直接映射，数据不再复制到“用户态缓冲区”

 系统上下文切换减少为2次，可以提升一倍的性能

文件分段

kafka的队列topic被分为了多个区partition，每个partition又分为多个段segment，所以一个队列中的消息实际上是保存在N多个片段文件中通过分段的方式，每次文件操作都是对一个小文件的操作，非常轻便，同时也增加了并行处理能力

批量发送

Kafka允许进行批量发送消息，先将消息缓存在内存中，然后一次请求批量发送出去

比如可以指定缓存的消息达到某个量的时候就发出去，或者缓存了固定的时间后就发送出去

如100条消息就发送，或者每5秒发送一次,这种策略将大大减少服务端的I/O次数

数据压缩

Kafka还支持对消息集合进行压缩，Producer可以通过GZIP或Snappy格式对消息集合进行压缩,压缩的好处就是减少传输的数据量，减轻对网络传输的压力

## 1.21(★★★)kafka的作用以及架构？

Kafka的作用

(1)解耦

在项目启动之初来预测将来项目会碰到什么需求，是极其困难的。消息系统在处理过程中间插入了一个隐含的、基于数据的接口层，两边的处理过程都要实现这一接口。这允许你独立的扩展或修改两边的处理过程，只要确保它们遵守同样的接口约束。

(2) 冗余

有些情况下，处理数据的过程会失败。除非数据被持久化，否则将造成丢失。消息队列把数据进行持久化直到它们已经被完全处理，通过这一方式规避了数据丢失风险。许多消息队列所采用的"插入-获取-删除"范式中，在把一个消息从队列中删除之前，需要你的处理系统明确的指出该消息已经被处理完毕，从而确保你的数据被安全的保存直到你使用完毕。

(3) 扩展性

因为消息队列解耦了你的处理过程，所以增大消息入队和处理的频率是很容易的，只要另外增加处理过程即可。不需要改变代码、不需要调节参数。扩展就像调大电力按钮一样简单。

(4) 灵活性 & 峰值处理能力

在访问量剧增的情况下，应用仍然需要继续发挥作用，但是这样的突发流量并不常见；如果为以能处理这类峰值访问为标准来投入资源随时待命无疑是巨大的浪费。使用消息队列能够使关键组件顶住突发的访问压力，而不会因为突发的超负荷的请求而完全崩溃。

(5) 顺序保证

在大多使用场景下，数据处理的顺序都很重要。大部分消息队列本来就是排序的，并且能保证数据会按照特定的顺序来处理。Kafka保证一个Partition内的消息的有序性。

(6) 缓冲

在任何重要的系统中，都会有需要不同的处理时间的元素。例如，加载一张图片比应用过滤器花费更少的时间。消息队列通过一个缓冲层来帮助任务最高效率的执行———写入队列的处理会尽可能的快速。该缓冲有助于控制和优化数据流经过系统的速度。

Kafka架构

一个典型的kafka集群中包含若干producer（可以是web前端产生的page view，或者是服务器日志，系统CPU、memory等），若干broker（Kafka支持水平扩展，一般broker数量越多，集群吞吐率越高），若干consumer group，以及一个[Zookeeper](http://zookeeper.apache.org/) 集群。Kafka通过Zookeeper管理集群配置，选举leader，以及在consumer group发生变化时进行rebalance。producer使用push模式将消息发布到broker，consumer使用pull模式从broker订阅并消费消息。

Broker

Kafka集群包含一个或多个服务器，这种服务器被称为broker

Topic

每条发布到Kafka集群的消息都有一个类别，这个类别被称为Topic。（物理上不同Topic的消息分开存储，逻辑上一个Topic的消息虽然保存于一个或多个broker上但用户只需指定消息的Topic即可生产或消费数据而不必关心数据存于何处）

Partition

Parition是物理上的概念，每个Topic包含一个或多个Partition.

Producer

负责发布消息到Kafka broker

Consumer

消息消费者，向Kafka broker读取消息的客户端。

Consumer Group

每个Consumer属于一个特定的Consumer Group（可为每个Consumer指定group name，若不指定group name则属于默认的group）。

## 1.22(★)kafka高并发性能好还是高吞吐量性能好？

Kafka的设计是把所有的Message都要写入速度低容量大的硬盘，以此来换取更强的存储能力。实际上，Kafka使用硬盘并没有带来过多的性能损失（这一点是有条件限制的，这个条件是，消费者的消费速度要高于或等于生产者的速度）。

kafka重度依赖底层操作系统提供的PageCache功能。（文件缓存，速度相当于操作内存）当上层有写操作时，操作系统只是将数据写入PageCache，同时标记Page属性为Dirty。当读操作发生时，先从PageCache中查找，如果发生缺页才进行磁盘调度，最终返回需要的数据。写入PageCache的数据被定期批量保存到文件系统，减少了磁盘的操作次数，减少系统开销。

实际上PageCache是把尽可能多的空闲内存都当做了磁盘缓存来使用。同时如果有其他进程申请内存，回收PageCache的代价又很小，所以现代的OS都支持PageCache。使用PageCache功能同时可以避免JVM设计带来的GC开销大的问题。

1、kafka一开始是把数据写到PageCache，也就是缓存，如果消费者一直在消费，而且速度大于等于kafka的生产者发送数据的速度，那么消费者会一直从PageCache读取数据，速度等同于内存的操作，不会因为kafka写入磁盘的操作影响吞吐量。

2、当kafka的消费者消费速度不及生产者生产速度时，PageCache存的数据已经是最新的数据了，kafka消费端需要的数据已经被存储磁盘中，这时，kafka的消费速度会受到磁盘的读取速度影响。

问题：kafka的数据一开始就是存储在PageCache上的，定期flush到磁盘上的，也就是说，不是每个消息都被存储在磁盘了，如果出现断电或者机器故障等，PageCache上的数据就丢失了。

kafka是多副本的，当你配置了同步复制之后。多个副本的数据都在PageCache里面，出现多个副本同时挂掉的概率比1个副本挂掉的概率就很小了。（官方推荐是通过副本来保证数据的完整性的）

同时kafka也提供了相关的配置参数，来让你在性能与可靠性之间权衡（一般默认）：

#当达到下面的消息数量时，会将数据flush到日志文件中。默认10000

log.flush.interval.messages=10000

#当达到下面的时间(ms)时，执行一次强制的flush操作。interval.ms和interval.messages无论哪个达到，都会flush。默认3000ms

log.flush.interval.ms=1000

#检查是否需要将日志flush的时间间隔

log.flush.scheduler.interval.ms = 3000

## 1.22(★)说说对kafka分区的理解？

将Kafka视为日志提交服务的话，每一个数据流对应一个Topic，每一个Topic可以有多个Partition，Partition是每个Topic的全量副本，存放在任意一台[broker](https://cloud.tencent.com/developer/information/broker)服务器上，服务器与Partition是一对多的关系，而每个Partition只能被一个消费者（跨Group情况除外）消费。

首先，越多的Partition意味着越多的吞吐量。不管在生产端还是消费端，消息到所有Partition的写入都是同步进行的，一些复杂的运算（比如数据压缩）就会占用更多的硬件资源。

其次，越多的Partition需要越多的文件句柄（Open File Handles）。每个Partition都将一个目录映射给Kafka [broker](https://cloud.tencent.com/developer/information/broker)，在这个日志目录中，每个日志分段（Segment）会有两类文件--索引和实际数据。目前的版本而言，Kafka Baoker会打开这两类文件的举兵。因此越多的Partition就以为着在Broker的系统层有越多的OpenFiles。

越多的Partition会降低可用性

## 1.23(★★★)flume对接到kafka中的时候，offset是怎么维护的(强调不是spark消费时候偏移量的维护)？

kafka的消息所在的位置Topic、Partitions、Offsets三个因素决定。   
Kafka消费者消费的消息位置还与consumer的group.id有关。

earlieastLeaderOffsets：存储在broker上的leader节点的最早的消息偏移量

consumerOffsets：消费者消费的消息偏移量位置

为了表述方便，我们记earlieastLeaderOffsets为A，记consumerOffsets为B 。

情况一：正常情况下，消费的消息偏移量应该大于broker上存储的最早的消息偏移量，即 A < B：

我们知道，存储在broker上的kafka的消息常设置消息过期配置，当到达过期时间时过期的消息将会被清除。

情况二：如果A 依然小于 B，则仍可以正常消费：

情况三：然而，当 A > B 时，则说明还没有被消费的消息已经被清除：

此种情况会抛出 kafka.common.OffsetOutOfRangeException 异常。

consumerOffsets 小于 earlieastLeaderOffsets的影响与解决办法

当情况三发生时，在（B，A）区间内的消息还没有被消费就已经被清除了，将导致两个后果。

1. 消息丢失。

2. 抛出 kafka.common.OffsetOutOfRangeException 异常。

在对消息完整性有严格要求的系统中，消息的丢失造成的影响会比较严重，所以在这种情况下，要保证消息不会遭到丢失。

避免消息丢失包含两个方面：

还没有被消费过的消息不会被清除。

在没有外部系统清除kafka消息的情况下，协调设置broker的最大保留大小 log.retention.bytes 和 最大保留时间log.retention.hours 等，来配合消费者端的读取消息。可以通过读取和监控消费者消费的offsets，来保证消息不会被意外清除。

消费者端消费消息没有遗漏。

当消费者意外中断时，重新启动消费时能够从上一次中断的消息偏移量开始消费

## 1.24(★★★)使用kafka的时候遇到了什么问题？

<https://blog.csdn.net/chizizhixin/article/details/78563595>

Kafka实际使用过程中遇到的一些问题及解决方法：

1.关于Kafka的分区：

开始使用Kafka的时候，没有分区的概念，以为类似于传统的MQ中间件一样，就直接从程序中获取Kafka中的数据。 后来程序搭建了多套，发现永远只有一个消费者（消费者应用部署在多个tomcat上）会从Kafka中获取数据进行处理，后来才知道有分区这么一个概念。

具体不说了，网上有很多资料，总的概括：Kafka的分区，相当于把一个[Topic](https://www.baidu.com/s?wd=Topic&tn=24004469_oem_dg&rsv_dl=gh_pl_sl_csd)再细分成了多个通道，一个消费者应用可以从一个分区或多个分区中获取数据。 有4个分区，1个消费者：这一个消费者需要负责消费四个分区的数据。 有4个分区，2个消费者：每个消费者负责两个分区 有4个分区，3个消费者：消费者1负责1个分区，消费者2负责1个分区，消费者3负责两个分区 有4个分区，4个消费者：一人一个 有4个分区，5个及以上消费者：4个消费者一人一个，剩下的消费者空闲不工作。

部署的时候尽量做到一个消费者对应一个分区。

2.分区数据量不均衡：

Topic上设置了四个分区，压测过程中，发现每个分区的数据量差别挺大的，极端的时候，只有一个分区有数据，其余三个分区空闲。 解决方法，在用生产者生产数据的时候，send方法需要指定key。Kafka会根据key的值，通过一定的算法，如hash，将数据平均的发送到不同的分区上。

3.spring-integration-kafka：

在使用spring-integration-kafka做消费者的时候，发现[CPU](https://www.baidu.com/s?wd=CPU&tn=24004469_oem_dg&rsv_dl=gh_pl_sl_csd)和内存占用量占用非常的大，后来又发现不管生产者发送了多少数据，Kafka的Topic中一直没有数据，这时候才知道spring-integration-kafka会将Topic中的数据全拉到本地，缓存起来，等待后续的处理。 解决方法： <int:channel id="inputFromKafka"> <int:queue capacity="25"/> --这里加个配置，相当于缓存多少数据到本地 </int:channel>

## 1.25(★★)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

## 1.26(★★★)实时方向，sparkstreaming拉取kafka数据多长时间一次？一次是全部拉取吗？

Spark Streaming是将流式计算分解成一系列短小的批处理作业。这里的批处理引擎是Spark，也就是把Spark Streaming的输入数据按照batch size(如1秒)分成一段一段的数据(Discretized Stream)，每一段数据都转换成Spark中的RDD(Resilient Distributed Dataset)，然后将Spark Streaming中对DStream的Transformation操作变为针对Spark中对RDD的Transformation操作，将RDD经过操作变成中间结果保存在内存中。整个流式计算根据业务的需求可以对中间的结果进行叠加，或者存储到外部设备。图1显示了Spark Streaming的整个流程。

Kafka是分布式发布-订阅消息系统。它最初由LinkedIn公司开发，之后成为Apache项目的一部分。Kafka是一个分布式的，可划分的，冗余备份的持久性的日志服务。它主要用于处理活跃的流式数据，如图2所示。

众所周知，大数据时代对数据处理的实时性、稳定性、准确性要求越来越高；现在兴起的组合架构有SparkStreaming对接Kafka，借助SparkStreaming基于内存迭代计算优势和Kafka高并发数据分发能力，进而达到数据处理的实时性；但SparkStreaming对接kafka过程中，仍然难免会出现潜在的数据丢失场景，具体过程如下：

1、两个Exectuor已经从接收器中接收到输入数据，并将它缓存到Exectuor的内存中；2、接收器通知输入源数据已经接收；3、Exectuor根据应用程序的代码开始处理已经缓存的数据；4、这时候Driver突然挂掉了；5、从设计的角度看，一旦Driver挂掉之后，它维护的Exectuor也将全部被kill；6、既然所有的Exectuor被kill了，所以缓存到它们内存中的数据也将被丢失。结果，这些已经通知数据源但是还没有处理的缓存数据就丢失了；7、缓存的时候不可能恢复，因为它们是缓存在Exectuor的内存中，所以数据被丢失了。

由上可见，急需一种防止零丢数的方法来保证SparkStreaming对接Kafka数据处理稳定性。

## 1.27(★)你介绍一下kafka？你们公司为什么用这个？

Kafka是由LinkedIn开发的一个分布式的消息系统，使用Scala编写，它以可水平扩展和高吞吐率而被广泛使用。目前越来越多的开源分布式处理系统如Storm，Spark，Fink都支持与Kafka集成。现在我们的数据实时处理平台也使用到了kafka。现在它已被多家不同类型的公司作为多种类型的数据管道和消息系统使用。

## 1.28(★)kafka的组件

Producer：消息生产者，就是向kafka broker发送消息的客户端

2、Consumer：消息消费者，向kafka broker拉取消息的客户端

3、Topic：每条发布到Kafka集群的消息都有一个类别，这个类别被称为Topic。（物理上不同Topic的消息分开存储，逻辑上一个Topic的消息虽然保存于一个或多个broker上但用户只需指定消息的Topic即可生产或消费数据而不必关心数据存于何处）

4、Broker：Kafka集群包含一个或多个服务器，这种服务器被称为broker

5、Partition：是物理上的概念，每个Topic包含一个或多个Partition（分区），partition中的每个消息都会分配一个有序的id(offset),kafka只保证按一个partition中的顺序将消息发送给Consumer，不保证一个topic整体（partition间）的顺序。

