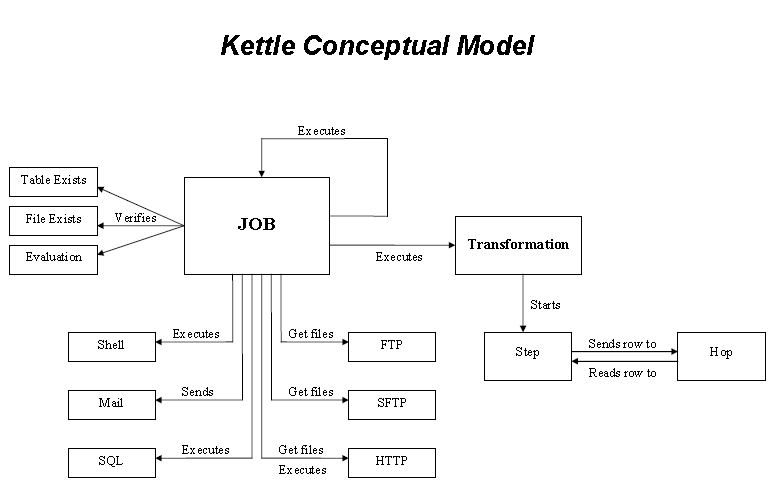
Kettle4.4源代码分析

2016年06月

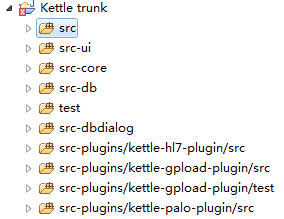
# 概念模型



要了解Kettle的执行分为两个层次：Job和Transformation。Job和Transformation文件存在两种模式：第一种是存成本地文件，第二种是存储到数据库中。

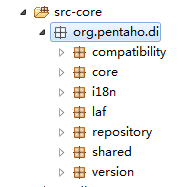
# 源代码结构

## 总体结构



Kettle源代码由7个source folder组成，分别是：src-core、src、src-ui、src-db、test、src-dbdialog、 src-plugins。

### src-core源代码文件夹



#### org.pentaho.di.compatibility包

系统用到的数值类型及对应接口。

#### org.pentaho.di.core.exception包

异常类

#### org.pentaho.di.core.xml包

XML相关接口及封装类。这些类涉及

#### org.pentaho.di.core.config 包

只有两个类，OgnlExpression 用来封装ognl表达式的。PropertySetter 辅助类。

#### org.pentaho.di.core.encryption 包（kettle 的密码的加密与解密）

#### org.pentaho.di.core.exception 包 kettle中的各种关于kettle异常管理，如文件、数据库、自定义的kettleException等。

#### org.pentaho.di.core.logging包

Log设置。

#### org.pentaho.di.core.plugins包

组件加载。

#### org.pentaho.di.core.row包

行的数据、元信息、操作。

#### org.pentaho.di.core.gui包

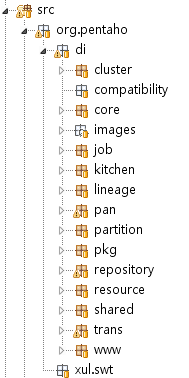
界面接口类和一些基础图形类。

#### 其他

org.pentaho.di.Const类维护系统常量及配置文件地址。

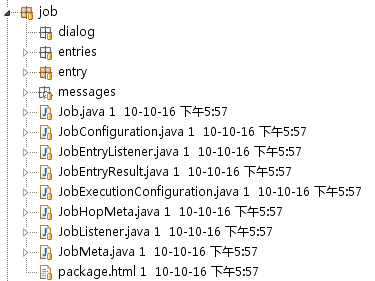
Org.

