###### 如何配置Hadoop的HDFS支持snappy压缩：

HDFS的压缩方式的检测命令：hadoop checknative

HDFS的支持的压缩方式是haddoop的lib下的native支持的，如果想要支持需要使用支持了sanppy的hadoop jar包解压之后的lib目录替换当前即可。

###### 对目录的操作权限

对目录可读，可以ls

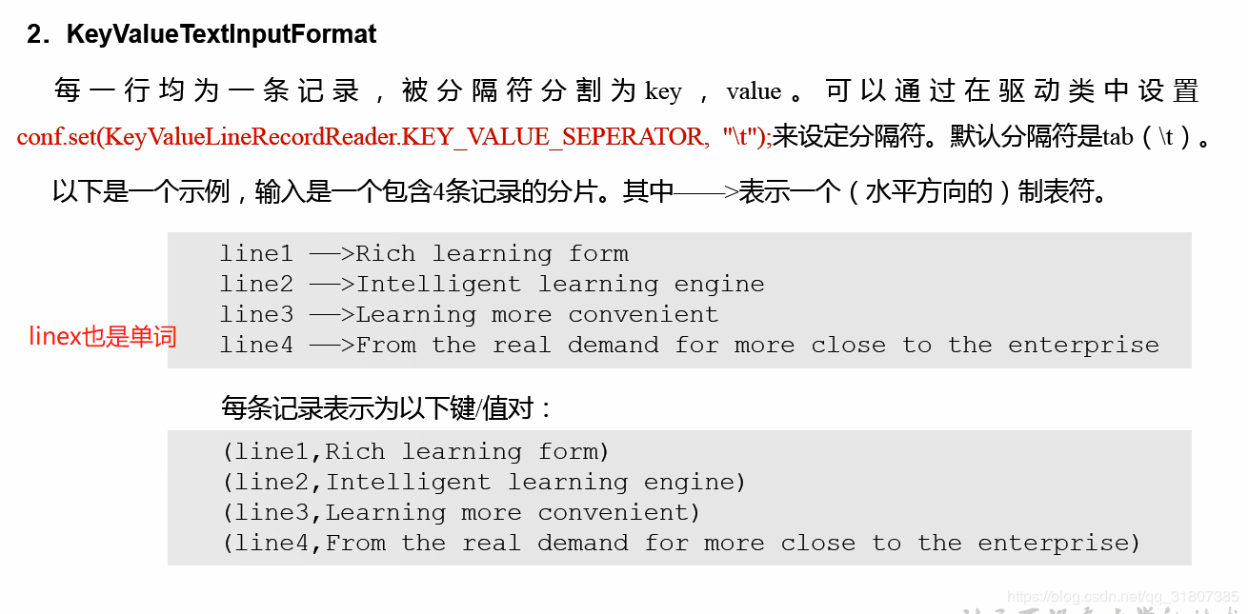
对目录可执行：可以cd进去

对目录有写权限：cd之后，创建，删除里面的文件

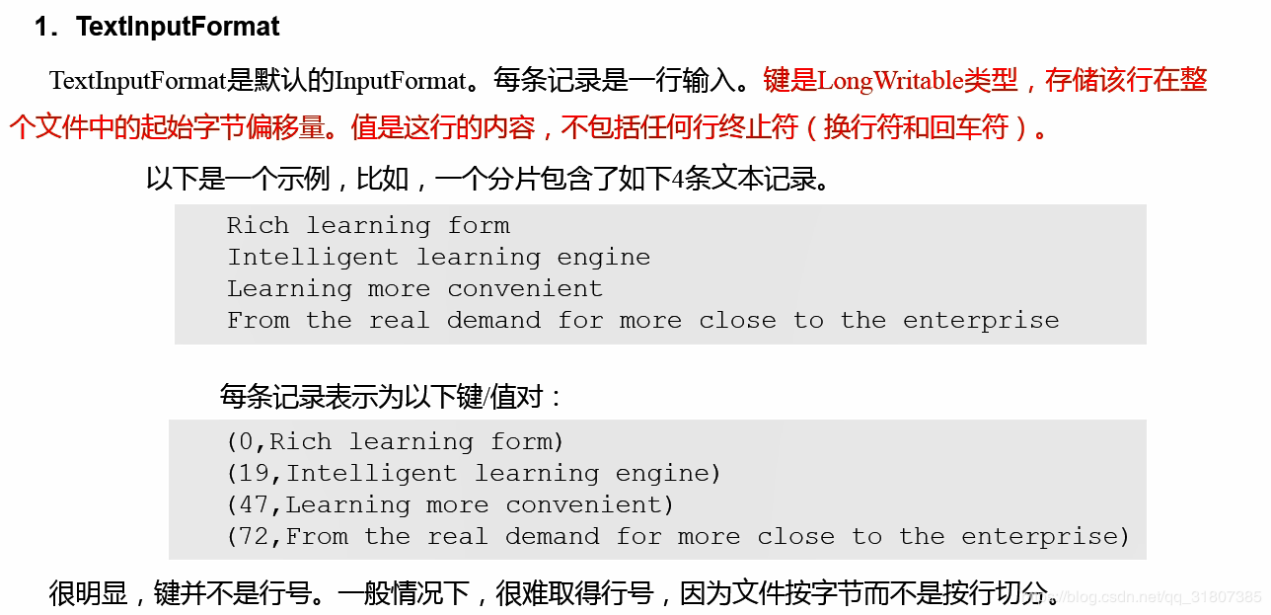
###### Map join

如果要对两张表走关联，在map阶段，将小表加入到缓存中，不通过reduce阶段，也能实现，这样干，因为没有reduce，所以就够避免数据倾斜。

###### KeyValueTextInputFormat



###### TextInputFormat



在格式化namenode的时候，一定要先删除log和data目录。格式化会产生新的集群ID，data目录中的原始数据会找不到之前的集群ID。

###### Java的多态，方法的重写

1. 父类引用指向子类对象
2. 编译时多态，方法的重载
3. 运行时多态，动态绑定

一个对象的多种形态。多指的是运行时多态，即在编译期间不能确定具体调用的是哪些方法，在运行的时候能知道。

多态，有点像是，蜘蛛侠：蜘蛛侠有一个人的形态，有一个蜘蛛超人的形态，要是世界没有危险，就是正常的人的形态，要是世界存在了危险，就有了蜘蛛超人的形态。需要根据具体的场景来判断是使用超人的形态，还是普通人的形态。在java中，一个对象可能有多个方法，有的是来自于对象本身，有的是来自于父类，在调用运行自己的方法的时候具体使用哪一个，JVM需要辨别出来，java使用的是动态绑定机制。

|  |
| --- |
| jvm在执行对象的**成员方法**时，会将这个**方法和对象的实际内存进行绑定**，然后进行调用；动态绑定和变量没有关系，只和方法有关系，也即**成员方法参与绑定，变量不参与绑定，方法看内存，变量看声明（父类的引用取父类的字段，子类的引用取子类的字段）**。具体调用哪个方法会一句该对象的实际内存。 |

案例：

|  |
| --- |
| package com.isea.kafka.producer;    public class Test {    public static void main(String[] args) {  AA aa = new BB();  System.out.println(aa.getResult());  }  }    class AA{  private int i = 10;    public int getResult(){  return i + 10;  }  }    class BB extends AA{  private int i = 20;    public int getResult(){  return i + 20;  }  }          这里注意的是在BB中有几个getResult的方法，是两个，一个是自己的，一个是从父类继承来的当前的实际内  存是BB，所以会查看BB中有没有这个方法（也即getResult方法），有，调用的就是本身的getResult方法 所  以打印的是40    -------------------------------------------------------------------------------    package com.isea.kafka.producer;    public class Test {  public static void main(String[] args) {  AA aa = new BB();    System.out.println(aa.getResult());//20  }  }    class AA{  private int i = 10;    public int getResult(){  return i + 10;  }  }    class BB extends AA{  private int i = 20;  }      这里打印的结果是20，这里只有一个getResult方式，是父类的，所以调用从父类继承来的getResult方法，  但是这个i的值是多少呢？又因为 动态绑定机制和变量没有关系，不会看i 的内存是谁，只看在哪里声明的，  所以这里会在哪里声明，在哪里使用，在父类声明，（父类中的方法中使用的变量可能在子类中声明么？不可能）    ------------------------------------------------------------------    package com.isea.kafka.producer;    public class Test {  public static void main(String[] args) {  AA aa = new BB();    System.out.println(aa.getResult());//30  }  }    class AA{  public int i = 10;    public int getResult(){  return getI() + 10;  }    public int getI() {  return i;  }  }    class BB extends AA{  public int i = 20;    @Override  public int getI() {  return i;  }  }      此时的打印结果是30，为什么呢？来一起分析一下，首先这调用了getResult方法，会发现当前对象的实际内  存中没有这个方法，在没有的情况下，一定会找父类中的getResult方法，在这个getResult方法中的getI也  是一个成员方法，只要是成员方法就会受到动态绑定机制的影响，就会看看当前对象中有没有这个方法，有这  个方法，返回i值， 这里并不是因为BB中的成员变量就是一个 i， 实际上是有两个 i ，因为从父类还继承了  一个 i，还可以访问到，所有有两个 i 。 所有这里有两个 i，因为这里是的this i，    @Override  public int getI() {  return this.i;  }  this默认是可以省略的。 |