6、Consumer Group：每个Consumer属于一个特定的Consumer Group（可为每个Consumer指定group name，若不指定group name则属于默认的group）

7、Offset：kafka的存储文件都是按照offset. Kafka来命名，用offset做名字的好处就是方便查找。例如你想找打1025的位置，只要找到1024.kafka的文件即可，当然第一个文件就是00000000000.kafka。

## 1.29(★★)kafka为什么要进行分区，怎么分区

若没有分区，一个topic对应的消息集在分布式集群服务组中，就会分布不均匀，即可能导致某台服务器A记录当前topic的消息集很多，若此topic的消息压力很大的情况下，服务器A就可能导致压力很大，吞吐也容易导致瓶颈。

分区：

每个partition可以被认为是一个无限长度的数组，新数据顺序追加进这个数组。物理上，每个partition对应于一个文件夹。一个broker上可以存放多个partition。这样，producer可以将数据发送给多个broker上的多个partition，consumer也可以并行从多个broker上的不同partition上读数据，实现了水平扩展。

## 1.30(★★)kafka存储机制

kafka以topic来进行消息管理，每个topic包含多个partition，每个partition对应一个逻辑log，有多个segment组成。

2）每个segment中存储多条消息（见下图），消息id由其逻辑位置决定，即从消息id可直接定位到消息的存储位置，避

免id到位置的额外映射。

3）每个part在内存中对应一个index，记录每个segment中的第一条消息偏移。

4）发布者发到某个topic的消息会被均匀的分布到多个partition上（或根据用户指定的路由规则进行分布），broker收到

发布消息往对应partition的最后一个segment上添加该消息，当某个segment上的消息条数达到配置值或消息发布时间超

过阈值时，segment上的消息会被flush到磁盘，只有flush到磁盘上的消息订阅者才能订阅到，segment达到一定的大小后

将不会再往该segment写数据，broker会创建新的segment。

## 1.31(★★)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。

# Hive

## Hive原理

#### （★★★）Hive的工作原理

1. 用户提交查询等任务给Driver。

2. 编译器获得该用户的任务Plan。

3. 编译器Compiler根据用户任务去MetaStore中获取需要的Hive的元数据信息。

4. 编译器Compiler得到元数据信息，对任务进行编译，先将HiveQL转换为抽象语法树，然后将抽象语法树转换成查询块，将查询块转化为逻辑的查询计划，重写逻辑查询计划，将逻辑计划转化为物理的计划（MapReduce）, 最后选择最佳的策略。

5. 将最终的计划提交给Driver。

6. Driver将计划Plan转交给ExecutionEngine去执行，获取元数据信息，提交给JobTracker或者SourceManager执行该任务，任务会直接读取HDFS中文件进行相应的操作。

7. 获取执行的结果。

8. 取得并返回执行结果。

#### （★）Hive有哪些保存元数据的方式，每个有什么特点？

1）. 内嵌模式：将元数据保存在本地内嵌的derby数据库中，内嵌的derby数据库每次只能访问一个数据文件，也就意味着它不支持多会话连接。

2）. 本地模式：将元数据保存在本地独立的数据库中（一般是mysql），这可以支持多会话连接。

3） .远程模式：把元数据保存在远程独立的mysql数据库中，避免每个客户端都去安装mysql数据库。

#### （★★★）三种元数据存储方式

参考上题的hive元数据存储

#### （★★）hive 如何权限控制？

1 Metastore Server中基于存储的授权

2 HiveServer2中基于SQL标准的授权

3使用Apache Ranger和Sentry进行授权

4旧的默认Hive授权（传统模式）

详情见官方文档[Hive—LanguageManual+Authorization](https://cwiki.apache.org/confluence/display/Hive/LanguageManual+Authorization)

#### （★★）hive 底层与数据库交互原理

Hive中有两类数据：表数据和元数据。和关系型数据库一样，元数据可以看做是描述数据的数据，包括

1.hive表的数据库名、表名、字段名称与类型、分区字段与类型

2.表及分区的属性、存放位置等 。

元数据存储路径和內表一样，分为本地和远程，可通过hive-site.xml文件设置

参考博客：https://blog.csdn.net/victorzzzz/article/details/81874674

Hive与mysql的关系：只是借用mysql来存储hive中的表的元数据信息，称为metastore

#### （★）Hive是按照什么粒度划分的

按照数据的粒度大小，hive数据可以被组织成：

1）databases： 避免不同表产生命名冲突的一种命名空间

2）tables：具有相同scema的同质数据的集合

3）partitions：一个表可以有一个或多个决定数据如何存储的partition key

4）buckets（或clusters）：在同一个partition中的数据可以根据某个列的hash值分为多个bucket。partition和bucket并非必要，但是它们能大大加快数据的查询速度。

详见 https://blog.csdn.net/w3045872817/article/details/78166612

#### （★）数据仓库与hive的联系

Hive:是一个数据仓库构建的工具，他可以通过derby或者mysql等关系型数据库来记录数据仓库元数据描述信息，可以将结构化的数据映射为一张数据库表，并提供类SQL查询功能，本质将SQL转换为MaoReduce程序。

## Hive数据导入

#### （★★★）Hive数据导入的几种方式

1.本地文件导入到Hive表；

2.Hive表导入到Hive表;

3.HDFS文件导入到Hive表;

4.创建表的过程中从其他表导入;

5.通过sqoop将mysql库导入到Hive表

## Hive表

#### （★★★）Hive几种表之间的区别：

Hive表有受控表(内部表)、外部表、分区表、桶表四种。

**内部表**，就是一般的表，前面讲到的表都是内部标，当表定义被删除的时候，表中的数据随之一并被删除。

**外部表**，数据存在与否和表的定义互不约束，仅仅只是表对hdfs上相应文件的一个引用，当删除表定义的时候，表中的数据依然存在

**分区表**，在Hive Select查询中一般会扫描整个表内容，会消耗很多时间做没必要的工作。有时候只需要扫描表中关心的一部分数据，因此建表时引入了partition概念。分区表指的是在创建表时指定的partition的分区空间

**分桶表**，对于每一个表（table）或者分区， Hive可以进一步组织成桶，也就是说桶是更为细粒度的数据范围划分。Hive也是 针对某一列进行桶的组织。Hive采用对列值哈希，然后除以桶的个数求余的方式决定该条记录存放在哪个桶当中。

### 内部表外部表

#### （★）Hive内部表和外部表的区别

**内部表**：加载数据到Hive所在的HDFS目录，删除时候，元数据和数据文件都删除

**外部表**：不加载数据到Hive所在的HDFS目录，删除时候，只删除表结构

#### （★）生产环境中为什么建议使用外部表？

删除外部表的仅仅会删除元数据，HDFS上的文件不会被删除，这样的好处是操作人员不小心删除表不会将数据也删除。

### 分区分桶

#### （★）HIVE分区和分桶，有什么好处

**分区:**

1、一个表可以拥有一个或者多个分区，每个分区以文件夹的形式单独存在表文件夹的目录下。

2、表和列名不区分大小写。

3、分区是以字段的形式在表结构中存在，通过describetable命令可以查看到字段存在， 但是该字段不存放实际的数据内容，仅仅是分区的表示（伪列） 。

**分桶:**

对于每一个表（table）或者分区， Hive可以进一步组织成桶，也就是说桶是更为细粒度的数据范围划分。Hive也是 针对某一列进行桶的组织。Hive采用对列值哈希，然后除以桶的个数求余的方式决定该条记录存放在哪个桶当中。

**好处:**

（1）获得更高的查询处理效率。桶为表加上了额外的结构，Hive 在处理有些查询时能利用这个结构。具体而言，连接两个在（包含连接列的）相同列上划分了桶的表，可以使用 Map 端连接 （Map-side join）高效的实现。比如JOIN操作。对于JOIN操作两个表有一个相同的列，如果对这两个表都进行了桶操作。那么将保存相同列值的桶进行JOIN操作就可以，可以大大较少JOIN的数据量。

（2）使取样（sampling）更高效。在处理大规模数据集时，在开发和修改查询的阶段，如果能在数据集的一小部分数据上试运行查询，会带来很多方便。

#### （★★）Hive的动态分区和静态分区有什么区别

静态分区是分区的时候指定所分的区，不能自动分区，分区的条件必须自己指定

动态分区可以在程序运行的时候以某一列变量的值进行分区，动态分区必须通过设置打开：如

set hive.exec.dynamic.partition=true;

set hive.exec.dynamic.partition.mode=nonstrict;

#### （★）假如一个分区的数据主表错误怎么通过Hivesql删除HDFS

Alter table ptable drop partition(daytime=’20140911’，city=’bj’)

元数据，数据文件都删除，但是目录daytime=20140911还在

## Hive查询

### UDF

#### （★★）添加自定义UDF的时候都是临时的，怎么永久添加？

1. 将打包好的jar包上传到HDFS上

2. CREATE FUNCTION [db\_name.]function\_name AS class\_name

[USING JAR|FILE|ARCHIVE 'file\_uri' [, JAR|FILE|ARCHIVE 'file\_uri'] ];

file\_uri:是hdfs上的jar包目录

#### （★★★）UDF函数(User Defined Funtion ):用户自定义函数

当Hive提供的内置函数无法满足你的业务处理需要时，此时就可以考虑使用用户自定义函数，用户自定义函数UDF。编写一个UDF，需要继承UDF类，并实现evaluate()函数。在查询执行过程中，查询中对应的每个应用到这个函数的地方都会对这个类进行实例化。对于每行输入都会调用到evaluate()函数。而evaluate()函数处理的值会返回给Hive。同时用户是可以重载evaluate方法的。Hive会像Java的方法重载一样，自动选择匹配的方法。

详细讲解，参考博客：

https://www.cnblogs.com/chushiyaoyue/p/6632090.html?utm\_source=itdadao&utm\_medium=referral

### 窗口函数

#### （★）Hive的开窗函数

over(partition by ......)主要和聚合函数sum()、count()、avg()等结合使用，实现分组聚合的功能

带上group by的hive sql语句只能显示与分组聚合相关的字段，而带上over(partition by ......)的hive sql语句能显示所有字段.。

#### （★）RANK、DENSE\_RANK以及ROW\_NUMBER的区别

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。

### 综合

#### （★）请用一个sql语句返回两张表的差集

select ★ from tb1 where id not in (select id from tb2)

#### （★）什么是左连接?

LEFT JOIN 关键字会从左表那里返回所有的行，即使在右表中没有匹配的行

RIGHT JOIN 关键字会右表那里返回所有的行，即使在左表中没有匹配的行

#### （★★）多表join如何处理?

#### （★）Hive如何解析JSON串

将json以字符串的方式整个入Hive表，然后通过使用UDF函数解析已经导入到hive中的数据，比如使用LATERAL VIEW json\_tuple的方法，获取所需要的列名。

在导入之前将json拆成各个字段，导入Hive表的数据是已经解析过得。这将需要使用第三方的SerDe。

#### （★★）Hive中UDF UDAF UDTF的区别。

UDF：操作作用于单个数据行，并且产生一个数据行作为输出。大多数函数都属于这一类。

UDAF 接受多个输入数据行，并且产生一个输出数据行。向COUNT和MAX这样的函数就是聚集函数。

UDTF 操作作用于单个数据行，并且产生多个数据行-------一个表作为输出。Lateral view explore()

#### （★★）Hive什么时候不能用combiner？ //P

比如使用avg函数的时候，就不能用combiner,一些在map端聚合会和正确结果不一样的时候，不能用combiner。

#### （★）使用Hive实现WordCount

（1） 创建表并关联数据：

create external table wordcount(line string) row format delimited fields terminated by '\n' stored as textfile location '/input/wordcount';

（2） 创建data.txt文件，并上传至HDFS中的/input/wordcount目录下

vi ruozedata.txt

hello,ruoze

hello,jepsondb

hello,www.ruozedata.com

hi,man

hi,gril

（3） 上传命令hdfs dfs -put data.txt /input/wordcount

（4） 拆分：把每行数据差分成单词，这里需要用到一个Hive的内置表生成函数（UDTF）explode(array),参数是array，其实就是行变多列

select word, count(★) from (select explode(split(line, ",")) as word from wordcounts) t group by word;

### Hive函数对比

#### （★）Hive 里面用什么代替 in 查询

hive不支持in 子查询的用法，可以考虑用left join的方式来替换in

#### （★）having和where的区别

Where 子句是用来指定 "行" 的条件的，而Having 子句是指定 “组” 的条件的，

即

Where 子句 = 指定行所对应的条件

Having 子句 = 指定组所对应的条件

#### （★）union all和 union的区别

Union 和 Union All 的区别之一在于对重复结果的处理。

UNION 在进行表链接后会筛选掉重复的记录， 所以在表链接后会对所产生的结果集进行排序运算， 删除重复的记录再返回结果。 实际大部分应用中是不会产生重复的记录， 最常见的是过程表与历史表 UNION。

而 UNION ALL 只是简单的将两个结果合并后就返回。 这样， 如果返回的两个结果集中有重复的数据， 那么返回的结果集就会包含重复的数据了。

从效率上说， UNION ALL 要比 UNION 快很多， 所以， 如果可以确认合并的两个结果集中不包含重复的数据的话， 那么就使用 UNION ALL

#### （★）Hive 的 sort by 和 order by 的区别

sort by：不是全局排序，其在数据进入reducer前完成排序

order by：会对输入做全局排序，因此只有一个reducer（多个reducer无法保证全局有序）。只有一个reducer，会导致当输入规模较大时，需要较长的计算时间。

## Hive文件压缩

#### （★★★）Hive的几种文件格式

1.textfile

2.sequencefile

3.rcfile

4.orc

5.自定义格式

#### （★）hive 中的压缩格式 RCFile、 TextFile、 SequenceFile 各有什么区别？

**TextFile**

文本文件，可以压缩，lzo,...Bzip2压缩文本压缩比较大，时间长，可切割,如果压缩文件不可切割，则由一个map处理大文件

**SequenceFile**

序列文件，kv对，格式紧凑。适合mr的output，可压缩可切割。

该文件和text文件都行方式存储。反之，rcFile/orc/parquet文件格式可以列存储。

SET hive.exec.compress.output=true;

SET io.seqfile.compression.type=BLOCK

**RCFile**

Record Columnar File.

kv存储，类似于sequencefile，将数据文件水平切割多个组。

若干group存放在一个hdfs中，先保存所有行的第一列，第二列，以此类推。

该文件可切割.可以跳过不相关部分，更快得到数据，成本更低。

参考博客：https://blog.csdn.net/tian\_qing\_lei/article/details/77484712

## Hive数据倾斜

#### （★★）数据倾斜问题怎么解决？

数据倾斜产生的原因为分区之后某一个reduce运算的数量比较小，而某一个reduce运行的数据量比较大，造成两个reduce处理数据不平等

合理设置map数量

总结：可以影响map的数量的因素都有哪些？

在input文件夹中，每一个文件就是一个map；input文件的数量, input的文件的大小 。

在MR任务中一个切片就是一个map任务 。

设置切片大小： 在hadoop中

FileInputFormat.setMaxInputSplitSize(job, size);