### src源代码文件夹

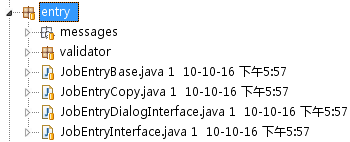


包含调度逻辑和具体的执行代码。最重要的两个包为org.pentaho.di.job和org.pentaho.di.trans。

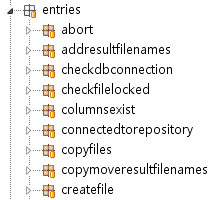
#### org.pentaho.di.job包



Job的每个执行单元称为entry。

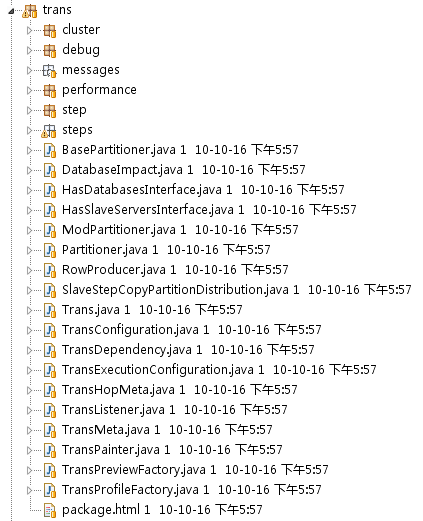


org.pentaho.di.job.entry包中存放了每个entry必须继承的基类以及必须实现的接口。

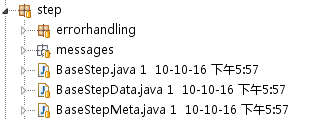


org.pentaho.di.job.entries包中存放了不同entry的具体实现。

#### org.pentaho.di.trans包



Transfromation中的每个执行步骤称为step。



org.pentaho.di.job.entry包中存放了每个entry必须继承的基类以及必须实现的接口。



org.pentaho.di.job.entries包中存放了不同entry的具体实现。

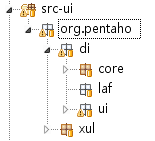
#### org.pentaho.di.kitchen包

Job的命令行执行器类。

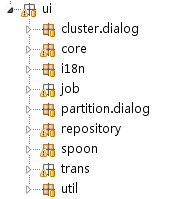
#### org.pentaho.di.pan包

Transformation的命令行执行器类。

### src-ui源代码文件夹

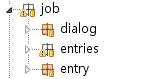


最主要的就是其下的org.pentaho.di.ui包。



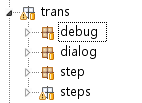
结构如上图所示，其中最重要的3个子包为：org.pentaho.di.ui.job、org.pentaho.di.ui.trans和org.pentaho.di.ui.spoon。

#### org.pentaho.di.ui.job包



每个entry的参数设置面板类、进程对话类及需要继承的基类。

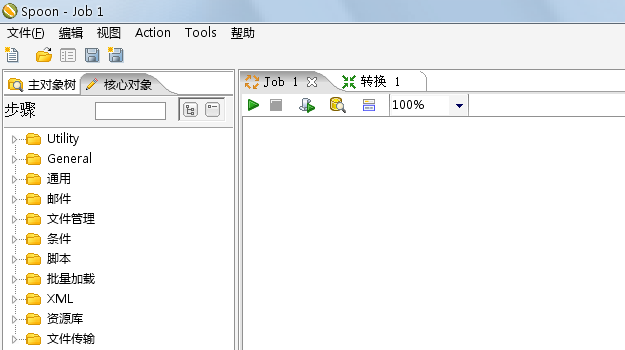
#### org.pentaho.di.ui.trans包



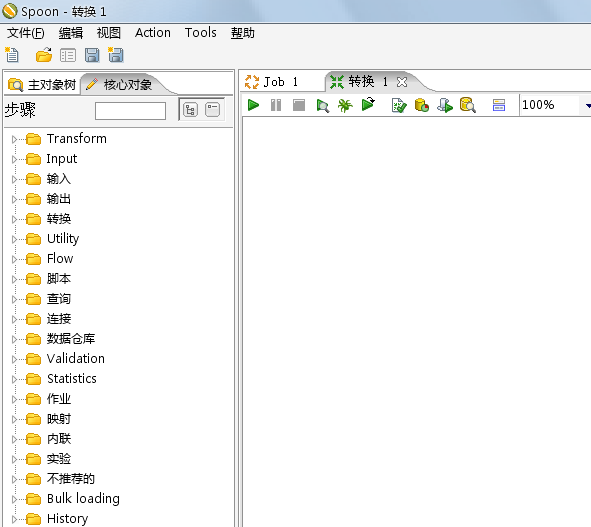
每个step的参数设置面板类、进程对话类及需要继承的基类。

#### org.pentaho.di.ui.spoon包

整个软件的入口对应org.pentaho.di.ui.spoon包中的Spoon类。



选中Job标签后，红框内的编辑区对象对应org.pentaho.di.ui.spoon.job包中的JobGraph类。

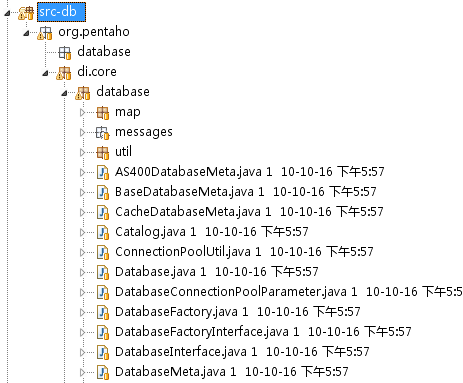


选中转换标签后，红框内的编辑区对象对应org.pentaho.di.ui.spoon.trans包中的TransGraph类。

#### 其他

org.pentaho.di.core.KettleEnvironment类负责初始化Kettle运行环境，主要包括调用org.pentaho.di.core.plugins.PluginRegistry类的init()方法加载组件。

### src-db源代码文件夹



org.pentaho.di.core.database下包含各数据库对应的类以及对应的元数据类，还包括不同类型数据库必须继承的基类和必须实现的接口。

### test源代码文件夹

测试代码。（未阅读）

# 数据流图

## 顶层数据流图



## 第一层数据流图



## 第二层数据流图

### 作业执行



### 转换执行



# Kettle启动

## 顶层

对应org.pentaho.di.ui.spoon包中Spoon类：

public static void main(String[] a) throws KettleException {

try {

//3.2.初始化Kettle运行环境

KettleEnvironment.init();

...

