Hive 高级操作

目录

1、Hive 数据类型

1.1、原子数据类型

数据类型	长度	备注
Tinyint	1 字节的有符号整数	-128~127
SmallInt	1 个字节的有符号整数	-32768~32767
Int	4 个字节的有符号整数	-2147483648 ~ 2147483647
BigInt	8 个字节的有符号整数	9223372036854775808 ~ 9223372036854775807
Boolean	布尔类型，true 或者 false	true、false
Float	单精度浮点数	
Double	双精度浮点数	
String	字符串	
TimeStamp	整数	支持 Unix timestamp，可以达到纳秒精度

1、Hive 支持日期类型(老版本不支持)，在 Hive 里日期一般都是用字符串来表示的，而常用的日期格式转化操作则是通过自定义函数进行操作，当然也可以直接指定为日期类型
2、Hive 是用 Java 开发的，Hive 里的基本数据类型和 java 的基本数据类型也是一一对应的，除了 String 类型。
3、有符号的整数类型：TINYINT、SMALLINT、INT 和 BIGINT 分别等价于 Java 的 Byte、Short、Int 和 Long 原子类型，它们分别为 1 字节、2 字节、4 字节和 8 字节有符号整数。
4、Hive 的浮点数据类型 FLOAT 和 DOUBLE,对应于 Java 的基本类型 Float 和 Double 类型。
5、Hive 的 BOOLEAN 类型相当于 Java 的基本数据类型 Boolean。
6、Hive 的 String 类型相当于数据库的 Varchar 类型，该类型是一个可变的字符串，不过它不能声明其中最多能存储多少个字符，理论上它可以存储 2GB 的字符数。

Numeric Types

- TINYINT (1-byte signed integer, from -128 to 127)
- SMALLINT (2-byte signed integer, from -32,768 to 32,767)
- INT/INTEGER (4-byte signed integer, from -2,147,483,648 to 2,147,483,647)
- BIGINT (8-byte signed integer, from -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807)
- FLOAT (4-byte single precision floating point number)
- DOUBLE (8-byte double precision floating point number)
- DOUBLE PRECISION (alias for DOUBLE, only available starting with Hive 2.2.0)
- DECIMAL
  - Introduced in Hive 0.11.0 with a precision of 38 digits
  - Hive 0.13.0 introduced user-definable precision and scale
- NUMERIC (same as DECIMAL, starting with Hive 3.0.0)

Date/Time Types

- TIMESTAMP (Note: Only available starting with Hive 0.8.0)
- DATE (Note: Only available starting with Hive 0.12.0)
- INTERVAL (Note: Only available starting with Hive 1.2.0)

String Types

- STRING
- VARCHAR (Note: Only available starting with Hive 0.12.0)
- CHAR (Note: Only available starting with Hive 0.13.0)

Misc Types

- BOOLEAN
- BINARY (Note: Only available starting with Hive 0.8.0)

Complex Types

- arrays: ARRAY<data_type> (Note: negative values and non-constant expressions are allowed as of Hive 0.14.)
- maps: MAP<primitive_type, data_type> (Note: negative values and non-constant expressions are allowed as of Hive 0.14.)
- structs: STRUCT<col_name : data_type [COMMENT col_comment], ...>
- union: UNIONTYPE<data_type, data_type, ...> (Note: Only available starting with Hive 0.7.0.)

1.2、复杂数据类型

复杂数据类型包括数组（ARRAY）、映射（MAP）和结构体（STRUCT），具体如下所示：

类型	解释	举例
STRUCT	与C/C++中的结构体类似，可通过“.”访问每个域的值，比如STRUCT {first STRING; last STRING}，可通过name.first访问第一个成员。	struct('John', 'Doe')
MAP	存储key/value对，可通过['key']获取每个key的值，比如'first'→'John' and 'last'→'Doe',可通过name['last']获取last name。	map('first', 'John', 'last', 'Doe')
ARRAY	同种类型的数据集合，从0开始索引，比如['John', 'Doe'],，可通过name[1]获取“Doe”。	array('John', 'Doe')