FileInputFormat.setMinInputSplitSize(job, size);

设置reduce个数

set mapreduce.job.reduces

根据业务自定分区规则

## Hive小文件处理

#### （★★）Hive文件合并

（1） Map操作之前合并小文件

① 每个Map最大输入大小设置为2GB（单位：字节）

setmapred.max.split.size=2048000000

② 执行Map前进行小文件合并

sethive.input.format=org.apache.hadoop.hive.ql.io.CombineHiveInputFormat

（2） 输出时进行合并

① #在Map-only的任务结束时合并小文件

sethive.merge.mapfiles = true

② #在Map-Reduce的任务结束时合并小文件

sethive.merge.mapredfiles= true

③ #合并文件后的大小为1GB左右

sethive.merge.size.per.task = 1024000000

④ #当输出文件的平均大小小于1GB时，启动一个独立的map-reduce任务进行文件merge

sethive.merge.smallfiles.avgsize=1024000000

（3） 如果需要压缩输出文件，就需要增加一个压缩编解码器，同时还有两个压缩方式和多种压缩编码器，压缩方式一个是压缩输出结果，一个是压缩中间结果，按照自己的需求选择，我需要的是gzip就选择的GzipCodec，同时也可以选择使用BZip2Codec、SnappyCodec、LzopCodec进行压缩。压缩文件：

① sethive.exec.compress.output=true;

#默认为false，是否对输出结果进行压缩

② setmapred.output.compression.codec=org.apache.hadoop.io.compress.GzipCodec;

#压缩格式设置

③ setmapred.output.compression.type=BLOCK;

#一共三种压缩方式（NONE, RECORD,BLOCK），BLOCK压缩率最高，一般用BLOCK。

#### （★★）对于小文件的处理办法

我们可以通过一些配置项来使Hive在执行结束后对结果文件进行合并：

hive.merge.mapfiles 在map-only job后合并文件，默认true

hive.merge.mapredfiles 在map-reduce job后合并文件，默认false

hive.merge.size.per.task 合并后每个文件的大小，默认256000000

hive.merge.smallfiles.avgsize 平均文件大小，是决定是否执行合并操作的阈值，默认16000000

Hive在对结果文件进行合并时会执行一个额外的map-only脚本，mapper的数量是文件总大小除以size.per.task参数所得的值，触发合并的条件是：

根据查询类型不同，相应的mapfiles/mapredfiles参数需要打开；

结果文件的平均大小需要大于avgsize参数的值。

详见 https://blog.csdn.net/dxl342/article/details/50932062

## hive优化

#### （★★★）性能调优（重点是 join 的时候如何放置大小表）

这里只讲解hive大小表join优化性能，其他优化参考hive优化总结

1. 小、大表 join

在小表和大表进行join时，将小表放在前边，效率会高。hive会将小表进行缓存。

2．Mapjoin

使用mapjoin将小表放入内存，在map端和大表逐一匹配。从而省去reduce

当一个大表和一个或多个小表做JOIN时，最好使用MAPJOIN，性能比普通的JOIN要快很多。另外，MAPJOIN 还能解决数据倾斜的问题。 MAPJOIN的基本原理是：在小数据量情况下，SQL会将用户指定的小表全部加载到执行JOIN操作的程序的内存中，从而加快JOIN的执行速度。

参考博客：https://blog.csdn.net/u012036736/article/details/84978689

#### （★★）HIVE怎么优化

把hive sql当做mapreduce程序来读，理解hadoop的核心能力，是hive优化的根本。

参考博客：https://www.cnblogs.com/smartloli/p/4356660.html

##### 表的优化

在表的优化中第一点：当数据量比较大的时候常用的手段就是为拆分表，大表拆小表，分区表 ，临时表 。外部表

小表和大表join ,要把数据量的小的表放在join的左边，先进行缓存 ,这样减少表Join的时候内存的消耗量

##### 并行执行

并行执行与java多线程的异步和同步概念上差不多，在MR运行任务中，存在很多的MR任务可以进行执行。有些MR任务和下一个MR任务存在依赖关系，但是有些MR任务没有依赖关系。

例如：

存在依赖关系的MR 它输出就是下一个MR任务的输入。 对于没有依赖关系的MR任务就可以使用并行执行，在同一时间运行多个MR任务。这样在运行的过程中效 率就会得到提升

可以通过以下参数进行设置：

set hive.exec.parallel=true ; // 开启任务并行执行

设置多少个任务可以同时运行

set hive.exec.parallel.thread.number=8; //默认值为8个任务可以同时运行

##### 严格模式

Hive中提供有严格模式，为了防止一些查询。出现不好的影响。例如笛卡儿积。在严格模式下是不能运行的

<property>

<name>hive.mapred.mode</name>

<value>strict</value>

<description>

The mode in which the Hive operations are being performed.

In strict mode, some risky queries are not allowed to run. They include:

Cartesian Product.

No partition being picked up for a query.

Comparing bigints and strings.

Comparing bigints and doubles.

Orderby without limit.

</description>

</property>

默认值为：非严格模式 nonstrict

开启严格模式： strict

开启了严格模式，会对查询语句进行一些限制：

对于分区表： 必须存在where语句对分区表中分区字段进行条件过滤，否则，不允许执行该查询

对于使用order by的语句必须使用limit 进行限定,由于order by 之后所有的数据都会被分到一个reduce中那这样reduce操作的数据量太多了，可能时间过长卡死。所以为了防止reduce时间过程。在order by的时候必须给定 limit 减少redue处理的数据量

限制了笛卡儿积的查询 .主要在多表join中会出现。笛卡儿积的出现会造成性能极大的消耗

##### JVM重用

在hive执行计算任务的时候，会把的执行计划上传到yarn集群中进行提交，运行MR任务。每次进行任务的运行的时候都会开启一个JVM进程运行的MR任务。如果提交任务频繁过多就会造成JVM频繁的开启和关闭。在JVM开启和关闭的过程中会造成大量的资源浪费

在处理小文件的时候，由于map任务较多。所以JVM会频繁的开启和关闭。所以对于小文件的处理优化，主要减少JVM开启的次数

在 mapred-default.xml配置文件中有如下参数

<property>

<name>mapreduce.job.jvm.numtasks</name>

<value>10</value>

<description>How many tasks to run per jvm. If set to -1, there is

no limit.

</description>

</property>

也可以在hive中临时设置JVM重用任务的运行数量

set mapreduce.job.jvm.numtasks

##### 推测执行

由于集群中的资源分配不均等，或者每个集群中节点的硬件性能，会导致某个任务运行的时间快或者某个任务运行的时间慢，或者某个任务在运行的时候直接卡死了

为了防止某些任务，在运行过程中，拖慢了整个MR任务的进度。在运行慢的任务节点上开启相同的任务，如果时间比原来的任务运行的快则直接输出推测运行的任务

设置开启推测执行的参数：

<property>

<name>mapreduce.map.speculative</name>

<value>true</value>

<description>If true, then multiple instances of some map tasks

may be executed in parallel.</description>

</property>

在hadoop中默认开启推测执行。推测执行不是说一卡死就开启任务必须运行%5以上才会开启推测执行

在hive中通过set参数也可以进行设置

set mapreduce.map.speculative=true

默认值为true

推测执行分为map端的推测执行以及reduce端的推测执行

<property>

<name>mapreduce.reduce.speculative</name>

<value>true</value>

<description>If true, then multiple instances of some reduce tasks

may be executed in parallel.</description>

</property>

##### 执行计划

Hive中提供的可以查看Hql语句的执行计划，在执行计划中会生成抽象语法树，在语法树中会显示HQL语句之间的依赖关系以及执行过程。通过这些执行的过程和依赖可以对HQL语句进行优化

##### 虚拟列

虚拟列本身是一个不存在列，在数据查询的时候，可以通过虚拟列去查询数据的的路径，以及数据的偏移量，这两个内容都是hive中为用户提供的虚拟列进行的查询

## Hive对比其他

#### （★★）Pig 和Hive语法有什么不同

①Language

在Hive中可以执行插入/删除 等操作，但是Pig中我没有发现有可以插入数据的方法

②Schemas

Hive中至少还有一个“表”的概念，但是Pig中我认为是基本没有表的概念，所谓的表建立在Pig Latin脚本中，对与Pig更不要提metadata了。

③Partitions

Pig中没有表的概念，所以说到分区对于Pig来说基本免谈，如果跟Hive说“分区”(Partition)他还是能明白的

参考博客：<https://blog.csdn.net/bluejoe2000/article/details/41476465>

#### （★）hive 跟 hbase 的区别是？

|  |  |  |
| --- | --- | --- |
| 区别点 | Hive | Hbase |
| 数据库类型 | 不是数据库 | NoSQL数据库 |
| 处理类型 | 不支持批处理（OLAP） | 支持实时数据流处理（OLTP） |
| 数据模式 | 支持所有 | 无数据模式 |
| 延迟 | 高延迟 | 低延迟 |
| 数据库模型 | 关系型DBMS | 宽列存储 |

#### （★）hive 相对于 Oracle 来说有那些优点？

Hive是一个数据仓库工具，而Oracle是数据库，Hive能够存储海量数据；

Hive还可以进行数据分析；

Hive免费；

#### （★）Spark Sql为什么比Hive快呢？

使用的计算框架不同：

Spark SQL使用的是Spark框架，而如果Hive没有配置Hive On Spark的话，默认是基于MapReduce的，这种方式从提交查询到返回结果需要相当长的时间，而Spark的执行能力原本就比MapReduce快，因此Spark SQL比Hive要快。

#### （★）Hive和HBase对接

Hive与HBase整合的实现是利用两者本身对外的API接口互相通信来完成的，其具体工作交由Hive的lib目录中的hive-hbase-handler-★.jar工具类来实现

详见 https://blog.csdn.net/carl810224/article/details/52382885

# HBase

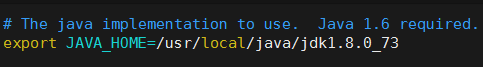
## HBase基本概念

### （★）描述Hbase，zookeeper搭建过程

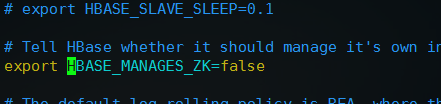
忽略Hadoop配置，开始部署HBase，这里使用的版本为：hbase-0.98.18-hadoop2-bin.tar.gz

和释放hadoop包一样将hbase释放到对应的目录并进入，这里是：/bigdata/hbase/hbase-0.98.18-hadoop2

首先编辑配置文件： vim conf/hbase-env.sh



去掉JAVA\_HOME前面的注释，改为自己实际的JDK安装路径，和配置hadoop类似



然后，去掉export HBASE\_MANAGES\_ZK=true前面的注释并改为export HBASE\_MANAGES\_ZK=false，配置不让HBase管理Zookeeper

　　配置完这两项之后，保存退出

　　编辑文件 vim conf/hbase-site.xml 在configuration标签之间加入如下配置：

<!-- 指定HBase在HDFS上面创建的目录名hbase -->

<property>

<name>hbase.rootdir</name>

<value>hdfs://hadoopha/hbase</value>

</property>

<property>

<name>hbase.master</name>

<value>60000</value>

</property>

<!-- 开启集群运行方式 -->

<property>

<name>hbase.cluster.distributed</name>

<value>true</value>

</property>

<property>

<name>hbase.tmp.dir</name>

<value>/bigdata/hbase/tmp</value>

</property>

<property>

<name>hbase.zookeeper.quorum</name>

<value>bigdata2,bigdata3,bigdata4,bigdata5,bigdata6</value>

</property>

　　分别将hadoop配置下的core-site.xml和hdfs-site.xml复制或者做软链接到hbase配置目录下：

cp /bigdata/hadoop/hadoop-2.6.0/etc/hadoop/core-site.xml conf/

cp /bigdata/hadoop/hadoop-2.6.0/etc/hadoop/hdfs-site.xml conf/

　　执行 vim conf/regionservers 编辑运行regionserver存储服务的Hbase节点，就相当于hadoop slaves中的DataNode节点

　这里是bigdata2~bigdata5

　　保存之后，配置完毕，将hbase发送至其他数据节点：

scp -r /bigdata/hbase/ bigdata2:/bigdata/

scp -r /bigdata/hbase/ bigdata3:/bigdata/

scp -r /bigdata/hbase/ bigdata4:/bigdata/

scp -r /bigdata/hbase/ bigdata5:/bigdata/

scp -r /bigdata/hbase/ bigdata6:/bigdata/

　　然后在bigdata1启动Hbase Master

bin/hbase-daemon.sh start master

　　启动成功，在bigdata1会增加进程：HMaster

　　然后在bigdata2启动regionserver进程，其余4台集群会跟随启动

bin/hbase-daemons.sh start regionserver

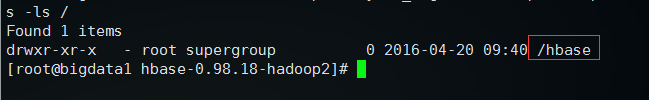
　　这里注意跟随启动时，bigdata2到所有机器ssh确保直接进入，如果配置好的免密也最好提前都进一遍，避免需要输入yes而导致错误

　　同时集群的时间一定同步，否则hbase会启动失败出现NoNode Error的异常

　　在bigdata2到bigdata6会增加进程：HRegionServer

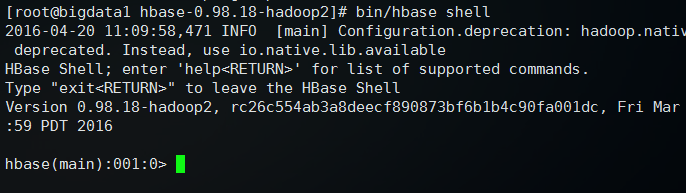
到这里HBase就部署完毕，并且包含zookeeper集群高可用配置

执行命令： /bigdata/hadoop/hadoop-2.6.0/bin/hdfs dfs -ls / 可以查看hbase是否在HDFS文件系统创建成功

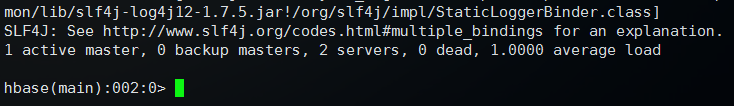


　　看到/hbase节点表示创建成功

　　然后执行： bin/hbase shell 可以进入Hbase管理界面



　　输入 status 查看状态



　　返回状态，表示HBase可以正常使用

　　输入 quit 可以退出管理，回到命令行

　　访问浏览器http://ip:60010可以打开Hbase管理界面

参考链接：https://www.cnblogs.com/freeweb/p/5526080.html

### （★）Hbase集群安装注意事项？

提示：

需要注意的地方是zookeeper的配置。这与hbase-env.sh文件相关，文件中HBASE\_MANAGES\_ZK环境变量用来设置是使用hbase默认自带的zookeeper还是独立的zookeeper。HBASE\_MANAGES\_ZK=false时使用独立的，为true时使用默认自带的。