//3.3.初始化Spoon界面, 读取配置文件（.xul）

staticSpoon = new Spoon(display);

staticSpoon.init(null);

//开启Spoon，enable各个组件

staticSpoon.start(splash, commandLineOptions);

}catch(...){

}

}

## 初始化Kettle运行环境

对应org.pentaho.di.core包中KettleEnvironment类：

public static void init(boolean simpleJndi) throws KettleException {

//若未曾初始化

if (initialized==null) {

//若不存在则创建一个home文件夹

createKettleHome();

//加载home中kettle.properties文件的内容

EnvUtil.environmentInit();

//Log初始设置：容量大小、超时时长

CentralLogStore.init();

//加载Kettle需要用到的变量值，从Kettle-variables.xml文件

KettleVariablesList.init();

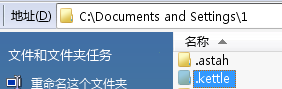
initialized = true;

}

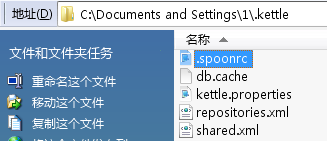
}

### Home文件夹

根据用户系统设置建立“.kettle”文件夹。



该文件夹内存放用户对于系统的喜好配置以及历史信息。这些信息会在启动时被读取。



## 初始化Spoon界面

读取ui\\*.xul文件进行部署。

对应Spoon类的init()方法：

public void init(TransMeta ti) {

//对界面布局进行设置

shell.setLayout(layout);

//加入ktr、kjb文件读写监听器

addFileListener(new TransFileListener());

addFileListener(new JobFileListener());

//加载初始化一些变量

try {

//SwtXulLoader类没有提供src

xulLoader = new SwtXulLoader();

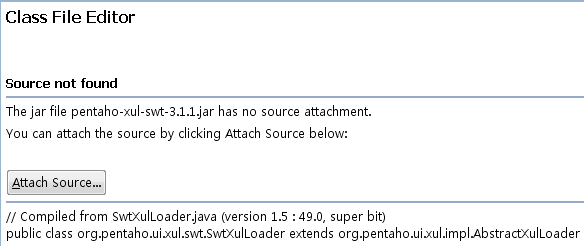
} catch (…) {

}

//加载部分固定组件及对应监听器

}

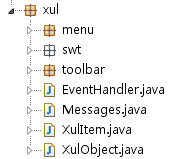
源代码无法查看的截图：



这里分析的源代码是最新的4.0版本。3.2版本的这部分代码是可以查看的，但两者结构不同：



上图为4.0版本



上图为3.2版本。

# Job、Transformation界面调用

通过用户界面操作（点击运行按钮）触发监听器，调用Spoon的runFile()方法。

## Spoon类runFile( )

public void runFile() {

executeFile(true, false, false, false, false, null, false);

}

## Spoon类executeFile( )

在这一层决定执行Job还是Transformation。

public void executeFile(boolean local, boolean remote, boolean cluster, boolean preview, boolean debug,Date replayDate, boolean safe) {

//获取当前活跃的Transformation元信息

TransMeta transMeta = getActiveTransformation();

if (transMeta != null)

executeTransformation(transMeta, local, remote, cluster, preview, debug, replayDate, safe);

//获取当前活跃的Job元信息

JobMeta jobMeta = getActiveJob();

if (jobMeta != null)

executeJob(jobMeta, local, remote, replayDate, safe);

}

## Spoon类getActiveTransformation( )、getActiveJob( )

public TransMeta getActiveTransformation() {

EngineMetaInterface meta = getActiveMeta();

if (meta instanceof TransMeta) {

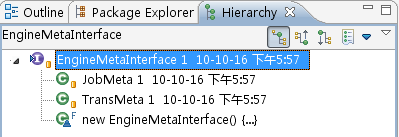
return (TransMeta) meta;

}

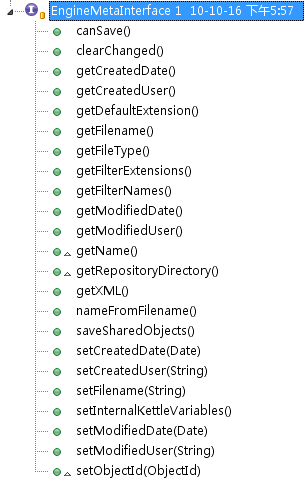
return null;

}

getActiveJob()类的实现同getActiveTransformation()。



如上图，JobMeta和TransMeta都实现了EngineMetaInterface。



上图可见，EngineMetaInterface包含Job、Transformation的整体操作。

## Spoon类getActiveMeta( )

public EngineMetaInterface getActiveMeta() {

if (tabfolder == null)

return null;

TabItem tabItem = tabfolder.getSelected();

if (tabItem == null)

return null;

//通过当前活跃的Tab标签确定返回类型。

TabMapEntry mapEntry = delegates.tabs.getTab(tabfolder.getSelected());

EngineMetaInterface meta = null;

if (mapEntry != null) {

if (mapEntry.getObject() instanceof TransGraph)

meta = (mapEntry.getObject()).getMeta();

if (mapEntry.getObject() instanceof JobGraph)

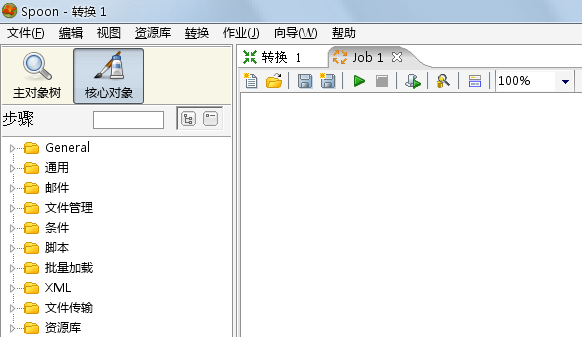
meta = (mapEntry.getObject()).getMeta();

}

return meta;