说明：

ARRAY：ARRAY 类型是由一系列相同数据类型的元素组成，这些元素可以通过下标来访问。比如有一个 ARRAY 类型的变量 fruits，它是由['apple','orange','mango']组成，那么我们可以通过 **fruits[1]**来访问元素 orange，因为 ARRAY 类型的下标是从 0 开始的

MAP：MAP 包含 key->value 键值对，可以通过 key 来访问元素。比如”userlist”是一个 map 类型，其中 username 是 key，password 是 value；那么我们可以通过 **userlist['username']**来得到这个用户对应的 password

STRUCT：STRUCT 可以包含不同数据类型的元素。这些元素可以通过”点语法”的方式来得到所需要的元素，比如 user 是一个 STRUCT 类型，那么可以通过 **user.address** 得到这个用户的地址。

示例：

```
CREATE TABLE student(
      name STRING,
      favors ARRAY<STRING>,
      scores MAP<STRING, FLOAT>,
      address STRUCT<province:STRING, city:STRING, detail:STRING, zip:INT>
)
    ROW FORMAT DELIMITED
    FIELDS TERMINATED BY '\t'
    COLLECTION ITEMS TERMINATED BY ';'
    MAP KEYS TERMINATED BY ':' ;
```

说明：

1、字段 name 是基本类型，favors 是数组类型，可以保存很多爱好，scores 是映射类型，可以保存多个课程的成绩，address 是结构类型，可以存储住址信息

2、ROW FORMAT DELIMITED 是指明后面的关键词是列和元素分隔符的

3、FIELDS TERMINATED BY 是字段分隔符

4、COLLECTION ITEMS TERMINATED BY 是元素分隔符（Array 中的各元素、Struct 中的各元素、Map 中的 key-value 对之间）

5、MAP KEYS TERMINATED BY 是 Map 中 key 与 value 的分隔符

6、LINES TERMINATED BY 是行之间的分隔符

7、STORED AS TEXTFILE 指数据文件上传之后保存的格式

总结：在关系型数据库中，我们至少需要三张表来定义，包括学生基本表、爱好表、成绩表；但在 Hive 中通过一张表就可以搞定了。也就是说，复合数据类型把多表关系通过一张表就可以实现了。

1.3、示例演示：

1.3.1、array

建表语句：

```
create table person(name string,work_locations string)
row format delimited fields terminated by '\t';

create table person1(name string,work_locations array<string>)
row format delimited fields terminated by '\t'
collection items terminated by ',';
```

数据：

```
huangbo  beijing,shanghai,tianjin,hangzhou
xuzheng   changchu,chengdu,wuhan
wangbaoqiang  dalian,shenyang,jilin
```

导入数据：

```
load data local inpath '/home/hadoop/person.txt' into table person;
```

查询语句：

```
Select * from person;
Select name from person;
Select work_locations from person;
Select work_locations[0] from person;
```

1.3.2、map

建表语句：

```
create table score(name string, scores map<string,int>)
row format delimited fields terminated by '\t'
collection items terminated by ','
map keys terminated by ':';
```

数据：

```
huangbo yuwen:80,shuxue:89,yingyu:95
```

xuzheng yuwen:70,shuxue:65,yingyu:81
wangbaoqiang yuwen:75,shuxue:100,yingyu:75

导入数据：
load data local inpath '/home/hadoop/score.txt' into table score;

查询语句：
Select * from score;
Select name from score;
Select scores from score;
Select s.scores['yuwen'] from score s;

1.3.3、struct

建表语句：
create table structtable(id int,course struct<name:string,score:int>)
row format delimited fields terminated by '\t'
collection items terminated by ',';

数据：
1 english,80
2 math,89
3 chinese,95

导入数据：
load data local inpath '/ home/hadoop / structtable.txt' into table structtable;

查询语句：
Select * from structtable;
Select id from structtable;
Select course from structtable;
Select t.course.name from structtable t;
Select t.course.score from structtable t;