某个节点的HRegionSever启动失败，这是由于这3个节点的系统时间不一致相差超过集群的检查时间30秒。

### （★）Hbase中的metastore用来做什么的

Hbase的metastore是用来保存数据的，其中保存数据的方式有有三种

第一种与第二种是本地储存，第三种是远程储存这一种企业用的比较多

### （★）start-hbase.sh为起点hbase的启动流程

start-hbase.sh的流程如下：

1.运行hbase-config.sh（作用后面解释）

2.解析参数（0.96版本及以后才可以带唯一参数autorestart，作用就是重启）

3.调用hbase-daemon.sh来启动master；调用hbase-daemons.sh来启动regionserver zookeeper master-backup

参考链接：https://blog.csdn.net/huoyunshen88/article/details/9241449

### （★）设计hbase表需要注意的点

当开始设计HBase中的表的时候需要考虑以下的几个问题：

1. Row Key的结构该如何设置，而Row Key中又该包含什么样的信息（这个很重要，下面的例子会有说明）

2. 表中应该有多少的列族

3. 列族中应该存储什么样的数据

4. 每个列族中存储多少列数据

5. 列的名字分别是什么，因为操作API的时候需要这些信息

6. 单元中(cell)应该存储什么样的信息

7. 每个单元中存储多少个版本信息

详细来源：<http://f.dataguru.cn/thread-854908-1-1.html>

### （★）Hbase的使用场景

1. 瞬间写入量很大，数据库不好支撑或需要很高成本支撑的场景。
2. 数据需要长久保存，且量会持久增长到比较大的场景
3. Hbase不适用与有join，多级索引，表关系复杂的数据模型

### （★）HBase 来源于哪篇博文？ C

A The Google File System  
B MapReduce  
C BigTable  
D Chubby

### （★）下面对 HBase 的描述哪些是正确的？ B、 C、 D

A 不是开源的  
B 是面向列的  
C 是分布式的  
D 是一种 NoSQL 数据库

### （★）HBase 依靠（）存储底层数据 A

A HDFS  
B Hadoop  
C Memory  
D MapReduce

### （★）HBase 依赖（）提供消息通信机制 A

A Zookeeper  
B Chubby  
C RPC  
D Socket

### （★）HBase 依赖（）提供强大的计算能力 D

A Zookeeper  
B Chubby  
C RPC  
D MapReduce

### （★）下面哪些选项正确描述了 HBase 的特性？ A、 B、 C、 D

A 高可靠性  
B 高性能  
C 面向列  
D 可伸缩

### （★）HBase 官方版本可以安装在什么操作系统上？ A、 B、 C

A CentOS  
B Ubuntu  
C RedHat  
D Windows

### （★）HBase 虚拟分布式模式需要（）个节点？ A

A 1  
B 2  
C 3

D 最少 3 个

### （★）HBase 分布式模式最好需要（）个节点？ C

A 1  
B 2  
C 3  
D 最少

### （★）下列哪些选项是安装 HBase 前所必须安装的？ A、 B

A 操作系统  
B JDK  
C Shell Script  
D Java Code

### （★） 解压.tar.gz 结尾的 HBase 压缩包使用的 Linux 命令是？ A

A tar -zxvf  
B tar -zx  
C tar -s  
D tar -nf

### （★）Hbase组件及其作用

1.Client

包含访问HBase的接口，并维护cache来加快对HBase的访问，比如region的位置信息

2.Master

为Region server分配region

负责Region server的负载均衡

发现失效的Region server并重新分配其上的region

管理用户对table的增删改查操作

3.Region Server

Regionserver维护region，处理对这些region的IO请求

Regionserver负责切分在运行过程中变得过大的region

4.Zookeeper作用

1.通过选举，保证任何时候，集群中只有一个master，Master与RegionServers 启动时会向ZooKeeper注册

2.存贮所有Region的寻址入口

3.实时监控Region server的上线和下线信息。并实时通知给Master

4.存储HBase的schema和table元数据

5.默认情况下，HBase 管理ZooKeeper 实例，比如， 启动或者停止ZooKeeper

6.Zookeeper的引入使得Master不再是单点故障

### （★★）hbase的compact

**HBase的compact分析**

HBase是基于LSM树存储模型的分布式NoSQL数据库。LSM树对比普遍的B+树来说，能够获得较高随机写性能的同时，也能保持可靠的随机读性能（可参考这里）。在进行读请求的时候，LSM树要把多个子树（类似B+树结构）进行归并查询，对于HBase来说，这些子树就是HFile（还包括内存上的树结构MemStore）。因此归并查询的子树数越少，查询的性能就越高。

**Compact的作用1**

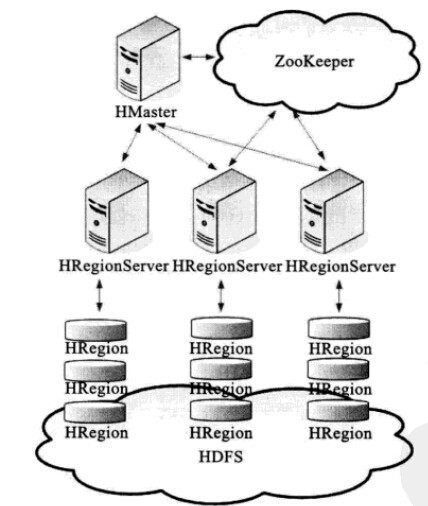
在写请求的这篇文章里，已经介绍过对于每个写请求，都必须写入MemStore以及HLog才算完成事务提交。当MemStore超过阀值的时候，就要flush到HDFS上生成一个HFile。因此随着不断写入，HFile的数量将会越来越多，根据前面所述，HFile数量过多会降低读性能。为了避免对读性能的影响，可以对这些HFile进行compact操作，把多个HFile合并成一个HFile。compact操作需要对HBase的数据进行多次的重新读写，因此这个过程会产生大量的IO。可以看到compact操作的本质就是以IO操作换取后续的读性能的提高。

**Compact的流程**

HBase的compact是针对HRegion的HStore进行操作的。compact操作分为major和minor两种，major会把HStore所有的HFile都compact为一个HFile，并同时忽略标记为delete的KeyValue（被删除的KeyValue只有在compact过程中才真正被"删除"），可以想象major会产生大量的IO操作，对HBase的读写性能产生影响。minor则只会选择数个HFile文件compact为一个HFile，minor的过程一般较快，而且IO相对较低。在日常任务时间，都会禁止mjaor操作，只在空闲的时段定时执行。

参考链接：https://www.cnblogs.com/cxzdy/p/5521357.html

### （★★）Hbase的体系结构



**HRegion**

　　　　当一张表中的数据特别多的时候，HBase把表拆成多个块，每个块就是一个HRegion，每个region中包含这个表里的所有行

**HRegionServer**

　　　　数据库的数据存在HDFS文件系统中，用户通过HRegionServer来获取数据，一台机器上一般只能运行一个HRegionServer，一个HRegion只能属于一个HRegionServer。

　　　　一个Server中有一个HLOG和多个HRegion组成，HLOG用来恢复数据。数据保存跟新时采用先写HLOG的方式。每个HRegion中有一个MemStore和多个StoreFile。数据操作时先缓存在MemStore中，当MemStore超过限制才会刷新到StoreFile中。数据的更新不直接更新采用追加的方式，当StoreFile数量达到最大限制会触发合并操作，这时才会做更新和删除。

**HMaster**

　　　　管理HRegionServer，给HRegionServer分配Region，并根据HRegionServer的加入和故障情况动态调整Region

**ZooKeeper**

　　　　负责监控各个机器的状态，当HRegionServer发生了故障，会通知HMaster去处理。当HMaster故障时，也负责HMaster的恢复工作。

### （★★）Major compack设置为手动会出现什么问题

手动触发：一般来讲，手动触发compaction通常是为了执行major compaction，原因有三，其一是因为很多业务担心自动major compaction影响读写性能，因此会选择低峰期手动触发；其二也有可能是用户在执行完alter操作之后希望立刻生效，执行手动触发major compaction；其三是HBase管理员发现硬盘容量不够的情况下手动触发major compaction删除大量过期数据；无论哪种触发动机，一旦手动触发，HBase会不做很多自动化检查，直接执行合并。

### （★★）Hbase的major compact和minor compact的区别

两者的区别在于：Minor compact只是进行文件merge操作，而Major compact除了做文件Merge操作，还会将其中的delete项删除。

资料来源：https://www.cnblogs.com/cxzdy/p/5368715.html

### （★★）Hbase的工作原理？假如有个RegionServer宕机了，怎么解决

HBase是一种Hadoop Database，是一个高可靠性、高性能、面向列、可伸缩、 实时读写的分布式数据库。利用Hadoop HDFS存储HBase的数据文件,利用Hadoop MapReduce来处理HBase中的海量数据,利用Zookeeper作为HBase分布式协同服务。主要用来存储非结构化和半结构化的松散数据。

工作原理参考：https://blog.csdn.net/qq\_15014327/article/details/83303740

RegionServer宕机之后一方面需要马上将其拉起来，另一方面在拉起来之后需要将该RegionServer上原有的Regions全部迁移回来。

详细资料：https://jingyan.baidu.com/article/7c6fb42832124a80642c90f4.html

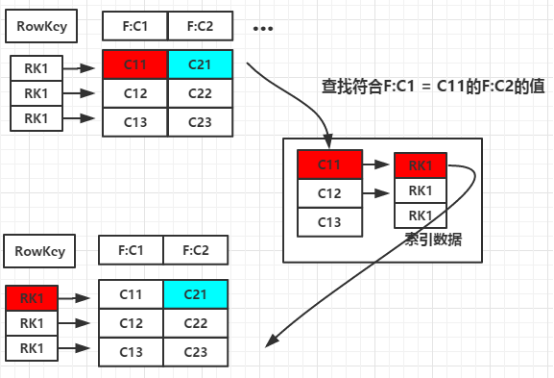
HBase RegionServer宕机恢复三部曲：<http://www.aboutyun.com/thread-20207-1-1.html>

### （★★★）Hbase在进行模型设计时重点在什么地方？一张表中定义多少个Column Family最合适？为什么？

具体看表的数据，一般来说划分标准是根据数据访问频度，如一张表里有些列访问相对频繁，

而另一些列访问很少，这时可以把这张表划分成两个列族，分开存储，提高访问效率

### （★★★）Hbase二级索引



如图，当要对F:C1这列建立索引时，只需要建立F:C1各列值到其对应行键的映射关系，如C11->RK1等，这样就完成了对F:C1列值的二级索引的构建，当要查询符合F:C1=C11对应的F:C2的列值时（即根据C1=C11来查询C2的值,图1青色部分）

其查询步骤如下：

1. 根据C1=C11到索引数据中查找其对应的RK，查询得到其对应的RK=RK1

2. 得到RK1后就自然能根据RK1来查询C2的值了 这是构建二级索引大概思路，其他组合查询的联合索引的建立也类似。

参考：https://blog.csdn.net/wypersist/article/details/79830811

### （★★★）Hbase内部是什么机制？

Hbase 是一个能适应联机业务的数据库系统

物理存储：hbase 的持久化数据是存放在 hdfs 上

存储管理：一个表是划分为很多 region 的，这些 region 分布式地存放在很多 regionserver 上 Region 内部还可以划分为 store，store 内部有 memstore 和 storefile

版本管理：hbase 中的数据更新本质上是不断追加新的版本，通过 compact 操作 来做版本间的文件合并 Region 的 split

集群管理：

Zookeeper 作用在于：

1、hbase regionserver 向 zookeeper 注册，提供 hbase regionserver 状态信 息（是否在线）。

2、hmaster 启动时候会将 hbase 系统表-ROOT- 加载到 zookeeper cluster，通 过 zookeeper cluster 可以获取当前系统表.META.的存储所对应 regionserver 信息。

+ HMaster 的作用： · 为 Region server 分配 region · 负责 Region server 的负载均衡 · 发现失效的 Region server 并重新分配其上的 region · HDFS 上的垃圾文件回收 · 处理 schema 更新请求 HRegionServer 作用： · 维护 master 分配给他的 region，处理对这些 region 的 io 请求 · 负责切分正在运行过程中变的过大的 region

### （★★★）hbase里的hlog的作用

1.如果此Region的MemStore已经有缓存已有写入的数据, 则直接返回;

2.如果没有缓存, 写入HLog(WAL), 再写入MemStore.成功后再返回.

MemStore内存达到一定的值调用flush成为StoreFile,存到HDFS

hbase在写入数据之前会先写入MemStore，成功了再写入HLog，当MemStore的数据丢失的时候，还可以用HLog的数据来进行恢复

## Hbase进阶

### （★★）Hbase写数据的原理是什么

1、客户端向 hregionServer 请求写数据

2、hregionServer 将数据先写入 hlog 中。

3、hregionServer 将数据后写入 memstore 中。

4、当内存中的数据达到阈值 64M 的时候，将数据 Flush 到硬盘中，并同时删除内存

和 hlog 中的历史数据。

5、将硬盘中数据通过 HFile 来序列化，再将数据传输到 HDFS 进行存储。并对 Hlog

做一个标记。

6、当 HDFS 中的数据块达到 4 块的时候，Hmaster 将数据加载到本地进行一个合并

（如果合并后数据的大小小于 256M 则当数据块再次达到 4 块时（包含小于 256M 的

数据块）将最新 4 块数据块再次进行合并，此时数据块大于 256M）。

7、若数据块大于 256M，则将数据重新拆分，将分配后的 region 重新分配给不同的

hregionServer 进行管理。

8、当 hregionServer 宕机后，将 hregionServer 上的 hlog 重新分配给不同的

hregionServer 进行加载（修改.META 文件中关于数据所在 server 的信息）。注意：

hlog 会同步到 HDFS 中。

### （★）说说 hbase 的 API 都有哪些 filter?

**1、RowFilter**

说明：筛选出匹配的所有的行，支持基于行键过滤数据，可以执行精确匹配，子字符串匹配或正则表达式匹配，过滤掉不匹配的数据。

**2、QualifierFilter**

说明：该Filter是一种类似RowFilter的比较过滤器，不同之处是它用来匹配列限定符而不是行健

**3、PrefixFilter**

说明：这是RowFilter的一种特例，它基于行健的前缀值进行过滤，它相当于给扫描构造函数Scan(byte[] startRow, byte[] stopRow)，提供了一个停止键，只是你不需要自己计算停止键。**4、KeyOnlyFilter**

说明：这个Filter只会返回每行的行键+列簇+列，而不返回值，对不需要值的应用场景来说，非常实用，减少了值的传递。

**5、TimestampsFilter**

说明：该过滤器允许针对返回给客户端的时间版本进行更细粒度的控制，使用的时候，可以提供一个返回的时间戳的列表，只有与时间戳匹配的单元才可以返回。当做多行扫描或者是单行检索时，如果需要一个时间区间，可以在Get或Scan对象上使用setTimeRange()方法来实现这一点。