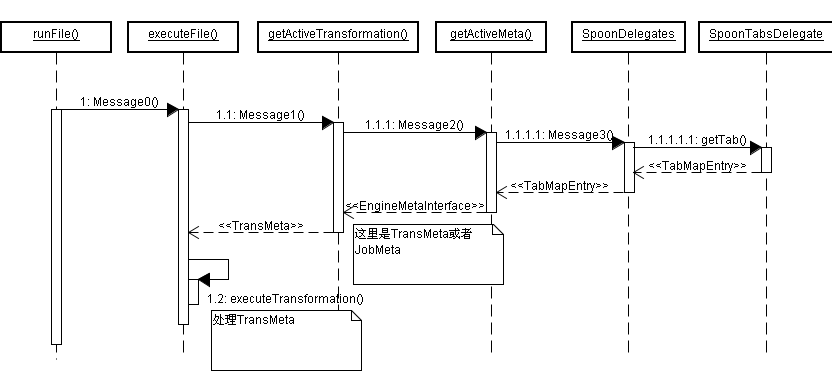
}

下图红色方框内为活跃的Tab。



## 选择执行时序图

下图总结了4.1-4.4节内容，即系统如何判定执行Job还是Transformation。



## Job调用后续步骤

最终转交给JobGraph负责执行。

实现方法同4.7。

## Transformation调用后续步骤

最终转交给TransGraph负责执行。

### Spoon类executeTransformation( )

public void executeTransformation(…) {

new Thread() {

//利用trans执行代理执行trans

delegates.trans.executeTransformation(transMeta, local, remote, cluster, preview, debug,

replayDate, safe);

}.start();

}

### SpoonTransformationDelegate类executeTransformation( )

public void executeTransformation(…){

//获取当前活跃的trans

TransGraph activeTransGraph = spoon.getActiveTransGraph();

//将配置设置入executionConfiguration后调用TransGraph实例执行

activeTransGraph.start(executionConfiguration);

}

# Job执行

## 相关类和接口

### JobGraph

维护整个Job编辑区的信息和相应操作。

主要成员变量：

**private** JobMeta jobMeta; 由Job编辑面板动态维护

**private** Repository rep;

**private** Job parentJob;

**private** JobTracker jobTracker;用于跟踪日志记录

**private** Date startDate, endDate, currentDate, logDate, depDate;

**private** **boolean** active, stopped;状态位

**private** List<RowMetaAndData> sourceRows; 返回结果的数据内容

**private** Result result;每次执行完一个jobentry返回结果

### JobMeta

维护整个Job的元数据。

主要成员变量：

**protected** String name;

**protected** String filename;

**public** List<JobEntryInterface> jobentries;保存jobentry列表

**public** List<JobHopMeta> jobhops;保存jobentries之间的链接关系。

**public** List<DatabaseMeta> databases;

### JobEntryInterface

每个具体org.pentaho.di.job.entries包下的 entry类需要实现的接口。

包含execute（）方法。

### Result

每一个jobEntryInterface的实现类在完成相应功能时，返回结果的类型。

主要成员变量：

**private** **boolean** result;执行是否出现异常

**private** **int** exitStatus; 执行结果状态

**private** List<RowMetaAndData> rows;一个jobEntry完成处理后的数据（若存在）

**private** Map<String, ResultFile> resultFiles;

### JobEntryCopy

维护每一个不同entry或者相同entry的不同副本的信息

主要成员：

**private** JobEntryInterface entry;具体entry,执行入口

**private** **int** nr;副本数，一个编辑区里可以出现多个相同组件

**private** **boolean** selected;

**private** Point location;图标位置

**private** **boolean** draw;

private long id;

## Job执行过程时序图

下图描述了上面各类之间的总体调用关系。



## 代码执行说明

### JobGraph类start( )

该类主要功能是实例化job、开启job的线程。主要代码如下：

job = **new** Job(log, jobMeta.getName(), jobMeta.getFilename(), **null**);

job.start();

### JobGraph类start( )

设置状态位，调用execute方法1，部分代码代码如下：

**public** **void** run(){

stopped**=false;**

finished=**false**;

initialized = **true**;

result = execute(**false**);

}

### Job类execute( )方法1

主要工作是从JobMeta的JobHopMeta找到job入口jobentry信息，根据开始条件调用真正执行jobentry的execute方法2，代码如下所示：

startpoint=jobMeta.findJobEntry(JobMeta.*STRING\_SPECIAL\_START*, 0, false);

// 找到Job开始组件

JobEntrySpecial jes = (JobEntrySpecial) startpoint.getEntry();

// JobEntrySpecial是启动job的job项目

Result res = null;

while ( (jes.isRepeat() || isFirst) && !isStopped()){

//符合开始条件时，调用execute方法2

isFirst = false;

res = execute(0, null, startpoint, null,

Messages.*getString*("Job.Reason.Started"));

}

### Job类execute( )方法2

主要功能是根据参数startpoint，提取对应的jobentry，执行对应的jobentry操作，再根据JobMeta的hop信息依次得到下一个jobentry，嵌套调用execute方法2调用，代码如下：

JobEntryInterface jei = startpoint.getEntry();