###### Java方法重写

1. 在同一个类中
2. 只和方法的名字和参数有关（方法名相同，方法的参数不同）

|  |
| --- |
| package com.isea.kafka.producer;  public class Test {  public static void main(String[] args) {  AA aa = new BB();  test(aa);  }  }  public static void test(AA aa){  System.out.println("aaaa");  }  public static void test(BB bb){  System.out.println("bbbb");  }  class AA{  }  class BB extends AA{  }  此时打印的是aaaa，因为方法的重载只和方法的名字和参数列表有关系，（使用哪个方法，在编译期间就已经  确定了，但是和方法的重写是在运行的确定的，运行时体现了方法的多态。）这里参数一个，只能看参数的类  型，这里是参数aa是AA类型的，所以会调用第一个test方法。  以上是引用类型的重载，下面是基本类型的重载：  package com.isea.kafka.producer;  public class Test {  public static void main(String[] args) {  byte b = 10;  test(b);  }  public static void test(byte b){ // ①  System.out.println("bbbb");  }  public static void test(short d) { // ②  System.out.println("dddd");  }  public static void test(char c) { //③  System.out.println("cccc");  }  public static void test(int i) { // ④  System.out.println("iiii");  }  }  ①②③④均存在时，使用①，自动匹配为byte类型  ②③④存在时，使用②，将8位的byte转为 16位的short更快  ③④存在时，使用④，不转为 char类型是因为char类型没有负数。 |

###### 面向对象编程：

中国人：黄色的脸，黑色的眼。黄色，黑色；中国人；脸；眼； 把这首歌词，设计成对象：

|  |
| --- |
| package test;    /\*\*  \* 中国人，黄色的脸，黑色的眼  \*/    public class Chinese {  private Face face = new Face("yellow");  }    class Color {  protected String color;  }    class Face extends Color {    Eye[] eyes = new Eye[2];    public Face(String color) {  this.color = color;  eyes[0] = new Eye("black");  eyes[1] = new Eye("block");  }  }    class Eye extends Color {  public Eye(String color) {  this.color = color;  }  } |

###### 面向接口编程：

|  |
| --- |
| 什么情况下用面向接口编程 ？    ①：List<String> list = new ArrayList<String>();    ②：ArrayList<String> arrayList = new ArrayList<String>();    ①是面向接口编程，什么时候写成②写法呢？有些时候使用到了该类特定的方法的时候，需要使用到下面的写法，因  为虽然接口有特定的规范，但是这个规范不可能什么都包括，什么功能都覆盖，所以使用下面的写法。    java中的动态代理是面向接口代理的。这句话应该怎么去理解？你有接口才能做动态代理，也即谁实现了这个接口，那么该接口就可以做实现类的代理。    public class Test {  public static void main(String[] args) {  A a = new C();  }  }    interface A{    }  class B implements A{    }    class C extends B{    }      这里面构建类的时候之所以可以这样写，并不是因为 B实现了A，C继承了B，所以 C也实现了A，于是接口引用  指向实现类的地址。为什么呢？我们获取interface的个数：  System.out.println(C.class.getInterfaces().length); // 0      这里面 A a = new C(); 为什么可以呢？  因为子类对象可以代替父类对象出现，因此A a = new B() 是可以的，使用一个B的子类来代替了他，所以语  法结构这里有两个关系，一个是接口和实现类的关系，一个是父类和子类的关系。不能说C 实现了 A |

###### 代码块，静态代码块，构造方法，final关键字和类加载

JVM在使用一个类的时候，会加载这个类的相关信息到内存中，具体就是方法区的位置，什么时候会加载一个类？

①使用到该类的时候，比如，直接使用类名直接调用该类的静态方法和静态属性的时候，此时并不会调用该类的构造方法；

②在构建一个类的对象的时候，会加载该类，这个过程会依次调用该类的静态代块，普通代码块，构造方法。

|  |
| --- |
| package test;    public class User {  public static final int i = 10;  static {  System.out.println("User..static...");  }    {  System.out.println("User...");  }  public User(){  System.out.println("User()...");  }  }        package test;    public class Test {  public static void main(String[] args) {  User user = new User();  System.out.println(user.i);  }  }    /\*\*  \* User..static...  \* User...  \* User()...  \* 10  \*/    由此可见，在创建一个类的时候，会加载类，加载的顺序是：先加载静态代码块，再加载普通代码块；最后加  执行构造方法  --------------------------------------------------------------------------------------    package test;    public class User {  public static int i = 10;  static {  System.out.println("User..static...");  }    {  System.out.println("User...");  }    public User(){  System.out.println("User()...");  }  }    package test;    public class Test {  public static void main(String[] args) {  System.out.println(User.i);  }  }    /\*\*  \* User..static...  \* 10  \*/      由此可见我，User.i 即调用到类的static的属性的时候，需要加载该类到jvm的方法区，只需要执行静态代  码块。    ------------------------------------------------------------------------------------    package test;    public class User {  public static final int i = 10;  static {  System.out.println("User..static...");  }    {  System.out.println("User...");  }    public User(){  System.out.println("User()...");  }  }      package test;    public class Test {  public static void main(String[] args) {  System.out.println(User.i);  }  }    /\*\*  \* 10  \*/    由此可见：在调用类的静态的final属性的时候，不需要加载该类，因为final修饰的属性值会在方法区中的特  殊的位置来存储，和该类没有关系，并不需要加载该类，就能够访问到这个变量。 |

###### ThreadLocal 用于线程的数据共享

ThreadLocal工具类，任何的框架的底层都会使用到这个工具类，用于线程的数据共享。 Java提供了工具类（ThreadLocal），可以直接对线程的共享空间进行处理 。

对于上图的解释： 为了避免客户端请求服务器的时候，服务器频繁创建线程来处理请求，服务器在启动的时候生成多个线程放在线程池中，客户端请求先到达服务器，在到达线程池，看有没有可用的线程，如果有，拿过来一个用，然后该线程陆续在controller层，service层，DAO层，View层执行任务，如果几个层级之间有共享的数据，彼此传来传去很难受，我们可以把共享数据保存在缓存中，java中提供了ThreadLocal工具类，我们对这共享数据进行保存，获取，移除。

###### NullPointException什么情况下容易发生：

1. 自动拆箱
2. 增强for循环
3. 调用成员方法或者是成员属性的时候。

###### Hive中表的数据导入的五种方式：

|  |
| --- |
| ①：**load data**方式，如果路径是local的话，路劲里面还会继续存在，若为HDFS，加载到表中之后，数据还在。  ②：insert into [ value() , select ]  ③：as , like ；as会获得数据，like只会获得表结构  ④：**location**：先网HDFS上上传数据，在把该文件所在的文件夹的路径通过location的方式指定给表  ⑤：import：导入export出来的数据 import table table\_name from 'hdfs路径';  > load data [local] inpath '/opt/module/datas/student.txt' [overwrite] into table student [partition (partcol1=val1,…)];  （1）load data:表示加载数据  （2）local:表示从本地加载数据到hive表；否则从HDFS加载数据到hive表  （3）inpath:表示加载数据的路径  （4）overwrite:表示覆盖表中已有数据，否则表示追加  （5）into table:表示加载到哪张表  （6）student:表示具体的表  （7）partition:表示上传到指定分区 |

|  |
| --- |
| Create table if exist table\_name（id int，name string）  Row format delimited  Fields terminated by ‘\t’  Load data local inpath “\*\*\*\*” into table stu； |