**6、FirstKeyOnlyFilter**

说明：该Filter的作用，是找每一行的第一列数据，找到之后，就会停止扫描。

**7、ColumnPrefixFilter**

说明：该Filter是按照列名的前缀来扫描单元格的，只会返回符合条件的列数据

**8、ValueFilter**

说明：该Filter主要是对值进行过滤，用法和RowFilter类似，只不过侧重点不同而已，针对的是单元值，使用这个过滤器可以过滤掉不符合设定标准的所有单元

**9、ColumnCountGetFilter**

说明：该Filter用来返回每行最多返回多少列，但返回的总数不超过设置的列数

**10、SingleColumnValueFilter**

说明：根据列的值来决定这一行数据是否返回，落脚点在行，而不是列。我们可以设置filter.setFilterIfMissing(true);如果为true，当这一列不存在时，不会返回，如果为false，当这一列不存在时，会返回所有的列信息。

**11、SingleColumnValueExcludeFilter**

说明：该Filter和SingleColumnValueFilter作用类似，唯一的区别在于，返回的数据不包含扫描条件的列。

**12、FilterList**

具体示例参考：https://blog.csdn.net/liuchuanhong1/article/details/53287080

### （★★）HBASE的rowkey怎么创建好？列族怎么创建比较好？HBASE 的特性,以及你怎么去设计 rowkey 和 columnFamily ,怎么去建一个table

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)

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。

因为HBASE是列式数据库，列非表schema的一部分，所以在设计初期只需要考虑rowkey 和 columnFamily即可，rowkey有位置相关性，所以如果数据是练习查询的，最好对同类数据加一个前缀，而每个columnFamily实际上在底层是一个文件，那么文件越小，查询越快，所以讲经常一起查询的列设计到一个列簇，但是列簇不宜过多。

1、列族的数量及列族的势

建议将HBase列族的数量设置的越少越好。当强，对于两个或两个以上的列族HBase并不能处理的很好。这是由于HBase的Flushing和压缩是基于Region的。当一个列族所存储的数据达到Flushing的阈值时，该表中所有列族将同时进行Flushing操作。这将带来不必要的I/O开销，列族越多，该特性带来的影响越大。

此外，还要考虑到同一个表中不同列族所存储的记录数量的差别，即列族的势(Cardinality)。当两个列族数量差别过大时会使包含记录数量较少列族的数据分散在多个Region上，而Region有可能存储在不同的RegionServer上。这样，当进行查询或scan操作的时候，系统效率将会受到影响。

2、行键(RowKey)的设计

首先应该避免使用时序或单调(递减/递增)行键。因为当数据到来的时候，HBase首先需要根据记录的行键来确定存储的位置，即Region的位置，如果使用时序或单调行键，那么连续到来的数据将被分配到同一个Region中，而此时系统的其他Region/RegionServer处于空闲状态，这是分布式最不希望看到的状态。

3、尽量最小化行键和列族的大小

在HBase中，一个具体的值由存储该值的行键、对应的列(列族：列)以及该值的时间戳决定。HBase中索引是为了加速随即访问的速度，索引的创建是基于“行键+列族：列+时间戳+值”的，如果行键和列族的大小过大，甚至超过值本身的大小，纳闷将会增加索引的大小。并且在HBase中数据记录往往非常之多，重复的行键、列将不但使索引的大小过大，也将加重系统的负担

4、版本的数量

默认情况下为3个，可以通过HColumnDescriptor进行设置，建议不要设置的过大

### （★★）Hbase的rowkey设计，影响hbase的性能有哪些

1. hbase.hregion.max.filesize 应该设置多少合适。  
 默认是 256， HStoreFile 的最大值。如果任何一个 Column Family（或者说 HStore）的 HStoreFiles 的大小超过这个值，那么，其所属的 HRegion 就会 Split 成两个。众所周知 hbase 中数据一开始会写入 memstore，当 memstore 满 64MB 以后，会 flush 到 disk 上而成为storefile。当 storefile 数量超过 3 时，会启动 compaction 过程将它们合并为一个 storefile。这个过程中会删除一些 timestamp 过期的数据，比如 update 的数据。而当合并后的 storefile 大小大于 hfile 默认最大值时，会触发 split 动作，将它切分成两个 region。  
2、 autoflush=false 的影响  
 无论是官方还是很多 blog 都提倡为了提高 hbase 的写入速度而在应用代码中设置 autoflush=false，然后 lz 认为在在线应用中应该谨慎进行该设置 原因如下：  
2.1、 autoflush=false 的原理是当客户端提交 delete 或 put 请求时，将该请求在客户端缓存，直到数据超过 2M(hbase.client.write.buffer 决定)或用户执行了 hbase.flushcommits()时才向 regionserver 提交请求。因此即使 htable.put()执行返回成功，也并非说明请求真的成功了。假如还没有达到该缓存而 client 崩溃，该部分数据将由于未发送到 regionserver 而丢失。这对于零容忍的在线服务是不可接受的。  
2.2、 autoflush=true 虽然会让写入速度下降 2-3 倍，但是对于很多在线应用来说这都是必须打开的，也正是 hbase 为什么让它默认值为 true 的原因。当该值为 true 时，每次请求都会发往 regionserver,而regionserver 接收到请求后第一件事就是写 hlog，因此对 io 的要求是非常高的，为了提高 hbase 的写入速度，应该尽可能高地提高 io 吞吐量，比如增加磁盘、使用 raid 卡、减少 replication 因子数等 。从性能的角度谈 table 中 family 和 qualifier 的设置  
3、对于传统关系型数据库中的一张 table，在业务转换到 hbase 上建模时，从性能的角度应该如何设置 family和 qualifier 呢？  
 最极端的， ①每一列都设置成一个 family， ②一个表仅有一个 family，所有列都是其的一个 qualifier，那么有什么区别呢？  
从读的方面考虑：  
 family 越多，那么获取每一个 cell 数据的优势越明显，因为 io 和网络都减少了。如果只有一个 family，那么每一次读都会读取当前 rowkey 的所有数据，网络和 io 上会有一些损失。当然如果要获取的是固定的几列数据，那么把这几列写到一个 family 中比分别设置 family要更好，因为只需一次请求就能拿回所有数据。  
从写的角度考虑：  
 首先，内存方面来说，对于一个Region，会为每一个表的每一个Family分配一个Store，而每一个Store，都会分配一个 MemStore，所以更多的 family 会消耗更多的内存。  
 其次，从 flush 和 compaction 方面说，目前版本的 hbase，在 flush 和 compaction 都是以 region 为单位的，也就是说当一个 family 达到 flush 条件时，该 region 的所有 family 所属的 memstore 都会 flush 一次，即使 memstore 中只有很少的数据也会触发 flush 而生成小文件。这样就增加了 compaction 发生的机率，而 compaction 也是以 region 为单位的，这样就很容易发生 compaction 风暴从而降低系统的整体吞吐量。  
 第三，从 split 方面考虑，由于 hfile 是以 family 为单位的，因此对于多个 family 来说，数据被分散到了更多的 hfile 中，减小了 split 发生的机率。这是把双刃剑。更少的 split 会导致该 region 的体积比较大，由于 balance 是以 region 的数目而不是大小为单位来进行的，因此可能会导致 balance 失效。而从好的方面来说，更少的 split 会让系统提供更加稳定的在线服务。而坏处我们可以通过在请求的低谷时间进行人工的 split 和 balance 来避免掉。因此对于写比较多的系统，如果是离线应该，我们尽量只用一个 family 好了，但如果是在线应用，那还是应该根据应用的情况合理地分配 family

### （★★★）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等指标进行简述）

1. 通过flume将不同系统的日志收集到kafka中
2. 通过storm实时的处理PV、UV、IP
3. 通过kafka的consumer将日志生产到HBASE中。
4. 通过离线的MapReduce或者Hive，处理HBASE中的数据

### （★）介绍一下Hbase的过滤器

参考：http://blog.sina.com.cn/s/blog\_ae33b83901017km4.html

### （★★）Hbase怎么预分区

建表时可以通过 shell 命令预分区，也可以在代码中建表做预分区

### （★★★）Hbase怎么给web前台提供接口来访问（HTABLE可以提供对HTABLE的访问，但是怎么查询同一天记录的多个版本数据）？

使用 HTable 来提供对 HBase 的访问，可以使用时间戳来记录一条数据的多 个版本

### （★★★）htable API有没有线程安全问题，在程序中是单例还是多例？

多例：当多线程去访问同一个表的时候会有

### （★★★）Hbase的并发问题

HBase分别提供了行锁和读写锁来实现行级数据、Store级别以及Region级别的并发控制。除此之外，HBase还提供了MVCC机制实现数据的读写并发控制。MVCC，即多版本并发控制技术，它使得事务引擎不再单纯地使用行锁实现数据读写的并发控制，取而代之的是，把行锁与行的多个版本结合起来，经过简单的算法就可以实现非锁定读，进而大大的提高系统的并发性能。HBase正是使用行锁 ＋ MVCC保证高效的并发读写以及读写数据一致性。

参考链接：https://blog.csdn.net/u012164361/article/details/72758012

### （★）现在我们要对oracle和hbase中的某些表进行更新，你是怎么操作？

提示： disable ‘表名‘

Alter ‘表名‘，name =》’列名‘，versions=》3

Enable ‘表名‘

### （★★）怎么将mysql的数据导入到hbase中？不能使用sqoop，速度太慢了

提示：

A、一种可以加快批量写入速度的方法是通过预先创建一些空的 regions，这样当数据写入hbase时，会按照region分区情况，在集群内作数据的负载均衡

B、hbase里面有这样一个hfileoutputformat类，他可以实现可以将数据转换成hfile格式，通过new这个类，进行相关配置，这样会在hdfs下面产生一个文件，这个时候利用hbase提供的jruby的loadtable.rb脚本就可以进行批量导入。

### （★★）RowKey的后缀匹配怎么实现？例如ROWKEY是yyyyMMDD-UserID形式，如UserID为条件查询数据，怎样实现。

使用rowkey过滤器实现

### （★★）怎么知道hbase表里哪些做索引？哪些没做索引？

提示：

有且仅有一个：rowkey，所以hbase的快速查找建立在rowkey的基础的，而不能像一般的关系型数据库那样建立多个索引来达到多条件查找的结果

### （★★★）简述hbase的瓶颈

提示：

Hbase的瓶颈就是硬盘传输速度，Hbase的操作，它可以往数据里面insert，也可以update一些数据，但update的实际上 也是insert，只是插入一个新的时间戳的一行，delete数据，也是insert，只是insert一行带有delete标记的一行。Habse的所有操作都是追加插入操作。Hbase是一种日志集数据库。它的存储方式，像是日志文件一样。它是批量大量的往硬盘中写，通常都是以文件形式的读写。这个读写速度，就取决于硬盘与机器之间的传输有多快。而orale的瓶颈是硬盘寻道时间。它经常的操作时随机读写。要update一个数据，先要在硬盘中找到这个block，然后把它读入内存，在内存中的缓存中修改，果断时间再回写回去。由于你寻找的block不同，这就存在一个随机的读。硬盘的寻道时间主要有转速来决定的。而寻到时间，技术基本没有改变，这就形成了寻到时间瓶颈。

### （★）Hbase一行数据如何存储

每一行中的每一列数据，都被包装成独立的拥有特定结构的KeyValue

详细链接：https://blog.csdn.net/nosqlnotes/article/details/79647096

拓展：<https://blog.csdn.net/nosqlnotes/article/details/79682656>

Rowkey 设计的原则，下列哪些选项的描述是正确的？ A、 B、 C  
A 尽量保证越短越好  
B 可以使用汉字  
C 可以使用字符串  
D 本身是无序的  
HBase 构建二级索引的实现方式有哪些？ A、 B  
A MapReduce  
B Coprocessor  
C Bloom Filter  
D Filter

关于 HBase 二级索引的描述，哪些是正确的？ A、 B  
A 核心是倒排表  
B 二级索引概念是对应 Rowkey 这个“一级” 索引  
C 二级索引使用平衡二叉树  
D 二级索引使用 LSM 结构  
下列关于 Bloom Filter 的描述正确的是？ A、 C  
A 是一个很长的二进制向量和一系列随机映射函数  
B 没有误算率  
C 有一定的误算率  
D 可以在 Bloom Filter 中删除元素

### （★★★）简述hbase filter的实现原理是什么？集合实际项目经验，写出几个使用filter的场景

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。

1. 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
2. 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。

### （★）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()

}

}

## 3. HBase和Mapreduce、Hive整合

### （★）HBase中的批量加载底层使用（）实现。 A

A MapReduce  
B Hive

C Coprocessor  
D Bloom Filter

### （★）MapReduce 与 HBase 的关系，哪些描述是正确的？ B、 C

A 两者不可或缺， MapReduce 是 HBase 可以正常运行的保证  
B 两者不是强关联关系，没有 MapReduce， HBase 可以正常运行  
C MapReduce 可以直接访问 HBase  
D 它们之间没有任何关系

### （★★★）hive hbase区别 hbase hive查询速度区别（二级索引机制好，hive查询底层还是mr高延迟）

结论：Hbase和Hive在大数据架构中处在不同位置，Hbase主要解决实时数据查询问题，Hive主要解决结构化数据处理和计算问题，一般是配合使用。

1. Hive中的表是纯逻辑表，就只是表的定义等，即表的元数据。Hive本身不存储数据，它完全依赖HDFS和MapReduce。这样就可以将结构化的数据文件映射为为一张数据库表，并提供完整的SQL查询功能，并将SQL语句最终转换为MapReduce任务进行运行。 而HBase表是物理表，适合存放非结构化的数据。

2. Hive是基于MapReduce来处理数据,而MapReduce处理数据是基于行的模式；HBase处理数据是基于列的而不是基于行的模式，适合海量数据的随机访问。

3. HBase的表是疏松的存储的，因此用户可以给行定义各种不同的列；而Hive表是稠密型，即定义多少列，每一行有存储固定列数的数据。

4. Hive使用Hadoop来分析处理数据，而Hadoop系统是批处理系统，因此不能保证处理的低迟延问题；而HBase是近实时系统，支持实时查询。

5. Hive不提供row-level的更新，它适用于大量append-only数据集（如日志）的批任务处理。而基于HBase的查询，支持和row-level的更新。

6. Hive提供完整的SQL实现，通常被用来做一些基于历史数据的挖掘、分析。而HBase不适用与有join，多级索引，表关系复杂的应用场景。

### （★）hive和hbase对接

打通Hive对HBase指定表的全表访问， 再建立一个新的空表， 把查询出来的数据全部导入到新表当中， 以后的所有数据分析操作在新表中完成。

参考博客：https://www.cnblogs.com/npumenglei/p/how\_to\_use\_hive\_load\_data\_from\_hbase.html