JobEntryInterface cloneJei = (JobEntryInterface)jei.clone()；

//以下是执行JobEntryInterface的实现类的execute()方法。

**final** Result result = cloneJei.execute(prevResult, nr, rep, **this**);

//根据jobMeta的Hop信息找到下一跳的个数

**int** nrNext = jobMeta.findNrNextJobEntries(startpoint);

**for** (**int** i=0;i<nrNext && !isStopped();i++){

//对于每一个下一跳分别找到每一个入口。

JobEntryCopy nextEntry = jobMeta.findNextJobEntry(startpoint, i);

// 得到startpoint, nextEntry 的信息，找到hop元数据信息，判断是否应该执行

JobHopMeta hi = jobMeta.findJobHop(startpoint, nextEntry);

//如果应该执行，则把前一个JobEntry执行结果result和应该执行的jobentry作为参数嵌套调用execute方法2。

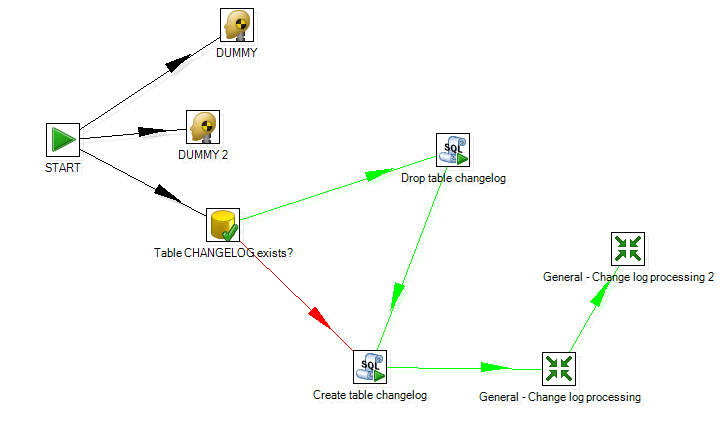
res = execute(nr+1, result, nextEntry, startpoint, nextComment);

}

## 执行示例说明

调试一个Job，其拓扑如下：

黑色Hop代表无条件执行下一个JobEntry，红色代表结果为false执行下一jobEntry，绿色代表结果为true执行下一跳。



7

6

5

4

3

2

1

General-Change log Processing和General-Change log Processin 2是两个trans。

job的执行顺序为jobEntry的深度遍历，如标号所示。

## JobEntry执行

### JobEntry类

具体每个组件的执行体对应org.pentaho.di.job.entries包内每个entry的具体实现。

execute()方法2中调用jobEntry的execute()完成jobEntry的具体功能。

### 不同jobEntry的实现

**final** Result result = cloneJei.execute(prevResult, nr, rep, **this**);

不同的Job项目（JobEntry）实现差别很大。

#### JobEntrySpecial

功能是开启一个job，只是简单地对传递来的preResult设置它的的result属性值为true，（Job项目据此判断前一结果执行完毕）。返回该对象即可。

#### JobEntryTableExit

功能是判断一个table是否存在数据库中。JobEntryTableExit Job项目有属性tablename和DatabaseMeta（对数据库的元数据信息描述）根据DatabaseMeta得到一个Dabase对象db，建立连接db.connect(); 调用db.checkTableExists(tablename)根据此返回值设置preResult的result属性为否为true。返回preResult对象。

#### JobEntryTrans

JobEntryJob和JobEntryTrans是嵌套job或trans的Job项目（JobEntry）。它们是比较复杂的job项目。

作用是执行一个trans。首先实例化一个TransMeta,之后实例化Trans。调用trans.start()，当执行完毕后调用函数trans.getResult()，并把结果加到preResult中，返回该对象即可。

#### 补充说明

Result中也可以有处理数据，这些处理数据可以作为下一个Job项目（JobEntry）的输入。但是容量受内存容量限制。

# Transformation执行

## 相关类和接口

### JobEntryTrans

实现了JobEntryInterface的execute（）方法，被job执行。由JobEntryTrans实例化Trans，并执行。

### TransGraph

当点击trans面板的run时，由TransGraph实例化Trans，并执行。

Trans主要成员有：

**private** TransMeta transMeta;

**private** Repository repository;

**private** Job parentJob;

**private** Trans parentTrans;

**private** List<RowSet> rowsets;

**private** List<StepMetaDataCombi> steps

其中最重要的是rowsets、steps。rowsets保存了所有hop对应的行元数据和数据信息。List<StepMetaDataCombi> steps封装了一个step的主要内容。

### TransMeta

描述了整个Trans的元数据信息。 主要的属性成员有：

**private** List<StepMeta> steps;

**private** List<TransHopMeta> hops;

**private** String name;

**private** Result previousResult;上一个jobentry的执行结果。

**private** List<RowMetaAndData> resultRows;这次trans执行后的数据结果。

**private** List<ResultFile> resultFiles;

resultRows成员将作为result比部分返回多行的元数据和数据（如果有的话）需要返回数据结果时。把resultRows加入Result结果的rows列表，并返回。

### StepMetaDataCombi

提取了一个step所需的主要信息。

**public** **class** StepMetaDataCombi

{

**public** StepMeta stepMeta;

**public** String stepname;

**public** **int** copy;

**public** StepInterface step;

**public** StepMetaInterface meta;

**public** StepDataInterface data;

}

### TransHopMeta

描述hop信息。

### StepMeta

描述step的公有基本信息（stepid，stepname），对于每一个具体的step，由成员变量StepMetaInterface step来描述。

### StepInterface

主要成员函数：

processRow()对一行的数据处理。

putRow()把处理后的数据放入下一个step的inputrowsets中。

### StepBase

实现了StepInterface是各step具体实现类的基类。完成了公用的处理函数，如putRow()，但是对于更具体的processRow()在StepBase的子类中。StepBase的主要成员有