博客的内容：

|  |
| --- |
| 1，创建一张表，并从本地向表中装载数据  create table if not exists stu4(id int,name string)  row format delimited  fields terminated by '\t';      > select \* from stu4;  +----------+------------+--+  | stu4.id | stu4.name |  +----------+------------+--+  +----------+------------+--+  > load data local inpath '/opt/module/hive/stu.txt' into table stu4;    > select \* from stu4;  +----------+------------+--+  | stu4.id | stu4.name |  +----------+------------+--+  | 1001 | zhangfei |  | 1002 | liubei |  | 1003 | guanyu |  | 1004 | zhaoyun |  | 1005 | caocao |  | 1006 | zhouyu |  +----------+------------+--+    2，创建一张表，并从HDFS上向表中装载数据：  create table if not exists stu5(id int,name string)  row format delimited  fields terminated by '\t';  > !sh hadoop fs -put /opt/module/hive/stu.txt /stu.txt  > select \* from stu5;  +----------+------------+--+  | stu5.id | stu5.name |  +----------+------------+--+  +----------+------------+--+  > load data inpath '/stu.txt' into table stu5;  > select \* from stu5;  +----------+------------+--+  | stu5.id | stu5.name |  +----------+------------+--+  | 1001 | zhangfei |  | 1002 | liubei |  | 1003 | guanyu |  | 1004 | zhaoyun |  | 1005 | caocao |  | 1006 | zhouyu |  +----------+------------+--+    3,加载数据覆盖表中已有的数据:  > select \* from stu5;  +----------+------------+--+  | stu5.id | stu5.name |  +----------+------------+--+  | 1001 | zhangfei |  | 1002 | liubei |  | 1003 | guanyu |  | 1004 | zhaoyun |  | 1005 | caocao |  | 1006 | zhouyu |  +----------+------------+--+    > load data local inpath '/opt/module/hive/stu2.txt' overwrite into table stu5;  > select \* from stu5;  +----------+------------+--+  | stu5.id | stu5.name |  +----------+------------+--+  | 1001 | zhangfei |  | 1002 | liubei |  | 1003 | guanyu |  +----------+------------+--+  insert：  1，创建一张分区表：insert进一些数据  > create table stu6(id int,name string)  partitioned by (month string)  row format delimited  fields terminated by '\t';    > insert into table stu6 partition(month = '12') values(1001,'zhangfei'),(1002,'liubei');    0: jdbc:hive2://hadoop108:10000> select \* from stu6;  +----------+------------+-------------+--+  | stu6.id | stu6.name | stu6.month |  +----------+------------+-------------+--+  | 1001 | zhangfei | 12 |  | 1002 | liubei | 12 |  +----------+------------+-------------+--+    2，根据select的内容插入数据：  0: jdbc:hive2://hadoop108:10000> select \* from stu6;  +----------+------------+-------------+--+  | stu6.id | stu6.name | stu6.month |  +----------+------------+-------------+--+  | 1001 | zhangfei | 12 |  | 1002 | liubei | 12 |  +----------+------------+-------------+--+    > insert overwrite table stu6 partition(month = '12') select id,name from stu\_par1 where month = '12';  0: jdbc:hive2://hadoop108:10000> select \* from stu6;  +----------+------------+-------------+--+  | stu6.id | stu6.name | stu6.month |  +----------+------------+-------------+--+  | 1001 | zhangfei | 12 |  | 1002 | liubei | 12 |  | 1003 | guanyu | 12 |  | 1004 | zhaoyun | 12 |  | 1005 | caocao | 12 |  | 1006 | zhouyu | 12 |  +----------+------------+-------------+--+  overwrite 对原来的数据进行了覆盖：    3，多表插入模式：  from stu\_par1  insert overwrite table stu6 partition(month = '11')  select id,name where month = '11'  insert overwrite table stu6 partition(month = '10')  select id,name where month = '10';    0: jdbc:hive2://hadoop108:10000> select \* from stu6;  +----------+------------+-------------+--+  | stu6.id | stu6.name | stu6.month |  +----------+------------+-------------+--+  | 1001 | zhangfei | 10 |  | 1002 | liubei | 10 |  | 1003 | guanyu | 10 |  | 1004 | zhaoyun | 10 |  | 1005 | caocao | 10 |  | 1006 | zhouyu | 10 |  | 1001 | zhangfei | 11 |  | 1002 | liubei | 11 |  | 1003 | guanyu | 11 |  | 1004 | zhaoyun | 11 |  | 1005 | caocao | 11 |  | 1006 | zhouyu | 11 |  | 1001 | zhangfei | 12 |  | 1002 | liubei | 12 |  | 1003 | guanyu | 12 |  | 1004 | zhaoyun | 12 |  | 1005 | caocao | 12 |  | 1006 | zhouyu | 12 |  +----------+------------+-------------+--+  创建表并加载数据（As Select）：  create table if not exists stu7  as select id,name from stu1;    0: jdbc:hive2://hadoop108:10000> select \* from stu7;  +----------+------------+--+  | stu7.id | stu7.name |  +----------+------------+--+  | 1001 | zhangfei |  | 1002 | liubei |  | 1003 | guanyu |  | 1004 | zhaoyun |  | 1005 | caocao |  | 1006 | zhouyu |  +----------+------------+--+  6 rows selected (0.149 seconds)  location：  1，HDFS的路径上有如下的内容：/ex 该目录下有stu.txt文件  create external table stu\_ex2(id int,name string)  row format delimited  fields terminated by '\t'  location '/ex';    0: jdbc:hive2://hadoop108:10000> select \* from stu\_ex2;  +-------------+---------------+--+  | stu\_ex2.id | stu\_ex2.name |  +-------------+---------------+--+  | 1001 | zhangfei |  | 1002 | liubei |  | 1003 | guanyu |  | 1004 | zhaoyun |  | 1005 | caocao |  | 1006 | zhouyu |  +-------------+---------------+--+  6 rows selected (0.093 seconds)  import：  import导入的 数据必须是export导出的数据：  1，将数据导出到HDFS上：  export table stu1 to '/export/data/stu1'    0: jdbc:hive2://hadoop108:10000> !sh hadoop fs -ls /export/data/stu1  Found 2 items  -rwxr-xr-x 3 isea supergroup 1329 2018-12-01 19:38 /export/data/stu1/\_metadata  drwxr-xr-x - isea supergroup 0 2018-12-01 19:38 /export/data/stu1/data    发现stu1目录下多了两个文件，数据存储在data中    2，将HDFS上的数据导入到stu8；  0: jdbc:hive2://hadoop108:10000> show tables;  +------------------------+--+  | tab\_name |  +------------------------+--+  | stu1 |  | stu2 |  | stu3 |  | stu4 |  | stu5 |  | stu6 |  | stu7 |  | stu\_ex1 |  | stu\_ex2 |  | stu\_par1 |  | stu\_par2 |  | values\_\_tmp\_\_table\_\_1 |  +------------------------+--+  0: jdbc:hive2://hadoop108:10000> import table stu8 from '/export/data/stu1';    0: jdbc:hive2://hadoop108:10000> select \* from stu8;  +----------+------------+--+  | stu8.id | stu8.name |  +----------+------------+--+  | 1001 | zhangfei |  | 1002 | liubei |  | 1003 | guanyu |  | 1004 | zhaoyun |  | 1005 | caocao |  | 1006 | zhouyu |  +----------+------------+ |