### （★）数据同样存在hdfs,为什么hbase支持在线查询

实时查询，可以认为是从内存中查询，一般响应时间在1秒内。HBase的机制是数据先写入到内存中，当数据量达到一定的量（如128M）， 再写入磁盘中， 在内存中，是不进行数据的更新或合并操作的，只增加数据，这使得用户的写操作只要进入内存中就可以立即返回，保证了HBase I/O的 高性能。

实时查询，即反应根据当前时间的数据，可以认为这些数据始终是在内存的，保证了数据的实时响应。

来源：https://www.cnblogs.com/1130136248wlxk/p/4990910.html

### （★★）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。

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。

* 1. 采用压缩流

|  |
| --- |
| //压缩文件  public static void compress(String codecClassNamE、 throws Exception{  Class<?> codecClass = Class.forName(codecClassNamE、;  Configuration conf = new Configuration();  FileSystem fs = FileSystem.get(conF、;  CompressionCodec codec = (CompressionCodeC、ReflectionUtils.newInstance(codecClass, conF、;  //指定压缩文件路径  FSDataOutputStream outputStream = fs.create(new Path("/user/Hadoop/text.gz"));  //指定要被压缩的文件路径  FSDataInputStream in = fs.open(new Path("/user/Hadoop/aa.txt"));  //创建压缩输出流  CompressionOutputStream out = codec.createOutputStream(outputStream);  IOUtils.copyBytes(in, out, conF、;  IOUtils.closeStream(in);  IOUtils.closeStream(out);  } |

* 1. 采用序列化文件

|  |
| --- |
| public void testSeqWrite() throws Exception {  Configuration conf = new Configuration();// 创建配置信息  conf.set("fs.default.name", "HDFS://master:9000");// HDFS默认路径  conf.set("Hadoop.job.ugi", "Hadoop,Hadoop");// 用户和组信息  String uriin = "HDFS://master:9000/ceshi2/";// 文件路径  FileSystem fs = FileSystem.get(URI.create(uriin), conF、;// 创建filesystem  Path path = new Path("HDFS://master:9000/ceshi3/test.seq");// 文件名  IntWritable k = new IntWritable();// key，相当于int  Text v = new Text();// value，相当于String  SequenceFile.Writer w = SequenceFile.createWriter(fs, conf, path,  k.getClass(), v.getClass());// 创建writer  for (int i = 1; i < 100; i++) {// 循环添加  k.set(i);  v.set("abcd");  w.append(k, v);  }  w.close();  IOUtils.closeStream(w);// 关闭的时候flush  fs.close();  } |

HBASE为列存数据库，本身存在压缩机制，所以无需设计。

### （★★）hive hbase的数据迁移

HBase数据迁移:

方式一:copyTable也是属于HBase数据迁移的工具之一，以表级别进行数据迁移。

copyTable的本质也是利用MapReduce进行同步的，与DistCp不同的时，

它是利用MR去scan 原表的数据，然后把scan出来的数据写入到目标集群的表。

这种方式也有很多局限，如一个表数据量达到T级，同时又在读写的情况下，

全量scan表无疑会对集群性能造成影响。

方式二:Export阶段: 将原集群表数据Scan并转换成Sequence File到Hdfs上，

因Export也是依赖于MR的，如果用到独立的MR集群的话，

只要保证在MR集群上关于HBase的配置和原集群一样且能和原集群策略打通(master&regionserver策略），

就可直接用Export命令，如果没有独立MR集群，则只能在HBase集群上开MR，

若需要同步多个版本数据，可以指定versions参数，否则默认同步最新版本的数据，还可以指定数据起始结束时间

Import阶段:　将原集群Export出的SequenceFile导到目标集群对应表

hive数据迁移

https://blog.csdn.net/github\_39577257/article/details/82390507

很详细可以看看

## 4.HBase和其它数据库对比

### （★★）Mongodb和hbase的区别

最近公司想要做数据分析，之前我们公司用的是免费的growing IO。他们分析仅限于界面跳转的转化率，不能详细地分析业务数据。我研究了一个需要埋点的产品，搞明白他们是在每个接口的调用埋点，将用户对接口的调用行为记录下来，进行分析。由于接口众多，每个接口的数据都不同。可以充分利用hbase宽表的特性，在一行中定义一个通用的字段来标示当前行的数据类型，操作人，然后定义不同的字段来记录每一种数据。在插入数据的时候，每一行只插入当前类型和当前数据。由于hbase的宽表特性，可以容纳上百万列。可以将一家公司所有的接口访问数据都记录到一张无限大的表中，再配合辅助的用户表，就可以在各种纬度上分析用户的行为了。

分析了他们的表结构，我想用mongodb也可以做同样的事情，并且mongo比hbase好的地方在于，他入门门槛相对较低，然后在索引方面，检索的速度远比hbase那种查询要快多了。hbase只能要么按照主键范围查询，要么全表检索。为什么大的互联网公司都在推行hbase呢，这个是困扰我的地方。问了一个前腾讯员工，搞明白了两者的区别。

原因就在于写入的速度，hbase由于只维护一个主键，写入的速度要比mongodb这种要维护所有索引的数据库快多了。hbase占用两台机器能完成的事情，mongodb要占用更多的机器，每台机器按一年20000的费用，几百台下来就是一笔很大的费用。但是代价就是hbase记录下东西以后，只能事后通过全表检索或按照索引范围的方式进行整体分析，而不能对具体每个人的数据进行实时分析，更强调数据分析能力而不是实时数据查询能力，因此各有千秋吧。像用户行为分析的这种，一开始产品经理可能会具体看某一个人的数据，但是新鲜过后，只会看程序的分析结果了。因此从经济的角度出发，对于用户行为分析这种不需要实时数据的需求来说，hbase+mysql就可以用最经济的方式解决了。mongodb比较适合需要实时返回数据的大数据应用。

### （★）Redis,传统数据库,HBase,Hive 每个之间的区别

redis：分布式缓存，强调缓存，内存中数据

传统数据库：注重关系

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

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

### （★）hbase与mysql的区别

①定义：

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）。

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。

②数据存储方式

a)MySQL采用行存储

MySQL行存储的方式比较适合OLTP业务。

b)HBase是面向列的NoSql数据库

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。

③适用场景：

比较点 MySQL HBase

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

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

扩展性 单机、扩展性差 水平扩展

事务 支持 不支持

一致性 强一致性 强一致性、时间线一致性

二级索引 支持 不一致性

全文索引 支持 不支持

### （★）存入HBASE或者mysql的依据是什么

**数据类型：**Hbase只有简单的字符类型，所有的类型都是交由用户自己处理，它只保存字符串。而关系数据库有丰富的类型和存储方式。

**数据操作：**HBase只有很简单的插入、查询、删除、清空等操作，表和表之间是分离的，没有复杂的表和表之间的关系，而传统数据库通常有各式各样的函数和连接操作。

**存储模式：**HBase是基于列存储的，每个列族都由几个文件保存，不同的列族的文件时分离的。而传统的关系型数据库是基于表格结构和行模式保存的

**数据维护：**HBase的更新操作不应该叫更新，它实际上是插入了新的数据，而传统数据库是替换修改

**可伸缩性**：Hbase这类分布式数据库就是为了这个目的而开发出来的，所以它能够轻松增加或减少硬件的数量，并且对错误的兼容性比较高。而传统数据库通常需要增加中间层才能实现类似的功能

**数据量的多少和是否要及时反馈**

## 5.HBase优化

### （★） HBase 性能优化包含下面的哪些选项？

答： A、 B、 C、 D  
A 读优化  
B 写优化  
C 配置优化  
D JVM 优化

### （★★★）简述hbase性能优化的思路

1、在库表设计的时候，尽量考虑rowkey和columnfamily的特性

2、进行HBASE集群的调优：见HBASE调优

### （★★★）Hbase优化

1、表设计

建表时就分区，rowkey设置定长（64字节），CF2到3个

Max Versio，Time to live，Compact&Split

2、写表

多Htable并发写

Htable参数设置，手动flush，降低IO

WriteBuffer

批量写

多线程并发写

3、读表

多Htable并发读

Htable参数设置

批量读

释放资源

缓存查询结果

rowkey设计，一般设置成定长的，最大长度为64kb 实际开发设计成10-100Bytes，同时列簇设计时候，不要太多，并且设置表中的生命周期，过期的数据定时删除，例如数据只需要保存两天，那么可以减少不必要的空间开销

如果我们在查询的时候，最好要指定需要的列族，这样可以减少网络传输的数据量，否则的话会返回所有的列簇的数据，开销太大

还有就是客户端入库调优，也就是说Hbase的自动刷写是默认开启的，用户每次put的话都会提交到Hbaseserver上进行刷新一次，如果高速的插入数据，就会造成io负载过重，影响性能，这个时候就需要关闭自动刷写功能，setAutoFlush(false),如此一来这个put数据会写入缓存中，缓存满了才会刷新到Hbase中

### （★★）如何提高Hbase客户端的读写性能？请举例说明

开启 bloomfilter 过滤器，开启 bloomfilter 比没开启要快 3、4 倍

②Hbase 对于内存有特别的嗜好，在硬件允许的情况下配足够多的内存给它

③通过修改 hbase-env.sh 中的

export HBASE\_HEAPSIZE=3000 #这里默认为 1000m

④增大 RPC 数量

通过修改 hbase-site.xml 中的

hbase.regionserver.handler.count 属性，可以适当的放大。默认值为 10 有点小

## 6.其他

### （★）实时数据统计会用到哪些技术，它们各自的应用场景区别是什么？

flume：日志收集系统，主要用于系统日志的收集

kafka：消息队列，进行消息的缓存和系统的解耦

storm：实时计算框架，进行流式的计算。

### （★★★）如果让你设计，你觉得一个分布式文件系统应该如何设计，考虑哪方面内容；

互联网时代，流量和数据快速增长，分布式文件系统所要解决的主要场景需要非常大的磁盘空间，这在磁盘体系上垂直扩容是无法达到的，必须要分布式，同时分布式架构下，主机都是可靠性不是非常好的普通服务器，因此容错、高可用、持久化、伸缩性等指标，就成为必须要考量的特性。

1、对分布式文件系统的要求

对一个分布式文件系统而言，有一些特性是必须要满足的，否则就无法有竞争力。主要如下：

应该符合 POSIX 的文件接口标准，使该系统易于使用，同时对于用户的遗留系统也无需改造；

对用户透明，能够像使用本地文件系统那样直接使用；

持久化，保证数据不会丢失；

具有伸缩性，当数据压力逐渐增长时能顺利扩容；

具有可靠的安全机制，保证数据安全；

数据一致性，只要文件内容不发生变化，什么时候去读，得到的内容应该都是一样的。

支持的空间越大越好；

支持的并发访问请求越多越好；

性能越快越好；

硬件资源的利用率越高越合理，就越好。

2、架构模型

从业务模型和逻辑架构上，分布式文件系统需要这几类组件：

存储组件：负责存储文件数据，它要保证文件的持久化、副本间数据一致、数据块的分配 / 合并等等；

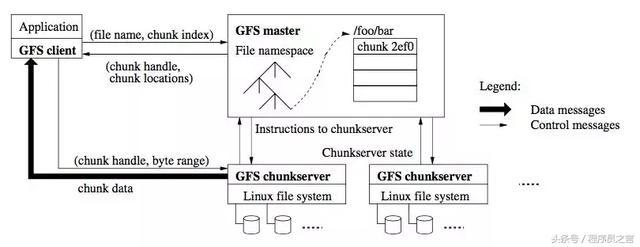
管理组件：负责 meta 信息，即文件数据的元信息，包括文件存放在哪台服务器上、文件大小、权限等，除此之外，还要负责对存储组件的管理，包括存储组件所在的服务器是否正常存活、是否需要数据迁移等；

接口组件：提供接口服务给应用使用，形态包括 SDK(Java/C/C++ 等)、CLI 命令行终端、以及支持 FUSE 挂载机制。

而在部署架构上，有着“中心化”和“无中心化”两种路线分歧，即是否把“管理组件”作为分布式文件系统的中心管理节点。两种路线都有很优秀的产品，下面分别介绍它们的区别。

有中心节点

以 GFS 为代表，中心节点负责文件定位、维护文件 meta 信息、故障检测、数据迁移等管理控制的职能，下图是 GFS 的架构图。



该图中 GFS master 即为 GFS 的中心节点，GF chunkserver 为 GFS 的存储节点。其操作路径如下：

Client 向中心节点请求“查询某个文件的某部分数据”；

中心节点返回文件所在的位置 (哪台 chunkserver 上的哪个文件) 以及字节区间信息；

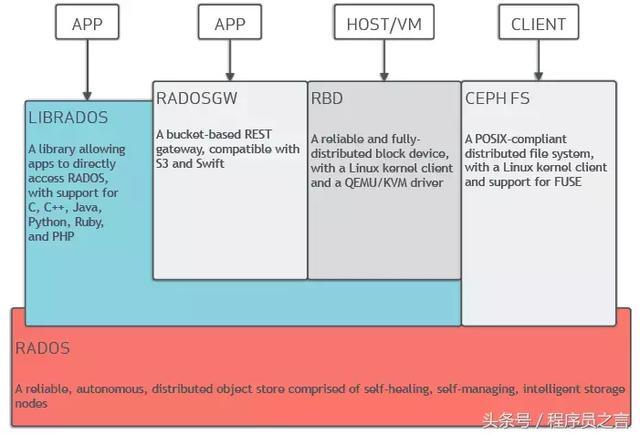
Client 根据中心节点返回的信息，向对应的 chunk server 直接发送数据读取的请求；

chunk server 返回数据。

在这种方案里，一般中心节点并不参与真正的数据读写，而是将文件 meta 信息返回给 Client 之后，即由 Client 与数据节点直接通信。其主要目的是降低中心节点的负载，防止其成为瓶颈。这种有中心节点的方案，在各种存储类系统中得到了广泛应用，因为中心节点易控制、功能强大。

无中心节点

以 ceph 为代表，每个节点都是自治的、自管理的，整个 ceph 集群只包含一类节点，如下图 (最下层红色的 RADOS 就是 ceph 定义的“同时包含 meta 数据和文件数据”的节点)。



A [Ceph Node] leverages commodity hardware and intelligent daemons, and a [Ceph Storage Cluster] accommodates large numbers of nodes, which communicate with each other to replicate and redistribute data dynamically. Ceph provides an infinitely scalable [Ceph Storage Cluster] based upon RADOS

无中心化的最大优点是解决了中心节点自身的瓶颈，这也就是 ceph 号称可以无限向上扩容的原因。但由 Client 直接和 Server 通信，那么 Client 必须要知道，当对某个文件进行操作时，它该访问集群中的哪个节点。ceph 提供了一个很强大的原创算法来解决这个问题——CRUSH 算法。

3、持久化

对于文件系统来说，持久化是根本，只要 Client 收到了 Server 保存成功的回应之后，数据就不应该丢失。这主要是通过多副本的方式来解决，但在分布式环境下，多副本有这几个问题要面对。

如何保证每个副本的数据是一致的?