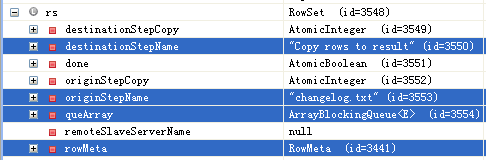
public ArrayList<RowSet> inputRowSets，outputRowSets；

StepBase的子类每次从inputRowSets中取出一行数据，向outputRowSets中写入一行数据。

### StepDataInterface

与step相关的数据信息。比如行的元数据信息。StepMetaInterface的实现类是与具体step相关的元数据信息，与StepMeta配合使用，共同描述具体step的元数据信息。

### RowSet



RowSet类中包含源step，目标step和由源向目标发送的一个rowMeta和一组data。其中data数据是以行为单位的队列（queArray）。一个RowSet作为此源step的outputrowsets的一部分。同时作为目标step的inputRowsets一部分。源Step每次向队列中写一行数据，目标step每次从队列中读取一行数据。

### RowMetaAndData

**public** **class** RowMetaAndData **implements** Cloneable{

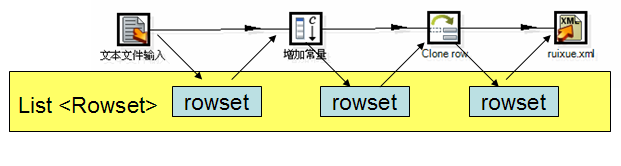
**private** RowMetaInterface rowMeta;//行的元数据，描述了每行的数据名字，数据类型。

**private** Object[] data;//数据

}

## 执行过程概述

Trans的执行机制是搭建一个结构，使得每一个step能够从自己的inputRowsets读，处理一行，将结果输出到自己的outputRowsets中。

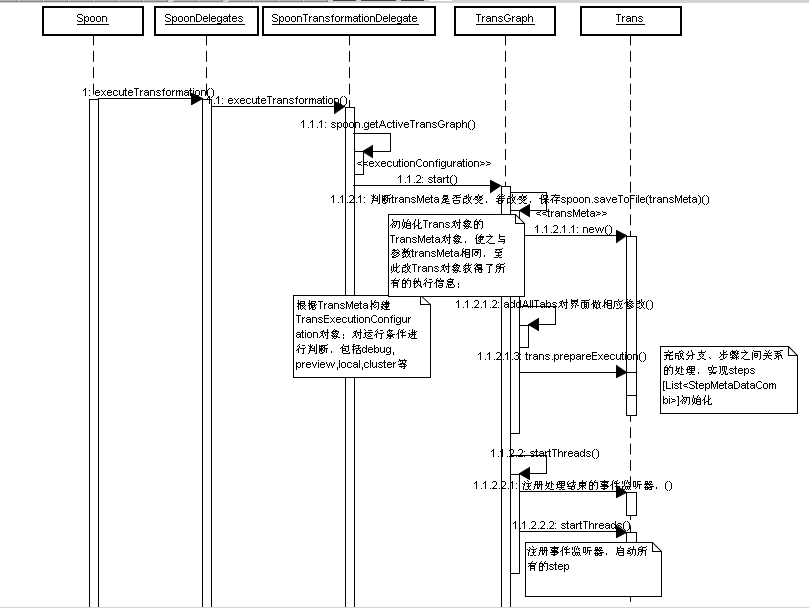


注意：一个rowset对象既属于前一个step成员outputRowsets的一部分，也属于后一个对象的inputRowsets的一部分。所有的rowset信息都在Trans对象中以List形式维护。

## Trans执行过程时序图

由于trans可以有TransGraph实例化，也可以由JobEntryTrans实例化。但基本过程是一样的，先实例化TransMeta，再实例化Trans，最终调用trans的start方法。

由TransGraph实例化如下图所示：



由JobEntryTrans实例化，如下图所示：



## Trans代码解释

### JobEntryTrans类execute( )

首先获取元数据，然后以此作为参数实例化trans

TransMeta transMeta = getTransMeta(rep);

Trans trans = **new** Trans(transMeta);

trans.execute(args);

### Trans类execute( )

具体执行前需要进行准备工作

**public** **void** execute(String[] arguments) **throws** KettleException{

prepareExecution(arguments);

startThreads();

}

### Trans的prepareExecution（）

搭建以下结构结构。

(1)、对每一个step根据hop信息进行找到下一个step或多个step。

(2)、对于每一个this step和nex tstep生成一个RowSet对象，作为缓存供this step写，同时供next step读取数据。

(3)、把此RowSet对象加入到Trans的List<RowSet>成员中保存。

List<StepMeta> hopsteps=transMeta.getTransHopSteps(**false**);

得到step列表

对每一个step进行如下设置

**for** (**int** i=0;i<hopsteps.size();i++)

{

StepMeta thisStep=hopsteps.get(i);

**//**找到下一个step的列表

List<StepMeta> nextSteps = transMeta.findNextSteps(thisStep);

**int** nrTargets = nextSteps.size();

**for** (**int** n=0;n<nrTargets;n++)

{

StepMeta nextStep = nextSteps.get(n);