###### Hive的数据类型

博客内容：

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Hive的数据类型主要有两类，一类是基本数据类型，一类是集合数据类型： 基本数据类型：  |  |  |  |  | | --- | --- | --- | --- | | Hive数据类型 | Java数据类型 | 长度 | 例子 | | TINYINT | byte | 1byte有符号整数 | 20 | | SMALINT | short | 2byte有符号整数 | 20 | | INT | int | 4byte有符号整数 | 20 | | BIGINT | long | 8byte有符号整数 | 20 | | BOOLEAN | boolean | 布尔类型，true或者false | TRUE  FALSE | | FLOAT | float | 单精度浮点数 | 3.14159 | | DOUBLE | double | 双精度浮点数 | 3.14159 | | STRING | string | 字符系列。可以指定字符集。可以使用单引号或者双引号。 | ‘now is the time’ “for all good men” | | TIMESTAMP |  | 时间类型 |  | | BINARY |  | 字节数组 |  |   集合数据类型：  |  |  |  | | --- | --- | --- | | 数据类型 | 描述 | 语法示例 | | STRUCT | 和c语言中的struct类似，都可以通过“点”符号访问元素内容。例如，如果某个列的数据类型是STRUCT{first STRING, last STRING},那么第1个元素可以通过字段.first来引用。 | struct()  例如struct<street:string, city:string> | | MAP | MAP是一组键-值对元组集合，使用数组表示法可以访问数据。例如，如果某个列的数据类型是MAP，其中键->值对是’first’->’John’和’last’->’Doe’，那么可以通过字段名[‘last’]获取最后一个元素 | map()  例如map<string, int> | | ARRAY | 数组是一组具有相同类型和名称的变量的集合。这些变量称为数组的元素，每个数组元素都有一个编号，编号从零开始。例如，数组值为[‘John’, ‘Doe’]，那么第2个元素可以通过数组名[1]进行引用。 | Array()  例如array<string> |   规律总结：  array和struct在提取值的时候：只会单纯的提取值，值前面的是字段的信息  map在提取值的时候：会加key和value都提取出来，key 和 value之外的 才是字段的信息  推荐，值之间的分隔符号使用 “ ，” 集合之间值的分隔符号使用 “\_” ，key 和 value之间的值用来分隔两者。 案例实操： {  "name": "songsong",  "friends": ["bingbing" , "lili"] , //列表Array,  "children": { //键值Map,  "xiao song": 18 ,  "xiaoxiao song": 19  }  "address": { //结构Struct,  "street": "hui long guan" ,  "city": "beijing"  }  }  基于上述数据结构，我们在Hive里创建对应的表，并导入数据。  转化之后的数据形式：  songsong,bingbing\_lili,xiao song:18\_xiaoxiao song:19,hui long guan\_beijing  hive (default)> create table person(  > name string,  > friends array<string>,  > children map<string,int>,  > address struct<street:string,city:string>  > )  > row format delimited  > fields terminated by ','  > collection items terminated by '\_'  > map keys terminated by ':';  OK  Time taken: 1.182 seconds  hive (default)> load data local inpath "/opt/module/hive/pson.data" into table default.person;  Loading data to table default.person  Table default.person stats: [numFiles=1, totalSize=75]  OK  Time taken: 1.2 seconds  hive (default)> select \* from person;  OK  person.name person.friends person.children person.address  songsong ["bingbing","lili"] {"xiao song":18,"xiaoxiao song":19} {"street":"hui long guan","city":"beijing"}  所有的中间过程在subline中写好，然后在粘贴到CLI中去：  {  "name": "songsong",  "friends": ["bingbing" , "lili"] , //列表Array,  "children": { //键值Map,  "xiao song": 18 ,  "xiaoxiao song": 19  }  "address": { //结构Struct,  "street": "hui long guan" ,  "city": "beijing"  }  }  songsong,bingbing\_lili,xiao song:18\_xiaoxiao song:19,hui long guan\_beijing  创建表  create table person(  name string,  friends array<string>,  children map<string,int>,  address struct<street:string,city:string>  )  row format delimited  fields terminated by ','  collection items terminated by '\_'  map keys terminated by ':';  load data local inpath "/opt/module/datas/pson.data" into table default.person;  另外一种方案：  hive (default)> create table human(  > name string,  > friends array<string>,  > children map<string,int>,  > address map<string,string>  > )  > row format delimited  > fields terminated by ','  > collection items terminated by '\_'  > map keys terminated by ':';  OK  Time taken: 0.303 seconds  hive (default)> load data local inpath "/opt/module/hive/human.data" into table default.human;  Loading data to table default.human  Table default.human stats: [numFiles=1, totalSize=87]  OK  Time taken: 0.356 seconds  hive (default)> select \* from human;  OK  human.name human.friends human.children human.address  songsong ["bingbing","lili"] {"xiao song":18,"xiaoxiao song":19} {"street":"hui long guan","city":"beijing"}  Time taken: 0.093 seconds, Fetched: 1 row(s)  hive (default)>  我的草稿纸入下：  {  "name": "songsong",  "friends": ["bingbing" , "lili"] , //列表Array,  "children": { //键值key  "xiao song": 18 ,  "xiaoxiao song": 19  }  "address": { //keys,  "street": "hui long guan" ,  "city": "beijing"  }  }  songsong,bingbing\_lili,xiao song:18\_xiaoxiao song:19,street:hui long guan\_city:beijing  创建表的第二种方式：  create table human(  name string,  friends array<string>,  children map<string,int>,  address map<string,string>  )  row format delimited  fields terminated by ','  collection items terminated by '\_'  map keys terminated by ':';  load data local inpath "/opt/module/hive/human.data" into table default.human; |

###### 分布式zookeeper的部署

常用的命令：ls ，delete ，get，create，最牛逼的是help

博客内容：

|  |
| --- |
| 安装zookeeper：  1，将事先准备好的zookeeper拷贝到linux文件夹下：  [isea@hadoop101 software]$ ll  总用量 408420  -rw-rw-r--. 1 isea isea 197657687 11月 14 17:55 hadoop-2.7.2.tar.gz  -rw-rw-r--. 1 isea isea 185515842 11月 14 17:55 jdk-8u144-linux-x64.tar.gz  -rw-rw-r--. 1 isea isea 35042811 11月 28 16:39 zookeeper-3.4.10.tar.gz    2，将其解压到module目录下  [isea@hadoop101 software]$ tar -zxvf zookeeper-3.4.10.tar.gz -C /opt/module/  [isea@hadoop101 module]$ ll  总用量 16  drwxrwxr-x. 3 isea isea 4096 11月 28 15:05 HA  drwxr-xr-x. 11 isea isea 4096 11月 28 15:03 hadoop-2.7.2  drwxr-xr-x. 8 isea isea 4096 7月 22 2017 jdk1.8.0\_144  drwxr-xr-x. 10 isea isea 4096 3月 23 2017 zookeeper-3.4.10    3，在/opt/module/zookeeper-3.4.10路径下创建zkData文件夹，修改配置文件，添加dirData路径，如下：  [isea@hadoop101 zookeeper-3.4.10]$ mkdir zkData    [isea@hadoop101 conf]$ cp zoo\_sample.cfg zoo.cfg  [isea@hadoop101 conf]$ vim zoo.cfg  dataDir=/opt/module/zookeeper-3.4.10/zkData    [isea@hadoop101 conf]$ pwd  /opt/module/zookeeper-3.4.10/conf  部署zookeeper集群：  4，在zkData目录下创建myid文件，并在该文件中写上该机器的编号  [isea@hadoop101 zkData]$ vim myid  1    5，在zoo.cfg中添加如下的内容：  [isea@hadoop101 conf]$ vim zoo.cfg    server.1=hadoop101:2888:3888  server.2=hadoop102:2888:3888  server.3=hadoop103:2888:3888      6，同步zookeeper到其他机器，并修改myid  [isea@hadoop101 module]$ pwd  /opt/module  [isea@hadoop101 module]$ xsync zookeeper-3.4.10/    [isea@hadoop102 zkData]$ cat myid  2      [isea@hadoop103 zkData]$ cat myid  3    6，启动集群，分别在各个机器上启动zookeeper：  [isea@hadoop101 zookeeper-3.4.10]$ bin/zkServer.sh start  ZooKeeper JMX enabled by default  Using config: /opt/module/zookeeper-3.4.10/bin/../conf/zoo.cfg  Starting zookeeper ... STARTED  [isea@hadoop101 zookeeper-3.4.10]$ bin/zkServer.sh status  ZooKeeper JMX enabled by default  Using config: /opt/module/zookeeper-3.4.10/bin/../conf/zoo.cfg  Error contacting service. It is probably not running.  [isea@hadoop101 zookeeper-3.4.10]$ bin/zkServer.sh status  ZooKeeper JMX enabled by default    这个时候发现集群并没有开始工作，所以去启动第二台机器：  [isea@hadoop102 zookeeper-3.4.10]$ bin/zkServer.sh start  ZooKeeper JMX enabled by default  Using config: /opt/module/zookeeper-3.4.10/bin/../conf/zoo.cfg  Starting zookeeper ... STARTED  [isea@hadoop102 zookeeper-3.4.10]$ bin/zkServer.sh status  ZooKeeper JMX enabled by default  Using config: /opt/module/zookeeper-3.4.10/bin/../conf/zoo.cfg  Mode: leader    此时在去查看第一台机器：  [isea@hadoop101 zookeeper-3.4.10]$ bin/zkServer.sh status  ZooKeeper JMX enabled by default  Using config: /opt/module/zookeeper-3.4.10/bin/../conf/zoo.cfg  Mode: follower    此时，集群已经开始工作。    [isea@hadoop103 zookeeper-3.4.10]$ bin/zkServer.sh start  ZooKeeper JMX enabled by default  Using config: /opt/module/zookeeper-3.4.10/bin/../conf/zoo.cfg  Starting zookeeper ... STARTED  [isea@hadoop103 zookeeper-3.4.10]$ bin/zkServer.sh status  ZooKeeper JMX enabled by default  Using config: /opt/module/zookeeper-3.4.10/bin/../conf/zoo.cfg  Mode: follower     集群操作：  7，启动客户端：  [isea@hadoop101 zookeeper-3.4.10]$ bin/zkCli.sh  Connecting to localhost:2181  WatchedEvent state:SyncConnected type:None path:null  [zk: localhost:2181(CONNECTED) 0] ls /  [zookeeper]  命令基本语法  功能描述  help  显示所有操作命令  ls path [watch]  使用 ls 命令来查看当前znode中所包含的内容  ls2 path [watch]  查看当前节点数据并能看到更新次数等数据  create  普通创建  -s  含有序列  -e  临时（重启或者超时消失）  get path [watch]  获得节点的值  set  设置节点的具体值  stat  查看节点状态  delete  删除节点  rmr  递归删除节点 |