1.3.4、uniontype

参考资料：http://yugouai.iteye.com/blog/1849192

2、视图

和关系型数据库一样，Hive 也提供了视图的功能，不过请注意，Hive 的视图和关系型数据

库的数据还是有很大的区别：
1、只有逻辑视图，没有物化视图；
2、视图只能查询，不能 Load/Insert/Update/Delete 数据；
3、视图在创建时候，只是保存了一份元数据，当查询视图的时候，才开始执行视图对应的那些子查询

创建视图

```
create view view_name as select * from carss;
create view carss_view as select * from carss limit 500;
```

查看视图

```
show tables;     // 可以查看表，也可以查看视图
desc view_name    // 查看某个具体视图的信息
desc carss_view
```

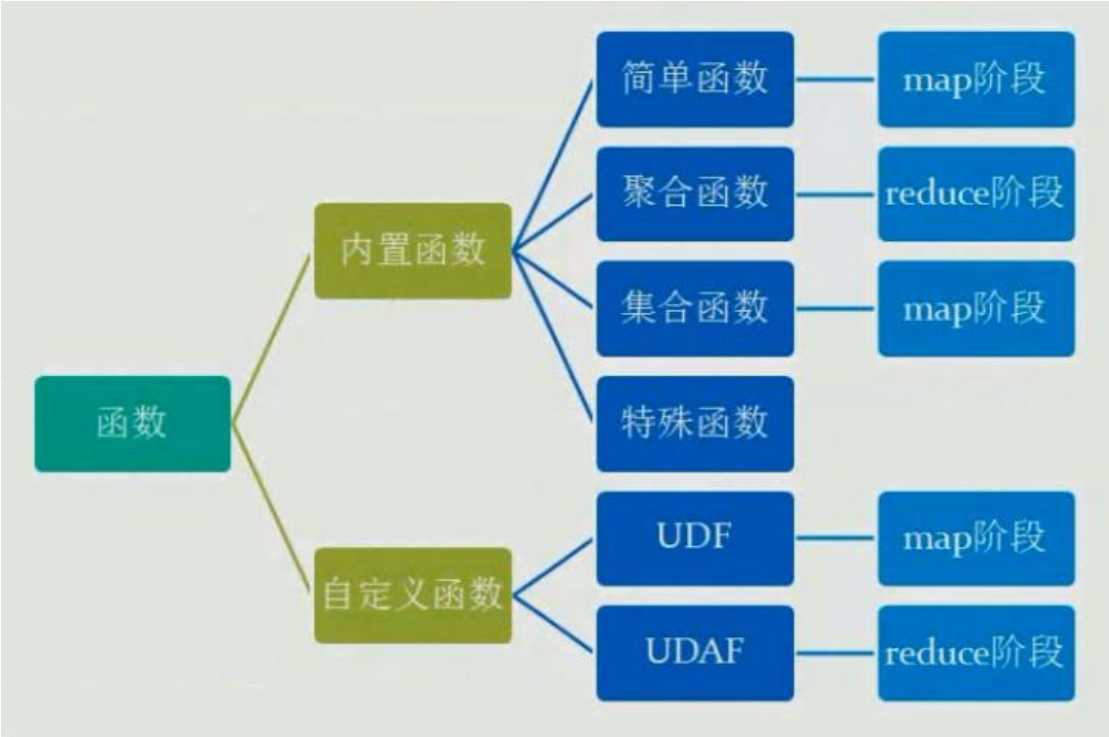
删除视图

```
drop view view_name
drop view if exists carss_view
```

使用视图

```
create view sogou_view as select * from sogou_table where rank > 3 ;
select count(distinct uid) from sogou_view;
```

3、Hive 函数



3.1、Hive 内置函数

1、内容较多，见《Hive 官方文档》
https://cwiki.apache.org/confluence/display/Hive/LanguageManual+UDF

2、测试内置函数的快捷方式：

第一种方式：直接使用，例如：select concat('a','a') ➔ aa

第二种方式：
1、创建一个 dual 表 create table dual(id string);
2、load 一个文件（一行，一个空格）到 dual 表
3、select substr('huangbo',2,3) from dual;