**//对于每一个hop信息生成RowSet,并设置RowSet**

RowSet rowSet=**new** RowSet(transMeta.getSizeRowset());

rowSet.setThreadNameFromToCopy(thisStep.getName(),0, nextStep.getName(), 0);

rowsets.add(rowSet);

}

}

(4)、根据TransMeta的step信息生成相应的StepMetaDataCombi（即steps）信息，加到steps列表中。

StepMetaDataCombi combi = new StepMetaDataCombi();

combi.stepname = stepMeta.getName();

combi.copy = c;

combi.stepMeta = stepMeta;

combi.meta = stepMeta.getStepMetaInterface();

StepDataInterface data = combi.meta.getStepData();

combi.data = data;

StepInterface step=combi.meta.getStep(stepMeta, data, c, transMeta, **this**);

在step初始化时，会把Trans中的List<RowSet>的相应的rowset加入到step的inputRowSets，和outputRowSets中。

combi.step = step;

steps.add(combi);

### Trans类startThreads( )

打开了所有的step线程，核心代码如下：

for (int i=0;i<steps.size();i++){

steps.get(i).step.start();

}

## Step执行

实现StepInterface的不同的step各个功能个不一样，但是它们之间也有一定的规律性。下图只列举了两个step，（TextInput）文本输入和Uniquerow(去重)。



### 启动

每一个具体的step启动线程时，自动调用run函数，它们统一调用基类的静态方法

**public** **void** run(){

BaseStep.*runStepThread*(**this**, meta, data);

}

### 处理

基类BaseStep采取了统一的处理方式，调用子类processRow以行为单位处理，核心代码如下。

while (stepInterface.processRow(meta, data) && !stepInterface.isStopped());

processRow( )通用过程是：调用基类BaseStep 的getRow( )得到数据，对一行数据处理，处理之后调用基类putRow( )方法数据保存至outputRowSets（即next step的inputRowSets）

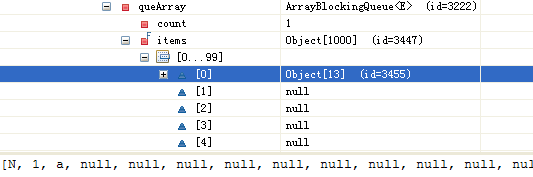
### 元数据与数据关系。

Trans中的ETL过程（每个step）以行为单位处理，其中行的元数据信息RowMeta和数据信息统一保存在RowSet对象中。

在RowSet中RowMeta的成员的调试结果如下。可见rowMeta储存了每列数据的名称和类型。第一列列名flag，数据是长度为1的String；第二列列名id…



RowSet的数据信息在queArray队列中，调试结果如下：可以看出第一个数据元素是一个Object包含了3列，数据内容为（N，1，a…）



# .kjb、.ktr文件格式及读写

## .kjb文件格式

.kjb是Job元数据文件的后缀。主要有三部分构成：job基本信息，job的entris组件信息和组件之间的链接hop信息。标签为job。

### job标签



job的基本信息包括name，描述信息，创建和修改日期等。

### entries标签

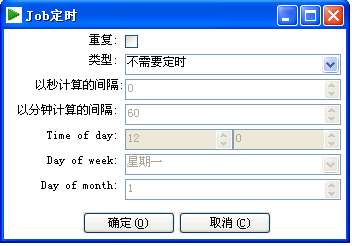
job的<entries>信息有两部分：

公有信息：每个entry都具有的信息。如name，description，jobentrytype等，由JobEntryBase保存。

私有信息：每个具体step继承了JobEntryBase并各自特有的信息，如开始entry的xml中其他信息，如下图所示。

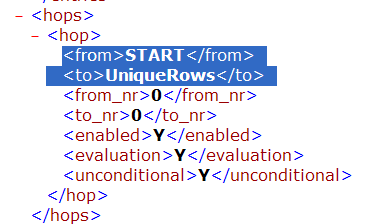


每个标签内容对应界面层元素如下图



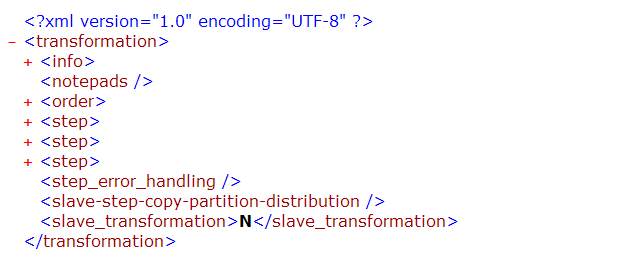
### hops标签

Hops信息主要有：源和目标entry信息；链接的状态，是否开启；链接的类型：条件执行还是无条件执行等。如下图所示：



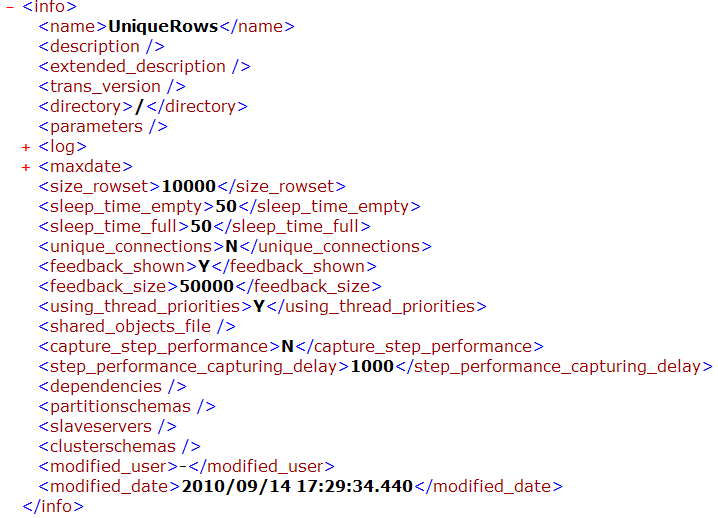
## .ktr文件格式

一个UniqueRiws.ktr 文件的Transformation顶层有Transformation信息<info> , <order>信息。<Step>信息组成。