###### Flume实时读取目录文件到HDFS：、

启动一个Flume：

bin/flume-ng agent --conf conf/ --name a3 --conf-file jobs/flume-dir-hdfs.cof

玩Flume玩的就是配置： 监控文件的变化，使用spolldir 。 **常用的还有exec，tailDir**

|  |
| --- |
| a3.sinks.k3.hdfs.batchSize = 100 表示积攒多少个Event才flush到HDFS一次，这里的sink会从channel不断的获取到event，但是不能来一个event就把这个event上传到HDFS中的文件中假如为文件A中去，因为这样做效率太低，所以这里设置100个event在把数据（event就是数据）写到HDFS中的文件A中去；而rollInternal=60 和rollSize表示的是HDFS上的文件夹达到了什么标准重新生成，而且设置rollCount=0 表示是否生成新文件和sink 过来的event的数量没有关系；上文所说的文件A 就是这里说的生成的新文件。event就是玩具，而新文件的生成就是箱子，玩具要一批一批放入箱子，是否使用一个新的箱子来装玩具，和玩具的数量没有关系，只和时间，和整个箱子的重量（理解为文件的大小）有关系。 |

博客内容：

|  |
| --- |
| 使用Flume监听整个目录的文件,监听 /opt/module/flume/upload 文件下是否有新文件的加入，做到实时读取该目录下的文件到HDFS上， 分析如下：  案例实操： 1，配置flume-dir-hdfs.cof  vim flume-dir-hdfs.cof  a3.sources = r3  a3.sinks = k3  a3.channels = c3  # Describe/configure the source  a3.sources.r3.type = spooldir  a3.sources.r3.spoolDir = /opt/module/flume/upload  #将上传的文件添加后缀  a3.sources.r3.fileSuffix = .COMPLETED  a3.sources.r3.fileHeader = true  #忽略所有以.tmp结尾的文件，不上传，^ 表示匹配开头  a3.sources.r3.ignorePattern = ([^ ]\*\.tmp)  # Describe the sink  a3.sinks.k3.type = hdfs  a3.sinks.k3.hdfs.path = hdfs://hadoop108:9000/flume/upload/%Y%m%d/%H  #上传文件的前缀  a3.sinks.k3.hdfs.filePrefix = upload-  #是否按照时间滚动文件夹  a3.sinks.k3.hdfs.round = true  #多少时间单位创建一个新的文件夹  a3.sinks.k3.hdfs.roundValue = 1  #重新定义时间单位  a3.sinks.k3.hdfs.roundUnit = hour  #是否使用本地时间戳  a3.sinks.k3.hdfs.useLocalTimeStamp = true  #积攒多少个Event才flush到HDFS一次  a3.sinks.k3.hdfs.batchSize = 100  #设置文件类型，可支持压缩  a3.sinks.k3.hdfs.fileType = DataStream  #多久生成一个新的文件  a3.sinks.k3.hdfs.rollInterval = 60  #设置每个文件的滚动大小大概是128M  a3.sinks.k3.hdfs.rollSize = 134217700  #文件的滚动与Event数量无关  a3.sinks.k3.hdfs.rollCount = 0  # Use a channel which buffers events in memory  a3.channels.c3.type = memory  a3.channels.c3.capacity = 1000  a3.channels.c3.transactionCapacity = 100  # Bind the source and sink to the channel  a3.sources.r3.channels = c3  a3.sinks.k3.channel = c3  关于上述配置中sink相关的设置的一个解释：  a3.sinks.k3.hdfs.batchSize = 100 表示积攒多少个Event才flush到HDFS一次，这里的sink会从channel不断的获取到event，但是不能来一个event就把这个event上传到HDFS中的文件中假如为文件A中去，因为这样做效率太低，所以这里设置100个event在把数据（event就是数据）写到HDFS中的文件A中去；而rollInternal=60 和rollSize表示的是HDFS上的文件夹达到了什么标准重新生成，而且设置rollCount=0 表示是否生成新文件和sink 过来的event的数量没有关系；上文所说的文件A 就是这里说的生成的新文件。event就是玩具，而新文件的生成就是箱子，玩具要一批一批放入箱子，是否使用一个新的箱子来装玩具，和玩具的数量没有关系，只和时间，和整个箱子的重量（理解为文件的大小）有关系。  2，启动HDFS（因为要获取HDFS的客户端从而向HDFS写入数据），启动监控文件夹的命令  bin/flume-ng agent --conf conf/ --name a3 --conf-file jobs/flume-dir-hdfs.cof  特点：  1) 不要在监控目录中创建并持续修改文件  2) 上传完成的文件会以.COMPLETED结尾  3) 被监控文件夹每500毫秒扫描一次文件变动  4) 这里的upload文件夹要存在  3，在/opt/module/flume/创建upload文件夹，并在该文件夹下添加文件。观察upload文件夹下的变化情况，ll  mkdir upload  touch isea.txt  touch iseayou.sh  touch iseayou.sql  检查数据：  [isea@hadoop101 upload]$ ll  总用量 0  -rw-rw-r--. 1 isea isea 0 12月 11 22:20 isea.txt.COMPLETED  -rw-rw-r--. 1 isea isea 0 12月 11 22:20 iseayou.txt.COMPLETED |

###### Kafka的安装和配置

Kafka依赖于zookeeper，所以server.properties 中要有zookeeper.connect 的配置项。

博客内容：

|  |
| --- |
| 安装步骤：  1，下载kafka，并将其上传到指定的目录：    地址：http://kafka.apache.org/downloads.html  [isea@hadoop108 software]$ ll  -rw-r--r-- 1 isea isea 42136632 12月 29 2017 kafka\_2.11-0.11.0.2.tgz    2，解压到指定的目录，并修改一个友好的名字：  [isea@hadoop108 software]$ tar -zxvf kafka\_2.11-0.11.0.2.tgz -C /opt/module/  [isea@hadoop108 opt]$ cd module/  [isea@hadoop108 module]$ ll  drwxr-xr-x 6 isea isea 4096 11月 11 2017 kafka\_2.11-0.11.0.2  drwxr-xr-x. 11 isea isea 4096 11月 27 00:19 zookeeper-3.4.10    [isea@hadoop108 module]$ mv kafka\_2.11-0.11.0.2/ kafka  [isea@hadoop108 module]$ ll    drwxr-xr-x 6 isea isea 4096 11月 11 2017 kafka  drwxr-xr-x. 11 isea isea 4096 11月 27 00:19 zookeeper-3.4.10    3，在kafka目录下创建logs文件夹：  [isea@hadoop108 module]$ cd kafka/  [isea@hadoop108 kafka]$ ll  总用量 52  drwxr-xr-x 3 isea isea 4096 11月 11 2017 bin  drwxr-xr-x 2 isea isea 4096 11月 11 2017 config  drwxr-xr-x 2 isea isea 4096 12月 9 23:04 libs  -rw-r--r-- 1 isea isea 28824 11月 11 2017 LICENSE  -rw-r--r-- 1 isea isea 336 11月 11 2017 NOTICE  drwxr-xr-x 2 isea isea 4096 11月 11 2017 site-docs  [isea@hadoop108 kafka]$ mkdir logs  [isea@hadoop108 kafka]$ ll  总用量 56  drwxr-xr-x 3 isea isea 4096 11月 11 2017 bin  drwxr-xr-x 2 isea isea 4096 11月 11 2017 config  drwxr-xr-x 2 isea isea 4096 12月 9 23:04 libs  -rw-r--r-- 1 isea isea 28824 11月 11 2017 LICENSE  drwxrwxr-x 2 isea isea 4096 12月 9 23:05 logs  -rw-r--r-- 1 isea isea 336 11月 11 2017 NOTICE  drwxr-xr-x 2 isea isea 4096 11月 11 2017 site-docs    4，配置server.properties    broker.id=8  delete.topic.enable=true  log.dirs=/opt/module/kafka/logs  zookeeper.connect=hadoo108:2181,hadoop109:2181,hadoop110:2181    5，此处环境变量可根据自己的需要配置    6，分发配置kafka，在各个机器上修改id  broker.id=9，broker.id=10    启动：  1，接下来是启动kafka，在启动kafka之前，先启动zookeeper，三个节点都需要执行下面的命令：    /opt/module/zookeeper-3.4.10/bin/zkServer.sh start    2，启动kafka：三个节点在kafka目录下都需要执行：  bin/kafka-server-start.sh config/server.properties &    或者使用下面的命令  bin/kafka-server-start.sh -daemon config/server.properties  关闭：  三台机器都需要执行：  bin/kafka-server-stop.sh stop |