3、查看内置函数：
show functions;
显示函数的详细信息：
desc function abs;
显示函数的扩展信息：
desc function extended concat;

4、内置函数列表

一、关系运算：

1. 等值比较: =
2. 等值比较:<=>
3. 不等值比较: <>和!=
4. 小于比较: <
5. 小于等于比较: <=
6. 大于比较: >
7. 大于等于比较: >=
8. 区间比较
9. 空值判断: IS NULL
10. 非空判断: IS NOT NULL
10. LIKE 比较: LIKE
11. JAVA 的 LIKE 操作: RLIKE
12. REGEXP 操作: REGEXP

二、数学运算：

1. 加法操作: +
2. 减法操作: －
3. 乘法操作: *
4. 除法操作: /
5. 取余操作: %
6. 位与操作: &
7. 位或操作: |
8. 位异或操作: ^
9. 位取反操作: ~

三、逻辑运算：

1. 逻辑与操作: AND 、&&
2. 逻辑或操作: OR 、||
3. 逻辑非操作: NOT、!

四、复合类型构造函数

1. map 结构
2. struct 结构
3. named_struct 结构
4. array 结构
5. create_union

五、复合类型操作符

1. 获取 array 中的元素
2. 获取 map 中的元素
3. 获取 struct 中的元素

六、数值计算函数

1. 取整函数: round
2. 指定精度取整函数: round
3. 向下取整函数: floor
4. 向上取整函数: ceil
5. 向上取整函数: ceiling
6. 取随机数函数: rand
7. 自然指数函数: exp
8. 以 10 为底对数函数: log10
9. 以 2 为底对数函数: log2
10. 对数函数: log
11. 幂运算函数: pow
12. 幂运算函数: power
13. 开平方函数: sqrt
14. 二进制函数: bin
15. 十六进制函数: hex
16. 反转十六进制函数: unhex
17. 进制转换函数: conv
18. 绝对值函数: abs
19. 正取余函数: pmod
20. 正弦函数: sin
21. 反正弦函数: asin
22. 余弦函数: cos
23. 反余弦函数: acos
24. positive 函数: positive
25. negative 函数: negative

七、集合操作函数

1. map 类型大小：size
2. array 类型大小：size
3. 判断元素数组是否包含元素：array_contains
4. 获取 map 中所有 value 集合
5. 获取 map 中所有 key 集合
6. 数组排序

八、类型转换函数

1. 二进制转换：binary
2. 基础类型之间强制转换：cast

九、日期函数

1. UNIX 时间戳转日期函数: from_unixtime
2. 获取当前 UNIX 时间戳函数: unix_timestamp
3. 日期转 UNIX 时间戳函数: unix_timestamp
4. 指定格式日期转 UNIX 时间戳函数: unix_timestamp

5. 日期时间转日期函数: to_date
6. 日期转年函数: year
7. 日期转月函数: month
8. 日期转天函数: day
9. 日期转小时函数: hour
10. 日期转分钟函数: minute
11. 日期转秒函数: second
12. 日期转周函数: weekofyear
13. 日期比较函数: datediff
14. 日期增加函数: date_add
15. 日期减少函数: date_sub

十、条件函数

1. If 函数: if
2. 非空查找函数: COALESCE
3. 条件判断函数：CASE

十一、字符串函数

1. 字符 ascii 码函数：ascii
2. base64 字符串
3. 字符串连接函数：concat
4. 带分隔符字符串连接函数：concat_ws
5. 数组转换成字符串的函数：concat_ws
6. 小数位格式化成字符串函数：format_number
7. 字符串截取函数：substr,substring
8. 字符串截取函数：substr,substring
9. 字符串查找函数：instr
10. 字符串长度函数：length
11. 字符串查找函数：locate
12. 字符串格式化函数：printf
13. 字符串转换成 map 函数：str_to_map
14. base64 解码函数：unbase64(string str)
15. 字符串转大写函数：upper,ucase
16. 字符串转小写函数：lower,lcase
17. 去空格函数：trim
18. 左边去空格函数：ltrim
19. 右边去空格函数：rtrim
20. 正则表达式替换函数：regexp_replace
21. 正则表达式解析函数：regexp_extract
22. URL 解析函数：parse_url
23. json 解析函数：get_json_object
24. 空格字符串函数：space
25. 重复字符串函数：repeat
26. 左补足函数：lpad