如何分散副本，以使灾难发生时，不至于所有副本都被损坏?

怎么检测被损坏或数据过期的副本，以及如何处理?

该返回哪个副本给 Client?

如何保证每个副本的数据是一致的

同步写入是保证副本数据一致的最直接的办法。当 Client 写入一个文件的时候，Server 会等待所有副本都被成功写入，再返回给 Client。

这种方式简单、有保障，唯一的缺陷就是性能会受到影响。假设有 3 个副本，如果每个副本需要 N 秒，则可能会阻塞 Client 3N 秒的时间，有几种方式，可以对其进行优化：

并行写：由一个副本作为主副本，并行发送数据给其他副本；

链式写：几个副本组成一个链 (chain)，并不是等内容都接受到了再往后传播，而是像流一样，边接收上游传递过来的数据，一边传递给下游。

还有一种方式是采用 CAP 中所说的 W+R>N 的方式，比如 3 副本 (N=3) 的情况，W＝2，R＝2，即成功写入 2 个就认为成功，读的时候也要从 2 个副本中读。这种方式通过牺牲一定的读成本，来降低写成本，同时增加写入的可用性。这种方式在分布式文件系统中用地比较少。

如何分散副本，以使灾难发生时，不至于所有副本都被损坏

这主要避免的是某机房或某城市发生自然环境故障的情况，所以有一个副本应该分配地比较远。它的副作用是会带来这个副本的写入性能可能会有一定的下降，因为它离 Client 最远。所以如果在物理条件上无法保证够用的网络带宽的话，则读写副本的策略上需要做一定考虑。可以参考同步写入只写 2 副本、较远副本异步写入的方式，同时为了保证一致性，读取的时候又要注意一些，避免读取到异步写入副本的过时数据。

怎么检测被损坏或数据过期的副本，以及如何处理

如果有中心节点，则数据节点定期和中心节点进行通信，汇报自己的数据块的相关信息，中心节点将其与自己维护的信息进行对比。如果某个数据块的 checksum 不对，则表明该数据块被损坏了；如果某个数据块的 version 不对，则表明该数据块过期了。

如果没有中心节点，以 ceph 为例，它在自己的节点集群中维护了一个比较小的 monitor 集群，数据节点向这个 monitor 集群汇报自己的情况，由其来判定是否被损坏或过期。

当发现被损坏或过期副本，将它从 meta 信息中移除，再重新创建一份新的副本就好了，移除的副本在随后的回收机制中会被收回。

该返回哪个副本给 Client

这里的策略就比较多了，比如 round-robin、速度最快的节点、成功率最高的节点、CPU 资源最空闲的节点、甚至就固定选第一个作为主节点，也可以选择离自己最近的一个，这样对整体的操作完成时间会有一定节约。

4、伸缩性

存储节点的伸缩

当在集群中加入一台新的存储节点，则它主动向中心节点注册，提供自己的信息，当后续有创建文件或者给已有文件增加数据块的时候，中心节点就可以分配到这台新节点了，比较简单。但有一些问题需要考虑。

如何尽量使各存储节点的负载相对均衡?

怎样保证新加入的节点，不会因短期负载压力过大而崩塌?

如果需要数据迁移，那如何使其对业务层透明?

如何尽量使各存储节点的负载相对均衡

首先要有评价存储节点负载的指标。有多种方式，可以从磁盘空间使用率考虑，也可以从磁盘使用率 +CPU 使用情况 + 网络流量情况等做综合判断。一般来说，磁盘使用率是核心指标。

其次在分配新空间的时候，优先选择资源使用率小的存储节点；而对已存在的存储节点，如果负载已经过载、或者资源使用情况不均衡，则需要做数据迁移。

怎样保证新加入的节点，不会因短期负载压力过大而崩塌

当系统发现当前新加入了一台存储节点，显然它的资源使用率是最低的，那么所有的写流量都路由到这台存储节点来，那就可能造成这台新节点短期负载过大。因此，在资源分配的时候，需要有预热时间，在一个时间段内，缓慢地将写压力路由过来，直到达成新的均衡。

如果需要数据迁移，那如何使其对业务层透明?

在有中心节点的情况下，这个工作比较好做，中心节点就包办了——判断哪台存储节点压力较大，判断把哪些文件迁移到何处，更新自己的 meta 信息，迁移过程中的写入怎么办，发生重命名怎么办。无需上层应用来处理。

如果没有中心节点，那代价比较大，在系统的整体设计上，也是要考虑到这种情况，比如 ceph，它要采取逻辑位置和物理位置两层结构，对 Client 暴露的是逻辑层 (pool 和 place group)，这个在迁移过程中是不变的，而下层物理层数据块的移动，只是逻辑层所引用的物理块的地址发生了变化，在 Client 看来，逻辑块的位置并不会发生改变。

中心节点的伸缩

如果有中心节点，还要考虑它的伸缩性。由于中心节点作为控制中心，是主从模式，那么在伸缩性上就受到比较大的限制，是有上限的，不能超过单台物理机的规模。我们可以考虑各种手段，尽量地抬高这个上限。有几种方式可以考虑：

以大数据块的形式来存储文件——比如 HDFS 的数据块的大小是 64M，ceph 的的数据块的大小是 4M，都远远超过单机文件系统的 4k。它的意义在于大幅减少 meta data 的数量，使中心节点的单机内存就能够支持足够多的磁盘空间 meta 信息；

中心节点采取多级的方式——顶级中心节点只存储目录的 meta data，其指定某目录的文件去哪台次级总控节点去找，然后再通过该次级总控节点找到文件真正的存储节点；

中心节点共享存储设备——部署多台中心节点，但它们共享同一个存储外设 / 数据库，meta 信息都放在这里，中心节点自身是无状态的。这种模式下，中心节点的请求处理能力大为增强，但性能会受一定影响。iRODS 就是采用这种方式。

5、高可用性

中心节点的高可用

中心节点的高可用，不仅要保证自身应用的高可用，还得保证 meta data 的数据高可用。

meta data 的高可用主要是数据持久化，并且需要备份机制保证不丢。一般方法是增加一个从节点，主节点的数据实时同步到从节点上。也有采用共享磁盘，通过 raid1 的硬件资源来保障高可用。显然增加从节点的主备方式更易于部署。

meta data 的数据持久化策略有以下几种方式。

直接保存到存储引擎上，一般是数据库。直接以文件形式保存到磁盘上，也不是不可以，但因为 meta 信息是结构化数据，这样相当于自己研发出一套小型数据库来，复杂化了。

保存日志数据到磁盘文件 (类似 MySQL 的 binlog 或 Redis 的 aof)，系统启动时在内存中重建成结果数据，提供服务。修改时先修改磁盘日志文件，然后更新内存数据。这种方式简单易用。

当前内存服务 + 日志文件持久化是主流方式。一是纯内存操作，效率很高，日志文件的写也是顺序写；二是不依赖外部组件，独立部署。

为了解决日志文件会随着时间增长越来越大的问题，以让系统能以尽快启动和恢复，需要辅助以内存快照的方式——定期将内存 dump 保存，只保留在 dump 时刻之后的日志文件。这样当恢复时，从最新一次的内存 dump 文件开始，找其对应的 checkpoint 之后的日志文件开始重播。

存储节点的高可用

在前面“持久化”章节，在保证数据副本不丢失的情况下，也就保证了其的高可用性。

6、性能优化和缓存一致性

这些年随着基础设施的发展，局域网内千兆甚至万兆的带宽已经比较普遍，以万兆计算，每秒传输大约 1250M 字节的数据，而 SATA 磁盘的读写速度这些年基本达到瓶颈，在 300-500M/s 附近，也就是纯读写的话，网络已经超过了磁盘的能力，不再是瓶颈了，像 NAS 网络磁盘这些年也开始普及起来。

但这并不代表，没有必要对读写进行优化，毕竟网络读写的速度还是远慢于内存的读写。常见的优化方法主要有：

内存中缓存文件内容；

预加载数据块，以避免客户端等待；

合并读写请求，也就是将单次请求做些积累，以批量方式发送给 Server 端。

缓存的使用在提高读写性能的同时，也会带来数据不一致的问题：

会出现更新丢失的现象。当多个 Client 在一个时间段内，先后写入同一个文件时，先写入的 Client 可能会丢失其写入内容，因为可能会被后写入的 Client 的内容覆盖掉；

数据可见性问题。Client 读取的是自己的缓存，在其过期之前，如果别的 Client 更新了文件内容，它是看不到的；也就是说，在同一时间，不同 Client 读取同一个文件，内容可能不一致。

这类问题有几种方法：

文件只读不改：一旦文件被 create 了，就只能读不能修改。这样 Client 端的缓存，就不存在不一致的问题；

通过锁：用锁的话还要考虑不同的粒度。写的时候是否允许其他 Client 读? 读的时候是否允许其他 Client 写? 这是在性能和一致性之间的权衡，作为文件系统来说，由于对业务并没有约束性，所以要做出合理的权衡，比较困难，因此最好是提供不同粒度的锁，由业务端来选择。但这样的副作用是，业务端的使用成本抬高了。

7、安全性

由于分布式文件存储系统，肯定是一个多客户端使用、多租户的一个产品，而它又存储了可能是很重要的信息，所以安全性是它的重要部分。

主流文件系统的权限模型有以下这么几种。

DAC: 全称是 Discretionary Access Control，就是我们熟悉的 Unix 类权限框架，以 user-group-privilege 为三级体系，其中 user 就是 owner，group 包括 owner 所在 group 和非 owner 所在的 group、privilege 有 read、write 和 execute。这套体系主要是以 owner 为出发点，owner 允许谁对哪些文件具有什么样的权限。

MAC: 全称是 Mandatory Access Control，它是从资源的机密程度来划分。比如分为“普通”、“机密”、“绝密”这三层，每个用户可能对应不同的机密阅读权限。这种权限体系起源于安全机构或军队的系统中，会比较常见。它的权限是由管理员来控制和设定的。Linux 中的 SELinux 就是 MAC 的一种实现，为了弥补 DAC 的缺陷和安全风险而提供出来。关于 SELinux 所解决的问题可以参考 What is SELinux?

RBAC: 全称是 Role Based Access Control，是基于角色 (role) 建立的权限体系。角色拥有什么样的资源权限，用户归到哪个角色，这对应企业 / 公司的组织机构非常合适。RBAC 也可以具体化，就演变成 DAC 或 MAC 的权限模型。

市面上的分布式文件系统有不同的选择，像 ceph 就提供了类似 DAC 但又略有区别的权限体系，Hadoop 自身就是依赖于操作系统的权限框架，同时其生态圈内有 Apache Sentry 提供了基于 RBAC 的权限体系来做补充。

8、其他

空间分配

有连续空间和链表空间两种。连续空间的优势是读写快，按顺序即可，劣势是造成磁盘碎片，更麻烦的是，随着连续的大块磁盘空间被分配满而必须寻找空洞时，连续分配需要提前知道待写入文件的大小，以便找到合适大小的空间，而待写入文件的大小，往往又是无法提前知道的 (比如可编辑的 word 文档，它的内容可以随时增大)；

而链表空间的优势是磁盘碎片很少，劣势是读写很慢，尤其是随机读，要从链表首个文件块一个一个地往下找。为了解决这个问题，出现了索引表——把文件和数据块的对应关系也保存一份，存在索引节点中 (一般称为 i 节点)，操作系统会将 i 节点加载到内存，从而程序随机寻找数据块时，在内存中就可以完成了。通过这种方式来解决磁盘链表的劣势，如果索引节点的内容太大，导致内存无法加载，还有可能形成多级索引结构。

文件删除

实时删除还是延时删除? 实时删除的优势是可以快速释放磁盘空间；延时删除只是在删除动作执行的时候，置个标识位，后续在某个时间点再来批量删除，它的优势是文件仍然可以阶段性地保留，最大程度地避免了误删除，缺点是磁盘空间仍然被占着。在分布式文件系统中，磁盘空间都是比较充裕的资源，因此几乎都采用逻辑删除，以对数据可以进行恢复，同时在一段时间之后 (可能是 2 天或 3 天，这参数一般都可配置)，再对被删除的资源进行回收。

怎么回收被删除或无用的数据? 可以从文件的 meta 信息出发——如果 meta 信息的“文件 - 数据块”映射表中包含了某个数据块，则它就是有用的；如果不包含，则表明该数据块已经是无效的了。所以，删除文件，其实是删除 meta 中的“文件 - 数据块”映射信息 (如果要保留一段时间，则是把这映射信息移到另外一个地方去)。

面向小文件的分布式文件系统

有很多这样的场景，比如电商——那么多的商品图片、个人头像，比如社交网站——那么多的照片，它们具有的特性，可以简单归纳下：

每个文件都不大；

数量特别巨大；

读多写少；

不会修改。

针对这种业务场景，主流的实现方式是仍然是以大数据块的形式存储，小文件以逻辑存储的方式存在，即文件 meta 信息记录其是在哪个大数据块上，以及在该数据块上的 offset 和 length 是多少，形成一个逻辑上的独立文件。这样既复用了大数据块系统的优势和技术积累，又减少了 meta 信息。

文件指纹和去重

文件指纹就是根据文件内容，经过算法，计算出文件的唯一标识。如果两个文件的指纹相同，则文件内容相同。在使用网络云盘的时候，发现有时候上传文件非常地快，就是文件指纹发挥作用。云盘服务商通过判断该文件的指纹，发现之前已经有人上传过了，则不需要真的上传该文件，只要增加一个引用即可。在文件系统中，通过文件指纹可以用来去重、也可以用来判断文件内容是否损坏、或者对比文件副本内容是否一致，是一个基础组件。

文件指纹的算法也比较多，有熟悉的 md5、sha256、也有 google 专门针对文本领域的 simhash 和 minhash 等。

### （★★★）每天百亿数据入Hbase，如何保证数据的存储正确和在规定的时间里全部录入完毕，不残留数据

答：看到这个题目的时候我们要思考的是它在考查什么知识点？

我们来看看要求：

1）百亿数据：证明数据量非常大

2）存入HBase：证明是跟HBase的写入数据有关

3）保证数据的正确：要设计正确的数据结构保证正确性

4）在规定时间内完成：对存入速度是有要求的

那么针对以上的四个问题我们来一一分析

1）数据量百亿条，什么概念呢？假设一整天60x60x24 = 86400秒都在写入数据，那么每秒的写入条数高达100万条，HBase当然是支持不了每秒百万条数据的，所以这百亿条数据可能不是通过实时地写入，而是批量地导入。批量导入推荐使用BulkLoad方式（推荐阅读：Spark之读写HBase），性能是普通写入方式几倍以上