###### Linux环境下安装Scala：

|  |
| --- |
| 步骤如下：  下载对应的scala的安装软件.scala-2.11.8.tgz  通过远程登录工具，将安装软件上传到对应的linux系统（xshell5  xftp5）  解压，scala，在/usr/local/下创建scala目录，来存放scala，  具体如下：  [isea@redis99 software]$ sudo mkdir /usr/local/scala  [isea@redis99 software]$ tar -zxvf scala-2.11.8.tgz  [isea@redis99 software]$ cd /usr/local/scala/  [isea@redis99 scala]$ ll  总用量 4  drwxrwxr-x. 6 isea isea 4096 3月 4 2016 scala-2.11.8      [isea@redis99 scala-2.11.8]$ sudo vim /etc/profile    #JAVA\_HOME  JAVA\_HOME=/opt/module/jdk1.8.0\_144  export PATH=$PATH:$JAVA\_HOME/bin    export PATH=$PATH:/usr/local/scala/scala-2.11.8/bin    [isea@redis99 scala-2.11.8]$ source /etc/profile  [isea@redis99 scala-2.11.8]$ scala  Welcome to Scala 2.11.8 (Java HotSpot(TM) 64-Bit Server VM, Java 1.8.0\_144).  Type in expressions for evaluation. Or try :help. |

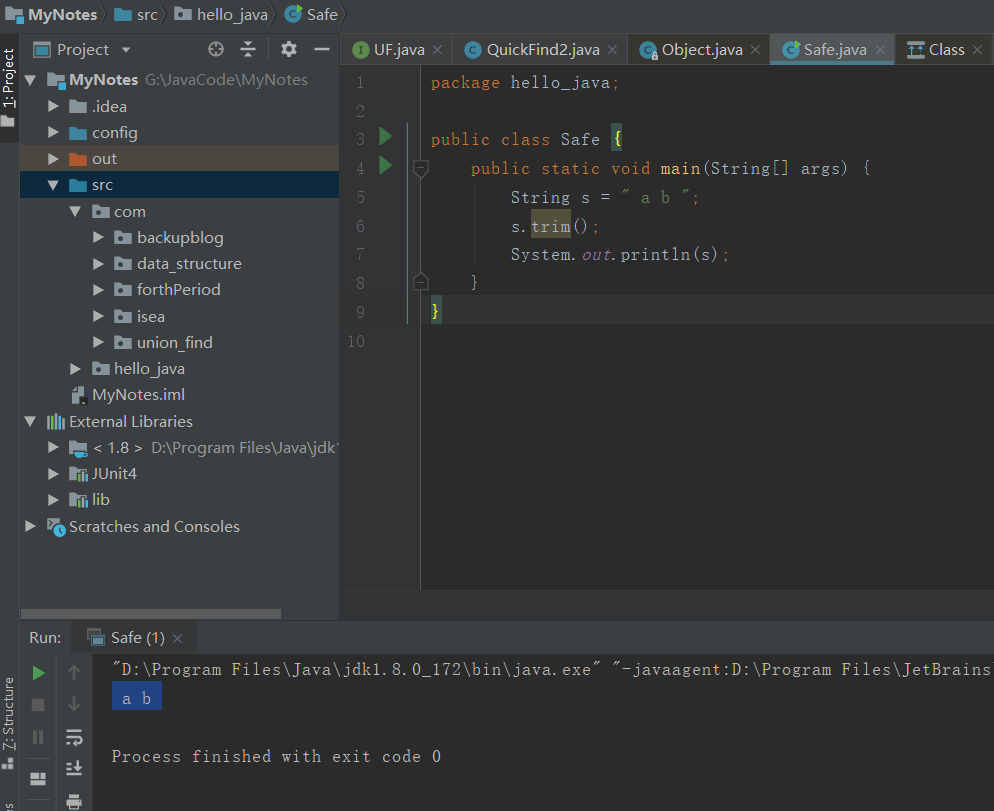
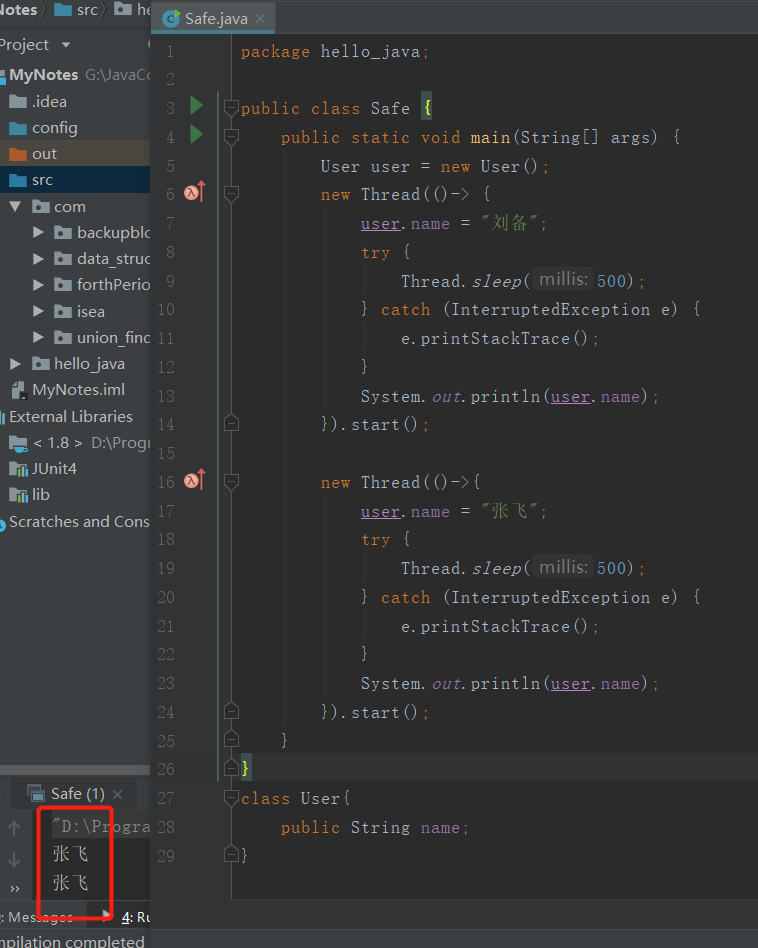
###### 自定义InputFormat

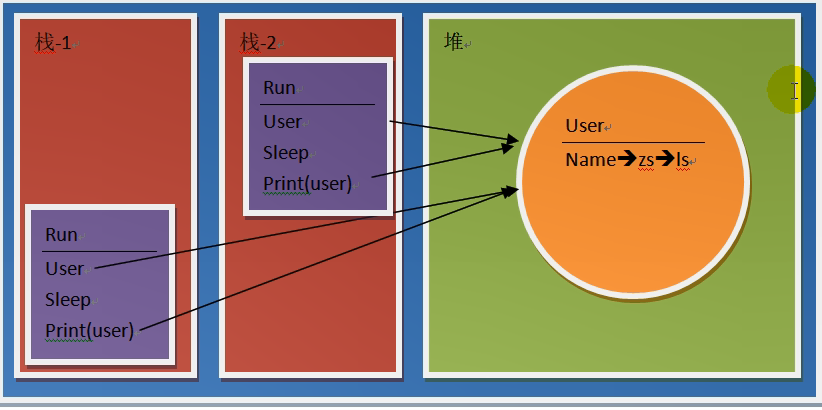
将多个小文件合并成一个SequenceFile文件（SequenceFile文件是Hadoop用来存储二进制形式的key-value对的文件格式），SequenceFile里面存储着多个文件，存储的形式为文件路径+名称为key，文件内容为value。

假设原来有三个小文件，有就有三个切片，使用三个MapTask来将其变成sequencefile文件之后，变成了一个文件，于是就启动了一个maptask进行后续的处理。