27. 右补足函数：rpad
28. 分割字符串函数: split
29. 集合查找函数: find_in_set
30. 分词函数：sentences
31. 分词后统计一起出现频次最高的 TOP-K
32. 分词后统计与指定单词一起出现频次最高的 TOP-K

十二、混合函数

1. 调用 Java 函数：java_method
2. 调用 Java 函数：reflect
3. 字符串的 hash 值：hash

十三、XPath 解析 XML 函数

1. xpath
2. xpath_string
3. xpath_boolean
4. xpath_short, xpath_int, xpath_long
5. xpath_float, xpath_double, xpath_number

十四、汇总统计函数（UDAF）

1. 个数统计函数: count
2. 总和统计函数: sum
3. 平均值统计函数: avg
4. 最小值统计函数: min
5. 最大值统计函数: max
6. 非空集合总体变量函数: var_pop
7. 非空集合样本变量函数: var_samp
8. 总体标准偏离函数: stddev_pop
9. 样本标准偏离函数: stddev_samp
10. 中位数函数: percentile
11. 中位数函数: percentile
12. 近似中位数函数: percentile_approx
13. 近似中位数函数: percentile_approx
14. 直方图: histogram_numeric
15. 集合去重数：collect_set
16. 集合不去重函数：collect_list

十五、表格生成函数 Table-Generating Functions (UDTF)

1. 数组拆分成多行：explode(array)
2. Map 拆分成多行：explode(map)

3.2、Hive 自定义函数 UDF

当 Hive 提供的内置函数无法满足业务处理需要时，此时就可以考虑使用用户自定义函数

UDF（user-defined function）作用于单个数据行，产生一个数据行作为输出。（数学函数，字符串函数）
UDAF（用户定义聚集函数 User- Defined Aggregation Funcation）：接收多个输入数据行，并产生一个输出数据行。（count，max）
UDTF（表格生成函数 User-Defined Table Functions）：接收一行输入，输出多行（explode）

3.2.1、一个简单的 UDF 示例

先开发一个简单的 java 类，继承 org.apache.hadoop.hive.ql.exec.UDF，重载 evaluate 方法

```
Package com.ghgj.hive.udf

import java.util.HashMap;
import org.apache.hadoop.hive.ql.exec.UDF;

public class ToLowerCase extends UDF {
    // 必须是 public，并且 evaluate 方法可以重载
    public String evaluate(String field) {
        String result = field.toLowerCase();
        return result;
    }
}
```

2、打成 jar 包上传到服务器
3、将 jar 包添加到 hive 的 classpath

```
hive>add JAR /home/hadoop/hivejar/udf.jar;
```

查看加入的 jar 的命令：

```
hive> list jar;
```

4、创建临时函数与开发好的 class 关联起来

```
hive>create temporary function tolowercase as 'com.ghgj.hive.udf.ToLowerCase';
```

5、至此，便可以在 hql 在使用自定义的函数

```
select tolowercase(name),age from student;
```

3.2.2、Json 数据解析 UDF 开发

现有原始 json 数据（rating.json）如下，

```
{"movie":"1193","rate":"5","timeStamp":"978300760","uid":"1"}
{"movie":"661","rate":"3","timeStamp":"978302109","uid":"1"}
{"movie":"914","rate":"3","timeStamp":"978301968","uid":"1"}
{"movie":"3408","rate":"4","timeStamp":"978300275","uid":"1"}
```

```
{"movie":"2355","rate":"5","timeStamp":"978824291","uid":"1"}
{"movie":"1197","rate":"3","timeStamp":"978302268","uid":"1"}
{"movie":"1287","rate":"5","timeStamp":"978302039","uid":"1"}
{"movie":"2804","rate":"5","timeStamp":"978300719","uid":"1"}
{"movie":"594","rate":"4","timeStamp":"978302268","uid":"1"}
```