2）存入HBase：普通写入是用JavaAPI put来实现，批量导入推荐使用BulkLoad

3）保证数据的正确：这里需要考虑RowKey的设计、预建分区和列族设计等问题

4）在规定时间内完成也就是存入速度不能过慢，并且当然是越快越好，使用BulkLoad

### （★★）开发中遇到的问题

Hbase节点运行很慢，发现是Hfile过小，hbase频繁split。

修改了hfile大小。或者设置major compack设置为手动

### （★★）你这边离线数据存hive，mysql，hbase，这个hbase的一个rowkey设计是怎么考虑的？

1)、RowKey长度原则：RowKey是一个二进制码流，可以是任意字符串，最大长度为64KB，实际应用中一般为10~100bytes，存为byte[]字节数组，一般设计成定长。建议是越短越好，不要超过16个字节。原因一是数据的持久化文件HFile中是按照KeyValue存储的，如果RowKey过长比如100字节，1000万列数据光RowKey就要占用100\*1000万=10亿个字节，将近1G数据，这会极大影响HFile的存储效率；原因二是memstore将缓存部分数据到内存，如果RowKey字段过长内存的有效利用率会降低，系统将无法缓存更多的数据，这会降低检索效率。因此RowKey的字节长度越短越好原因三是目前操作系统大都是64位，内存8字节对齐。控制在16个字节，8字节的整数倍利用操作系统的最佳特性。

2)、RowKey散列原则：如果RowKey是按时间戳的方式递增，不要将时间放在二进制码的前面，建议将RowKey的高位作为散列字段，由程序循环生成，低位放时间字段，这样将提高数据均衡分布在每个RegionServer实现负载均衡的几率，如果没有散列字段，首字段直接是时间信息，将产生所有数据都在一个RegionServer上堆积的热点现象，这样在做数据检索的时候负载将会集中在个别RegionServer，降低查询效率。

3)、RowKey唯一原则：必须在设计上保证其唯一性。

RowKey是按照字典排序存储的，因此，设计RowKey时候，要充分利用这个排序特点，将经常一起读取的数据存储到一块，将最近可能会被访问的数据放在一块。

举个例子：如果最近写入HBase表中的数据是最可能被访问的，可以考虑将时间戳作为RowKey的一部分，由于是字段排序，所以可以使用Long.MAX\_VALUE-timeStamp作为RowKey，这样能保证新写入的数据在读取时可以别快速命中

### （★★）hbase里的热点问题遇到过么？怎么解决的？

**一、出现热点问题原因**

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；

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；

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。

**二、如何解决热点问题?**

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。

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。

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

### （★★★）Hbase接收数据，如果短时间导入数量过多的话几会被锁，该怎么办？集群数16台，高可用性的环境

参考：

通过调用HTable.setAutoFlush（false）方法可以将HTable写客户端的自动flush关闭，这样可以批量写入数据到hbase，而不是由一条put就执行一次更新，只有当put填满客户端写缓存时，才实际向Hbase服务端发起写请求。默认情况下autoflush是开启的。

### （★★）Hbase选择题：

1. 8. 下面与 Zookeeper 类似的框架是？ D  
   A Protobuf  
   B Java  
   C Kafka  
   D Chubby  
   9. 下面与 HDFS 类似的框架是？ C  
   A NTFS  
   B FAT32  
   C GFS  
   D EXT3  
   10. 下面哪些概念是 HBase 框架中使用的？ A、 C  
   A HDFS  
   B GridFS  
   C Zookeeper  
   D EXT3  
   第二部分： HBase 核心知识点  
   11. LSM 含义是？ A  
   A 日志结构合并树  
   B 二叉树  
   C 平衡二叉树  
   D 长平衡二叉树  
   12. 下面对 LSM 结构描述正确的是？ A、 C  
   A 顺序存储  
   B 直接写硬盘  
   C 需要将数据 Flush 到磁盘  
   D 是一种搜索平衡树  
   13. LSM 更能保证哪种操作的性能？ B  
   A 读  
   B 写  
   C 随机读  
   D 合并  
   14. LSM 的读操作和写操作是独立的？ A  
   A 是。  
   B 否。  
   C LSM 并不区分读和写  
   D LSM 中读写是同一种操作  
   15. LSM 结构的数据首先存储在（）。 B  
   A 硬盘上  
   B 内存中  
   C 磁盘阵列中  
   D 闪存中  
   16 HFile 数据格式中的 Data 字段用于（）。 A  
   A 存储实际的 KeyValue 数据  
   B 存储数据的起点  
   C 指定字段的长度  
   D 存储数据块的起点  
   17 HFile 数据格式中的 MetaIndex 字段用于（）。 D  
   A Meta 块的长度  
   B Meta 块的结束点  
   C Meta 块数据内容  
   D Meta 块的起始点  
   18 HFile 数据格式中的 Magic 字段用于（）。 A  
   A 存储随机数，防止数据损坏  
   B 存储数据的起点  
   C 存储数据块的起点  
   D 指定字段的长度  
   19 HFile 数据格式中的 KeyValue 数据格式，下列选项描述正确的是（）。 A、 D  
   A 是 byte[]数组  
   B 没有固定的结构  
   C 数据的大小是定长的  
   D 有固定的结构  
   20 HFile 数据格式中的 KeyValue 数据格式中 Value 部分是（）。 C  
   A 拥有复杂结构的字符串  
   B 字符串  
   C 二进制数据  
   D 压缩数据

# Flume

## 基本概念和配置和使用方式

### 1.1(★) flume的基本概念和使用方式。

基本概念：

flume是一个分布式、可靠、和高可用的海量日志采集、聚合和传输的系统。

Flume支持在日志系统中定制各类数据发送方，用于收集数据；同时，Flume提供对数据进行简单处理，并写到各种数据接受方（可定制）的能力。

使用方式：

1. netcat方式+Logger日志显示
2. flume级联方式

### 1.2(★★)flume配置方式，flume集群（问的很详细）

配置：

flume是一个可容错的、健壮的并且非常简单的流式数据框架，他只需要简单配置source、channel以及sink后，编写一条命令就可实时采集数据。 agent的三个组成部分为：source、channel和sink。 source：用于采集数据，source是产生数据流的地方，同时，source会将产生的数据传输到channel上。 channel：连接source和sink，类似队列 sink：从channel收集数据，将数据写到目标源，可以是下一个flume的source，或者是hdfs或者hbase等。

Flume的配置围绕着source、channel、sink叙述，flume的集群是做在agent上的，而非机器上。

#配置一个agent agent的名称可以自定义

#指定agent的 sources，sinks，channels

#分别指定 agent的 sources，sinks，channels 的名称 名称可以自定义

a1.sources=s1

a1.channels=c1

a1.sinks=k1

#配置 source 根据 agent的 sources 的名称来对 source 进行配置

#source 的参数是根据 不同的数据源 配置不同---在文档查找即可

#配置目录 source flume这个文件夹用于存储需要读取的文件

a1.sources.s1.type=spooldir

a1.sources.s1.spoolDir=/home/hadoop/apps/apache-flume-1.8.0-bin/flume

#配置 channel 根据 agent的 channels的名称来对 channels 进行配置

#配置内存 channel

a1.channels.c1.type=memory

#配置 sink 根据 agent的sinks 的名称来对 sinks 进行配置

#配置一个 logger sink

a1.sinks.k1.type=logger

#绑定 特别注意 source的channel 的绑定有 s,sink的 channel的绑定没有 s

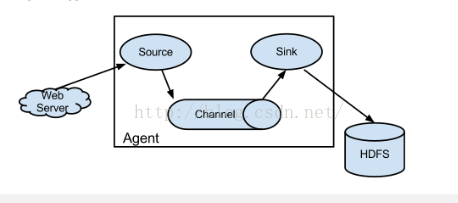
a1.sources.s1.channels=c1

a1.sinks.k1.channel=c1

### 1.3 (★★★)flume使用的时候是二次开发还是原有的配置

使用的是基于源码的二次开发

### 1.4(★)flume的框架是怎么搭建的(画图)？



## 与Kafka的整合

### 2.1 (★)数据怎么采集到Kafka，实现方式。

使用官方提供的flumeKafka插件，插件的实现方式是自定义了flume的sink，将数据从channle中取出，通过kafka的producer写入到kafka中，可以自定义分区等。

### 2. 2(★★)flume 多数据源 是到一个topic？

flume实时收集数据发送到Kafka不同topic。有一个springboot的工程，该工程对外提供一个restful的接口，应用方或其它部门调用该接口推送数据，接收到的数据里面会存放到本地磁盘的一个目录下，一天只会产生一个日志文件（根据业务划分，每一类业务一个文件）。通过使用flume实时采集日志，bizkey表示不同的业务日志的标识，按照bizkey值不同来发送到不同的topic

### 2.3(★)flume拉取数据到kafka中你们是用了几个topic

根据实际情况自由配置

### 2.4(★★)项目中Flume中间为什么用Kafka

系统架构方面的考虑：

（1）如果系统比较简单，应用场景比较单一，从简化系统的角度考虑，在满足应用需求的情况下可能只使用一个比较好。

（2）但是考虑到现有系统业务发展，为了后面的灵活扩展，在先用系统设计时留有一定的扩展性感觉更重要，

(3)可能使用Flume+kafka架构相对只使用Kafka会多占用1-2台机器做Flume日志采集，但是为了方便以后日志数据处理方式的扩展，可以采用Flume+kafka架构。

原理方面的考虑：

个人觉得原理不同，侧重点不同

Flume ：管道流方式，提供了很多的默认实现，让用户通过参数部署，及扩展API.

Kafka ：一个可持久化的分布式的消息队列。

（1）Kafka 是一个非常通用的系统。你可以有许多生产者和很多的消费者共享多个主题Topics。相比之下,Flume是一个专用工具被设计为旨在往HDFS,HBase发送数据。它对HDFS有特殊的优化，并且集成了Hadoop的安全特性。如果数据被多个系统消费的话，使用kafka；如果数据被设计给Hadoop使用，使用Flume。

（2）Flume可以使用拦截器实时处理数据。这些对数据屏蔽或者过量是很有用的。Kafka需要外部的流处理系统才能做到。

（3）Kafka和Flume都是可靠的系统,通过适当的配置能保证零数据丢失。然而，Flume不支持副本事件。于是，如果Flume代理的一个节点奔溃了，即使使用了可靠的文件管道方式，你也将丢失这些事件直到你恢复这些磁盘。如果你需要一个高可靠行的管道，那么使用Kafka是个更好的选择。

## Flume数据来源及处理方式

### 3.1 (★)flume获取数据是用什么方式

flume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

### 3.2 (★★)flume的采集处理分别有什么不同

Flume支持在日志系统中定制各类数据发送方，用于收集数据。

Flume提供对数据进行简单处理，并写到各种数据接受方（可定制）的能力 Flume提供了从console（控制台）、RPC（Thrift-RPC）、text（文件）、tail（UNIX tail）、syslog（syslog日志系统，支持TCP和UDP等2种模式），exec（命令执行）等数据源上收集数据的能力。

## Flume采集异常分析

### 1.(★★)flume管道内存，flume宕机了数据丢失怎么解决

1、Flume的channel分为很多种，可以将数据写入到文件

2、防止非首个agent宕机的方法数可以做集群或者主备

### 2.(★★)flume和kafka采集日志区别，采集日志时中间停了，怎么记录之前的日志。

Flume采集日志是通过流的方式直接将日志收集到存储层，而kafka试讲日志缓存在kafka集群，待后期可以采集到存储层。

Flume采集中间停了，可以采用文件的方式记录之前的日志，而kafka是采用offset的方式记录之前的日志。

### 3.(★★)flume的负载均衡和高可用以及数据重复丢失问题（两个人轮流问了一小时，有点懵逼了，高可用忘记了）

负载均衡：source里的event流经channel，进入sink组，在sink组内部根据负载算法（round\_robin、random）选择sink，后续可以选择不同机器上的agent实现负载均衡。

配置如下：

# Name the components on this agent

a1.sources = r1

a1.sinks = k1 k2

a1.channels = c1

# Describe/configure the source

a1.sources.r1.type = exec

a1.sources.r1.channels=c1

a1.sources.r1.command=tail -F /home/flume/xx.log

#define sinkgroups

a1.sinkgroups=g1

a1.sinkgroups.g1.sinks=k1 k2

a1.sinkgroups.g1.processor.type=load\_balance

a1.sinkgroups.g1.processor.backoff=true

a1.sinkgroups.g1.processor.selector=round\_robin

#define the sink 1

a1.sinks.k1.type=avro

a1.sinks.k1.hostname=192.168.1.112

a1.sinks.k1.port=9876

#define the sink 2

a1.sinks.k2.type=avro

a1.sinks.k2.hostname=192.168.1.113

a1.sinks.k2.port=9876

# Use a channel which buffers events in memory

a1.channels.c1.type = memory

a1.channels.c1.capacity = 1000

a1.channels.c1.transactionCapacity = 100

# Bind the source and sink to the channel

a1.sources.r1.channels = c1

a1.sinks.k1.channel = c1

a1.sinks.k2.channel=c1

2）Load balancing Log4J Appender

不同的agent处理同一个client产生的数据。

log4j.rootLogger=INFO,flume

log4j.appender.flume = org.apache.flume.clients.log4jappender.LoadBalancingLog4jAppender

log4j.appender.flume.Hosts = 192.168.1.111:41414 192.168.1.111:41414

高可用：在配置高可用的Flume NG时，需要注意一些事项。在Agent中需要绑定对应的Collector1和Collector2的IP和Port，另外，在配置Collector 节点时，需要修改当前Flume节点的配置文件，Bind的IP（或HostName）为当前节点的IP（或HostName），最后，在启动的时候，指定配置文件中的Agent的Name和配置文件的路径，否则会出错。

数据重复：Flume提供至少一次保证，事件至少被存储一次。有些场景会导致Flume最终会不只一次存储数据。

例如，RPC调用可以设置超时时间，若在超时时间内没有得到相应，及时RPC没有失败，也会被认为失败，从而引发重试。

若RPC没有失败，重试将导致相同事件再次发生。造成这数据存储端数据重复。

## 实际项目开发

### 5.1(★★)flume不采集Nginx日志，通过Logger4j采集日志，优缺点是什么？

优点：Nginx的日志格式是固定的，但是缺少sessionid，通过logger4j采集的日志是带有sessionid的，而session可以通过redis共享，保证了集群日志中的同一session落到不同的tomcat时，sessionId还是一样的，而且logger4j的方式比较稳定，不会宕机。

缺点：不够灵活，logger4j的方式和项目结合过于紧密，而flume的方式比较灵活，拔插式比较好，不会影响项目性能。

## 6. 未处理问题

### 6.1(//TODO) flume拉取文件的时候用了多少rdd

### 6.2(//TODO)flume 的实时和定时数据采集，项目和flume的解耦

### 6.3(//TODO)说说flume，flume的极限是多大（就是event的数量，当时懵逼了，就说了一般设置几十万左右，没测试过）