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| |  | | --- | | package com.isea.wholeformat;    import org.apache.hadoop.fs.Path;  import org.apache.hadoop.io.BytesWritable;  import org.apache.hadoop.io.Text;  import org.apache.hadoop.mapreduce.InputSplit;  import org.apache.hadoop.mapreduce.JobContext;  import org.apache.hadoop.mapreduce.RecordReader;  import org.apache.hadoop.mapreduce.TaskAttemptContext;  import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;    import java.io.IOException;    public class WholeFileInputFormat extends FileInputFormat<Text, BytesWritable> {  @Override  protected boolean isSplitable(JobContext context, Path filename) {  return false;  }    @Override  public RecordReader<Text, BytesWritable> createRecordReader(InputSplit split, TaskAttemptContext context) throws IOException, InterruptedException {  WholeRecordReader recordReader = new WholeRecordReader();  recordReader.initialize(split,context);  return recordReader;  }  } |  |  | | --- | | package com.isea.wholeformat;    import org.apache.hadoop.conf.Configuration;  import org.apache.hadoop.fs.FSDataInputStream;  import org.apache.hadoop.fs.FileSystem;  import org.apache.hadoop.fs.Path;  import org.apache.hadoop.io.BytesWritable;  import org.apache.hadoop.io.IOUtils;  import org.apache.hadoop.io.Text;      import org.apache.hadoop.mapreduce.InputSplit;  import org.apache.hadoop.mapreduce.RecordReader;  import org.apache.hadoop.mapreduce.TaskAttemptContext;  import org.apache.hadoop.mapreduce.lib.input.FileSplit;    import java.io.IOException;    public class WholeRecordReader extends RecordReader<Text, BytesWritable> {  FileSplit split;  Configuration configuration;  Text k = new Text();  BytesWritable v = new BytesWritable();  boolean isProcess = true;    @Override  public void initialize(InputSplit split, TaskAttemptContext context) throws IOException, InterruptedException {  // 初始化操作  this.split = (FileSplit) split;  configuration = context.getConfiguration();  }    @Override  public boolean nextKeyValue() throws IOException, InterruptedException {  // 核心业务逻辑    if (isProcess){  byte[] buf = new byte[(int) split.getLength()];    // 1,获取fs对象  Path path = split.getPath();  FileSystem fs = path.getFileSystem(configuration);    // 2,获取输入流  FSDataInputStream fis = fs.open(path);    // 3,拷贝  IOUtils.readFully(fis,buf,0,buf.length);    // 4,封装v  v.set(buf,0, buf.length);    // 5，封装key  k.set(path.toString());    // 6,关闭资源  IOUtils.closeStream(fis);    isProcess = false;  return true;  }    return false;  }    @Override  public Text getCurrentKey() throws IOException, InterruptedException {  return k;  }    @Override  public BytesWritable getCurrentValue() throws IOException, InterruptedException {  return v;  }    @Override  public float getProgress() throws IOException, InterruptedException {  return 0;  }    @Override  public void close() throws IOException {    }  } |  |  | | --- | | package com.isea.wholeformat;    import org.apache.hadoop.io.BytesWritable;  import org.apache.hadoop.io.Text;  import org.apache.hadoop.mapreduce.Mapper;    import java.io.IOException;    public class SequenceFileMapper extends Mapper<Text, BytesWritable,Text,BytesWritable> {  @Override  protected void map(Text key, BytesWritable value, Context context) throws IOException, InterruptedException {  context.write(key,value );  }  } |  |  | | --- | | package com.isea.wholeformat;    import org.apache.hadoop.io.BytesWritable;  import org.apache.hadoop.io.Text;  import org.apache.hadoop.mapreduce.Reducer;    import java.io.IOException;    public class SequenceFileReducer extends Reducer<Text, BytesWritable,Text,BytesWritable> {  @Override  protected void reduce(Text key, Iterable<BytesWritable> values, Context context) throws IOException, InterruptedException {  // 循环写入  for (BytesWritable bytesWritable : values) {  context.write(key,bytesWritable );  }  }  } |  |  | | --- | | package com.isea.wholeformat;    import org.apache.hadoop.conf.Configuration;  import org.apache.hadoop.fs.Path;  import org.apache.hadoop.io.BytesWritable;  import org.apache.hadoop.io.Text;  import org.apache.hadoop.mapreduce.Job;  import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;  import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;  import org.apache.hadoop.mapreduce.lib.output.SequenceFileOutputFormat;    import java.io.IOException;    public class SequenceFileDriver {  public static void main(String[] args) throws IOException, ClassNotFoundException, InterruptedException {  args = new String[] {"G:/input","G:/output3"};    // 1,获取job对象  Configuration configuration = new Configuration();  Job job = Job.getInstance(configuration);    // 1.5设置输入的InputFormat  job.setInputFormatClass(WholeFileInputFormat.class);  // 1.6 设置输出的outputFormat  job.setOutputFormatClass(SequenceFileOutputFormat.class);    // 2，设置jar的路径  job.setJarByClass(SequenceFileDriver.class);    // 3，关联Mapper和Reducer  job.setMapperClass(SequenceFileMapper.class);  job.setReducerClass(SequenceFileReducer.class);    // 4，设置Mapper输出的key和value类型  job.setMapOutputKeyClass(Text.class);  job.setMapOutputValueClass(BytesWritable.class);  // 5，设置最终的输出类型  job.setOutputKeyClass(Text.class);  job.setOutputValueClass(BytesWritable.class);    // 6, 设置输入输出的路径  FileInputFormat.setInputPaths(job,new Path(args[0]));  FileOutputFormat.setOutputPath(job,new Path(args[1]));    // 7, 提交job  boolean result = job.waitForCompletion(true);  System.exit(result ? 0 : 1);    }  } |   number of splits:3 |

小白问问，大姥解答

1. Final修饰类，和final修饰变量各表示什么？
   1. Final修饰类，表示这个类不能被继承，
   2. final修饰变量表示这个变量不能够在被改变。
2. 中文是全角的，英文是半角的。
3. 下面的这个题目中的打印结果：
   1. 
4. 给出下面代码的运行结果，并作出解释
   1. 
   2. 解释如下：

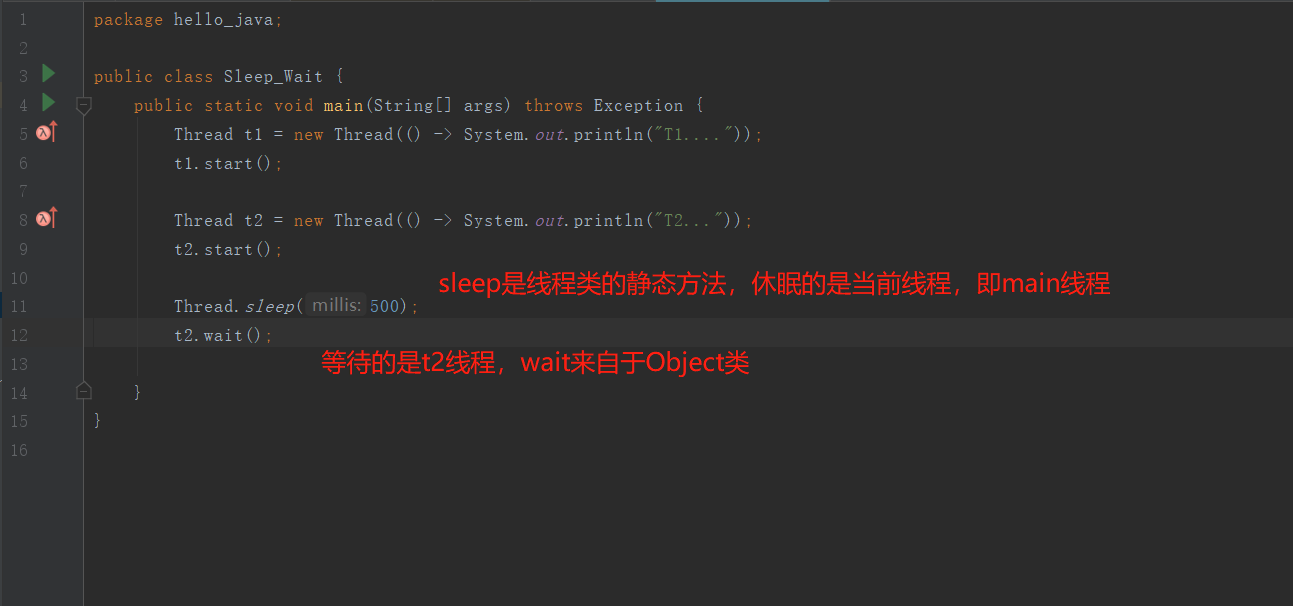


这里，每一个线程都会创建一个栈，线程1执行修改name之后，之后当前的线程休眠，然后线程2执行，再次修改name，之后线程1醒来，打印出name的值，但是这个时候打印的是线程2修改之后的值，这就是线程安全问题。