现在需要将数据导入到 hive 仓库中，并且最终要得到这么一个结果：

movie	rate	timeStamp	uid
1193	5	978300760	1

该怎么做、？？？（提示：可用内置 get_json_object 或者自定义函数完成）

3.2.3、Transform 实现

Hive 的 TRANSFORM 关键字提供了在 SQL 中调用自写脚本的功能。适合实现 Hive 中没有的功能又不想写 UDF 的情况

具体以一个实例讲解。
Json 数据：
{"movie":"1193","rate":"5","timeStamp":"978300760","uid":"1"}
需求：把 timestamp 的值转换成日期编号

1、先加载 rating.json 文件到 hive 的一个原始表 rate_json

```
create table rate_json(line string) row format delimited;
load data local inpath '/home/hadoop/rating.json' into table rate_json;
```

2、创建 rate 这张表用来存储解析 json 出来的字段：

```
create table rate(movie int, rate int, unixtime int, userid int) row format delimited fields terminated by '\t';
```

解析 json，得到结果之后存入 rate 表：

```
insert into table rate select
get_json_object(line,'$.movie') as moive,
get_json_object(line,'$.rate') as rate,
get_json_object(line,'$.timeStamp') as unixtime,
get_json_object(line,'$.uid') as userid
from rate_json;
```

3、使用 transform+python 的方式去转换 unixtime 为 weekday
先编辑一个 python 脚本文件

```
########python######代码
## vi weekday_mapper.py

#!/bin/python
import sys
```

```
import datetime
for line in sys.stdin:
  line = line.strip()
  movie,rate,unixtime,userid = line.split('\t')
  weekday = datetime.datetime.fromtimestamp(float(unixtime)).isoweekday()
  print '\t'.join([movie, rate, str(weekday),userid])
```

保存文件
然后，将文件加入 hive 的 classpath：

```
hive>add file /home/hadoop/weekday_mapper.py;
hive> insert into table lastjsontable select transform(movie,rate,unixtime,userid)
using 'python weekday_mapper.py' as(movie,rate,weekday,userid) from rate;
```

// 创建最后的用来存储调用 python 脚本解析出来的数据的表：lastjsontable
create table lastjsontable(movie int, rate int, weekday int, userid int) row format delimited fields terminated by '\t';

最后查询看数据是否正确：
select distinct(weekday) from lastjsontable;

4、HIVE 特殊分隔符处理

补充：hive 读取数据的机制：
1、 首先用 InputFormat<默认是：org.apache.hadoop.mapred.TextInputFormat >的一个具体实现类读入文件数据，返回一条一条的记录（可以是行，或者是你逻辑中的“行”）
2、 然后利用 SerDe<默认：org.apache.hadoop.hive.serde2.lazy.LazySimpleSerDe>的一个具体实现类，对上面返回的一条一条的记录进行字段切割

Hive 对文件中字段的分隔符默认情况下只支持单字节分隔符，如果数据文件中的分隔符是多字符的，如下所示：
01||huangbo
02||xuzheng
03||wangbaoqiang

4.1、使用 RegexSerDe 通过正则表达式来抽取字段

```
create table t_bi_reg(id string,name string)
row format serde 'org.apache.hadoop.hive.serde2.RegexSerDe'
with serdeproperties('input.regex'='(.*)\\|\\|(.*)','output.format.string'='%1$s %2$s')
stored as textfile;

hive>load data local inpath '/home/hadoop /hivedata/bi.dat' into table t_bi_reg;
```

```
hive>select * from t_bi_reg;
```