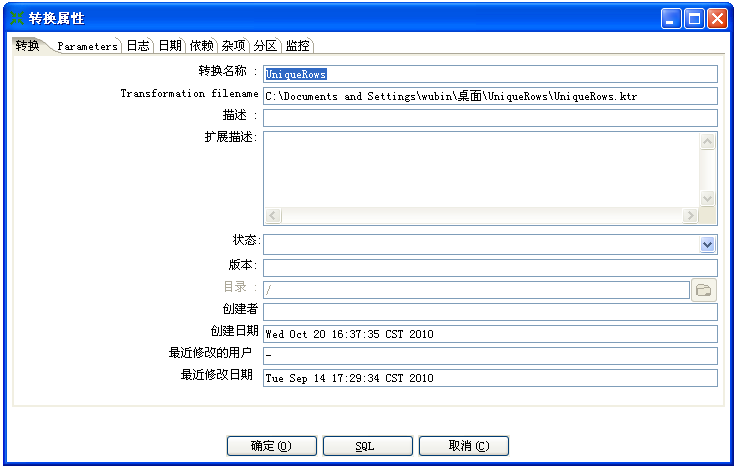


### info标签

<info>节点包括了一个Trans的信息。如trans名字，trans描述信息，日志设置信息，设置的最大RowSet数量，修改时间。

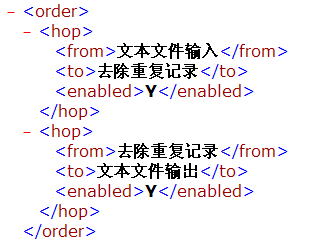


对应的界面层设置是：



### order标签

<order>对象是Step之间的链接信息，包括一个链接的源Step和目标Step，以及链接的使能状态。



### step标签

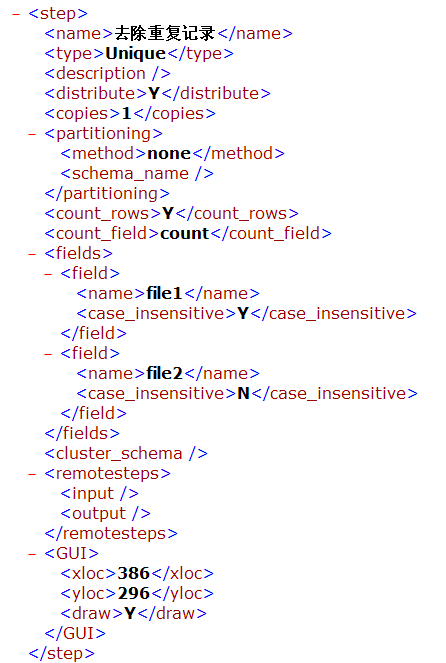
<step>节点信息描述了一个节点的信息。包括两部分信息：

1、每个step都有的信息：如stepname，step类型信息，所在面板位置信息，

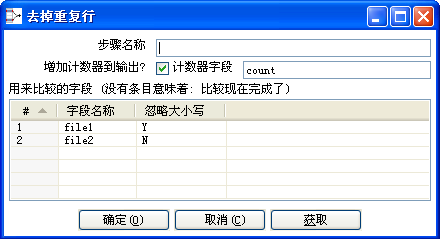
2、具体step特有的信息：如去重是否对重复的行计数，以及计数变量名。

文本输出的格式，分隔符，数据名和数据类型信息。

以下是去重step对应的<step>内容。

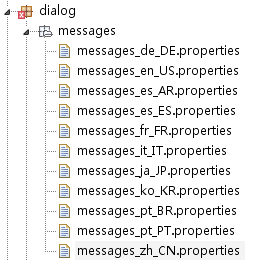


对应的GUI设置为：



# 配置文件

## 语言



位于src-ui中org.pentaho.di.ui.core.database.dialog.messages以及org.pentaho.di.ui.core. dialog.messages中messages\_xx\_xx.properties文件

包含各窗口文本内容在选择不同语言（中文、英语等）后对应的不同提示语句。

## 图标

### 图标位置

位于ui\images文件夹中。

### 配置文件

ui\下LAF.properties文件，内容如下：

splash\_image=ui/images/kettle\_splash.png

其中splash\_image为程序中用到的property的key值，返回值为ui/images/kettle\_splash.png。

### 代码说明

1、设置文件：

**package** org.pentaho.di.laf;

**public** **class** OverlayPropertyHandler **implements** PropertyHandler {

**protected** **static** **final** String *propFile* = "ui/laf.properties";

}

2、使用：

**final** Image kettle\_image = ImageUtil.*getImageAsResource*(display,

BasePropertyHandler.*getProperty*("splash\_image"));

## 系统常量

org.pentaho.di.core包中Const类。