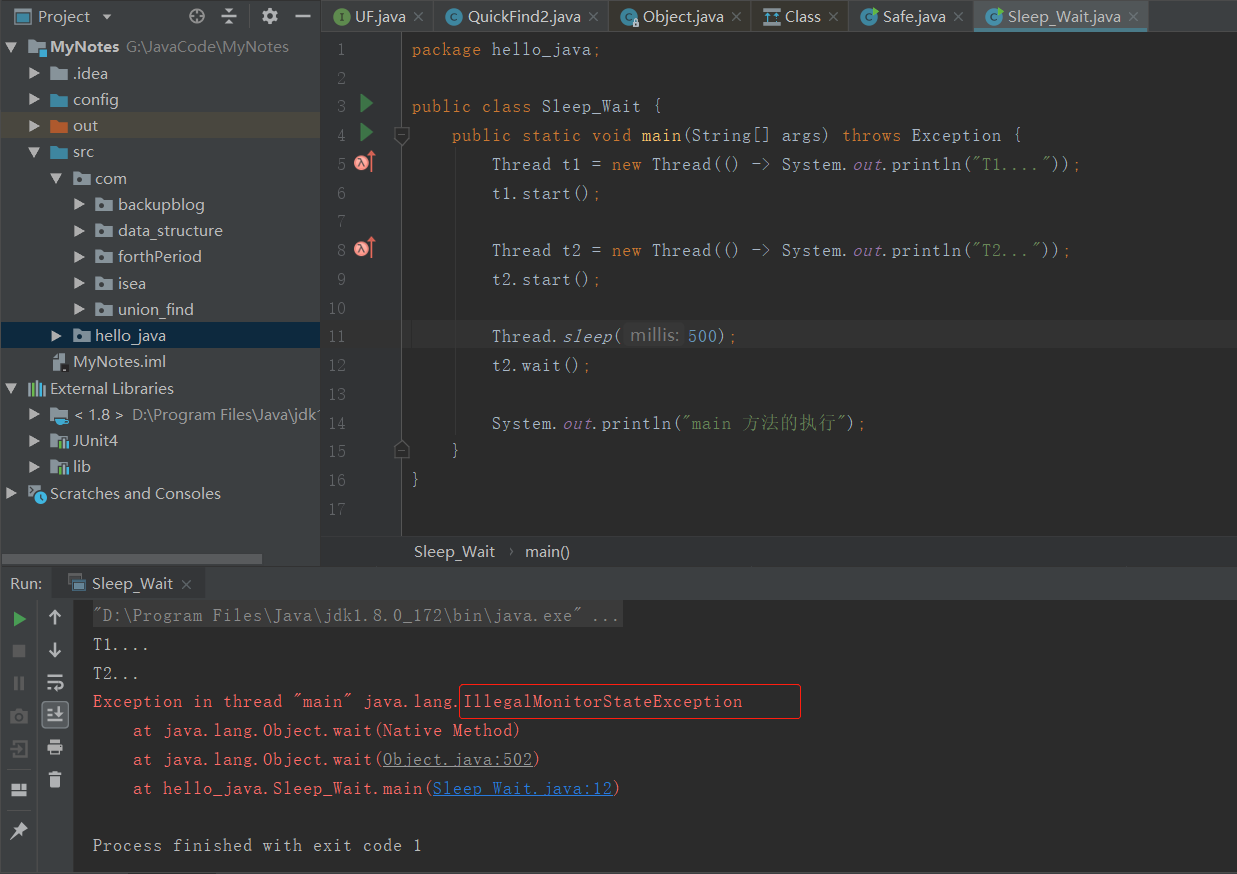
线程安全的解决办法之一就是多例，所谓多例就是堆中对应着有多个对象，各自操作各自的，不会相互影响。

所谓的线程安全问题，其实就是多线程在**并发**执行时，对**共享内存**中的**共享对象**进行**属性修改**时所导致的数据冲突问题。

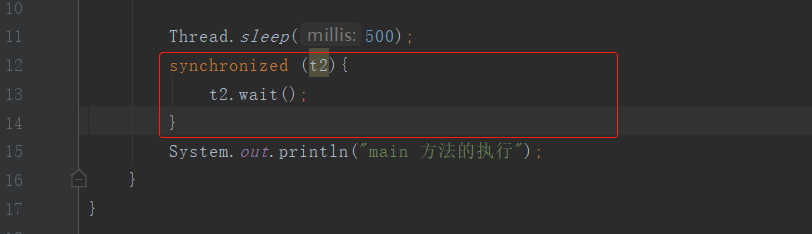
1. Sleep和wait的区别
   1. Sleep是一个静态的方式，使用Thread类来调用；
   2. Wait是object类的方法，和对象相关，表示的是当前线程对象等待。



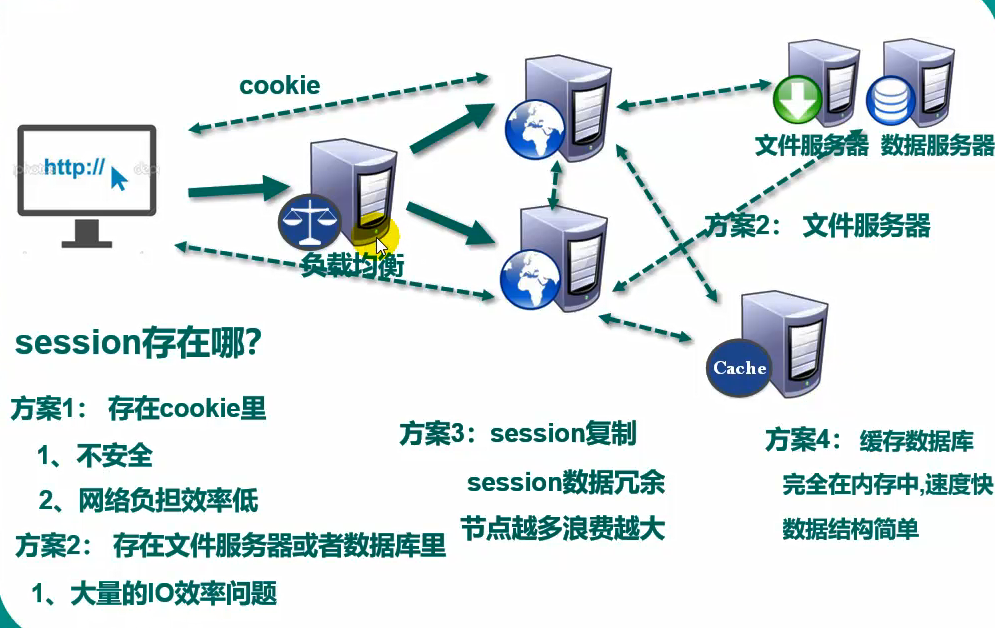
同时，这个程序会报出异常：异常的原因在于，在wait的时候，wait是会释放锁的，因此必须要拿到锁才行。

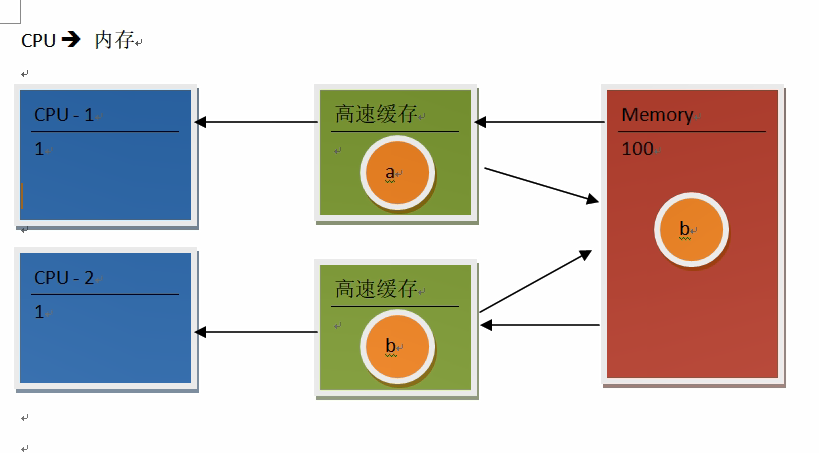


解决办法如下：



1. Redis能够完成什么功能？



1. Cpu的内存模型：
   1. 只有多核的处理器才有真正的多线程，
   2. Cpu为了提升自己的利用率，（cpu的速度快，而内存的速度慢），所以在计算机的内存模型中设置了缓存。
   3. 每一个cpu，都对应着一个独立的高速缓存，但是内存只有一块
   4. 对于内存中的数据，cpu不和内存直接进行交互，而是把内存中的数据复制到缓存中，然后cpu对缓存中的数据进行处理。这个时候会出现一定的问题，比如，假设内存中有一个共享数据，CPU1将缓存1 中的数据修改为a，然后cpu2将其缓存中的共享数据修改为b，那么这个时候，内存中的数据应该被同步成哪个数据呢？所以最后的结果是谁慢，是谁的值如下图：
   5. 
2. **请谈谈你对JVM的理解?java8的虚拟机有什么更新？**
3. **什么是OOM？什么是StackOverflowError?有哪些方法分析？**
4. **JVM的常用参数调优你知道哪些？**
5. **内存快照抓取和MAT分析DUMP文件知道吗？**
6. **谈谈JVM中，对类加载器你的认识？**
7. JVM内存模型以及分区，需要详细到每个区放什么
8. 堆里面的分区：Eden，survival from to，老年代，各自的特点。
9. GC的三种收集方法：标记清除、标记整理、复制算法的原理与特点，分别用在什么地方
10. Minor GC与Full GC分别在什么时候发生
11. 使用Linux命令查询file1中空行所在的行号
12. 有文件chengji.txt内容如下:
    1. 张三 40
    2. 李四 50
    3. 王五 60

使用Linux命令计算第二列的和并输出

1. Shell脚本里如何检查一个文件是否存在？
2. 用shell写一个脚本，对文本中无序的一列数字排序
3. 请用shell脚本写出查找当前文件夹（/home）下所有的文本文件内容中包含有字符”shen”的文件名称
4. 解释一下栈内存溢出和栈溢出的区别：