4.2、通过自定义 InputFormat 解决特殊分隔符问题

其原理是在 inputformat 读取行的时候将数据中的“多字节分隔符”替换为 hive 默认的分隔符（ctrl+A 亦即 \001）或用于替代的单字符分隔符，以便 hive 在 serde 操作时按照默认的单字节分隔符进行字段抽取

com.ghgj.hive.delimit2.BiDelimiterInputFormat

```
package com.ghgj.hive.delimit2;
import java.io.IOException;
import org.apache.hadoop.io.LongWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapred.FileSplit;
import org.apache.hadoop.mapred.InputSplit;
import org.apache.hadoop.mapred.JobConf;
import org.apache.hadoop.mapred.RecordReader;
import org.apache.hadoop.mapred.Reporter;
import org.apache.hadoop.mapred.TextInputFormat;

public class BiDelimiterInputFormat extends TextInputFormat {

    @Override
    public RecordReader<LongWritable, Text> getRecordReader(InputSplit genericSplit,
JobConf job, Reporter reporter)throws IOException {

        reporter.setStatus(genericSplit.toString());
        BiRecordReader reader = new BiRecordReader(job,(FileSplit)genericSplit);
    //  MyRecordReader reader = new MyRecordReader(job,(FileSplit)genericSplit);
        return reader;
    }
}
```

com.ghgj.hive.delimit2.BiRecordReader

```
package com.ghgj.hive.delimit2;
import java.io.IOException;
import java.io.InputStream;
import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;
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.fs.Seekable;
import org.apache.hadoop.io.LongWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.io.compress.CodecPool;
import org.apache.hadoop.io.compress.CompressionCodec;
import org.apache.hadoop.io.compress.CompressionCodecFactory;
import org.apache.hadoop.io.compress.Decompressor;
import org.apache.hadoop.io.compress.SplitCompressionInputStream;
import org.apache.hadoop.io.compress.SplittableCompressionCodec;
import org.apache.hadoop.mapred.FileSplit;
import org.apache.hadoop.mapred.LineRecordReader;
import org.apache.hadoop.mapred.RecordReader;

public class BiRecordReader implements RecordReader<LongWritable, Text> {
	private static final Log LOG = LogFactory.getLog(LineRecordReader.class
			.getName());

	private CompressionCodecFactory compressionCodecs = null;
	private long start;
	private long pos;
	private long end;
	private LineReader in;
	int maxLineLength;
	private Seekable filePosition;
	private CompressionCodec codec;
	private Decompressor decompressor;

	/**
	 * A class that provides a line reader from an input stream.
	 * @deprecated Use {@link org.apache.hadoop.util.LineReader} instead.
	 */
	@Deprecated
	public static class LineReader extends org.apache.hadoop.util.LineReader {
		LineReader(InputStream in) {
			super(in);
		}

		LineReader(InputStream in, int bufferSize) {
			super(in, bufferSize);
		}

		public LineReader(InputStream in, Configuration conf)
				throws IOException {
```

```
            super(in, conf);
        }
    }

    public BiRecordReader(Configuration job, FileSplit split) throws IOException {
        this.maxLineLength = job.getInt("mapred.linerecordreader.maxlength",
                Integer.MAX_VALUE);
        start = split.getStart();
        end = start + split.getLength();
        final Path file = split.getPath();
        compressionCodecs = new CompressionCodecFactory(job);
        codec = compressionCodecs.getCodec(file);

        // open the file and seek to the start of the split
        FileSystem fs = file.getFileSystem(job);
        FSDataInputStream fileIn = fs.open(split.getPath());

        if (isCompressedInput()) {
            decompressor = CodecPool.getDecompressor(codec);
            if (codec instanceof SplittableCompressionCodec) {
            final SplitCompressionInputStream cIn = ((SplittableCompressionCodec) codec)
                        .createInputStream(fileIn, decompressor, start, end,
                                SplittableCompressionCodec.READ_MODE.BYBLOCK);
                in = new LineReader(cIn, job);
                start = cIn.getAdjustedStart();
                end = cIn.getAdjustedEnd();
                filePosition = cIn; // take pos from compressed stream
            } else {
                in = new LineReader(codec.createInputStream(fileIn,
                        decompressor), job);
                filePosition = fileIn;
            }
        } else {
            fileIn.seek(start);
            in = new LineReader(fileIn, job);
            filePosition = fileIn;
        }
        // If this is not the first split, we always throw away first record
        // because we always (except the last split) read one extra line in
        // next() method.
        if (start != 0) {
            start += in.readLine(new Text(), 0, maxBytesToConsume(start));
        }
        this.pos = start;
```

```
    }

    private boolean isCompressedInput() {
        return (codec != null);
    }

    private int maxBytesToConsume(long pos) {
        return isCompressedInput() ? Integer.MAX_VALUE : (int) Math.min(
                Integer.MAX_VALUE, end - pos);
    }

    private long getFilePosition() throws IOException {
        long retVal;
        if (isCompressedInput() && null != filePosition) {
            retVal = filePosition.getPos();
        } else {
            retVal = pos;
        }
        return retVal;
    }

    public BiRecordReader(InputStream in, long offset, long endOffset,
            int maxLineLength) {
        this.maxLineLength = maxLineLength;
        this.in = new LineReader(in);
        this.start = offset;
        this.pos = offset;
        this.end = endOffset;
        this.filePosition = null;
    }

    public BiRecordReader(InputStream in, long offset, long endOffset,
            Configuration job) throws IOException {
        this.maxLineLength = job.getInt("mapred.linerecordreader.maxlength",
                Integer.MAX_VALUE);
        this.in = new LineReader(in, job);
        this.start = offset;
        this.pos = offset;
        this.end = endOffset;
        this.filePosition = null;
    }

    public LongWritable createKey() {
        return new LongWritable();
```

```
    }

    public Text createValue() {
        return new Text();
    }

    /** Read a line. */
    public synchronized boolean next(LongWritable key, Text value)
            throws IOException {

        // We always read one extra line, which lies outside the upper
        // split limit i.e. (end - 1)
        while (getFilePosition() <= end) {
            key.set(pos);

            int newSize = in.readLine(value,
maxLineLength,Math.max(maxBytesToConsume(pos), maxLineLength));
            String str = value.toString().replaceAll("\\|\\|", "\\|");
            value.set(str);
            pos += newSize;

            if (newSize == 0) {
                return false;
            }
            if (newSize < maxLineLength) {
                return true;
            }

            // line too long. try again
            LOG.info("Skipped line of size " + newSize + " at pos "
                    + (pos - newSize));
        }

        return false;
    }

    /**
     * Get the progress within the split
     */
    public float getProgress() throws IOException {
        if (start == end) {
            return 0.0f;
        } else {
            return Math.min(1.0f, (getFilePosition() - start)
```

```
                        / (float) (end - start));
            }
        }

        public synchronized long getPos() throws IOException {
            return pos;
        }

        public synchronized void close() throws IOException {
            try {
                if (in != null) {
                    in.close();
                }
            } finally {
                if (decompressor != null) {
                    CodecPool.returnDecompressor(decompressor);
                }
            }
        }
}
```

注意：上述代码中的 api 全部使用 hadoop 的老 api 接口 org.apache.hadoop.mapred….
然后将工程打包，并拷贝至 hive 安装目录的 lib 文件夹中，并重启 hive，使用以下语句建表即可：

```
hive> create table new_bi(id string,name string)
    > row format delimited
    > fields terminated by '|'
    > stored as inputformat 'com.ghgj.hive.delimit2.BiDelimiterInputFormat' outputformat
'org.apache.hadoop.hive.ql.io.HiveIgnoreKeyTextOutputFormat';

hive> load data local inpath '/home/hadoop/bi.dat' into table new_bi

hive>   select * from new_bi;
OK
01          huangbo
02          xuzheng
03          wangbaoqiang
```

注：还需要在 hive 中使用 add jar，才能在执行 hql 查询该表时把自定义 jar 包传递给 maptask
hive>add jar /home/hadoop/apps/hive/lib/myinput.jar