# DAP原子操作开发手册

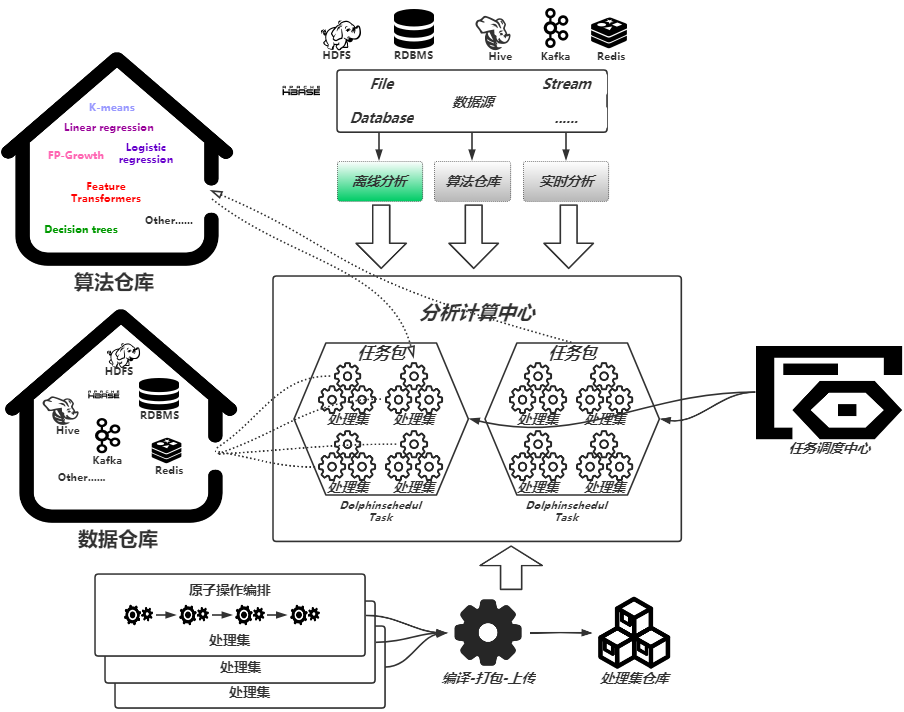
## **术语**

* 原子操作：最小执行单元；
* 操作组：一个或多个原子操作的编排，以组的形式体现；
* 操作集：一个或多个操作组的编排，以组的形式体现。用于生成最终执行的jar包；
* 生成器：将通过界面配置好的操作集结合模板生成对应的Scala源码；
* 执行器：提交至Spark执行的主类dps.mission.Launcher；

## 概述

原子操作是DAP（数据分析平台）最小执行单元，由一个或多个原子操作组成一个操作组，由一个或多个原子操作组成一个操作集。通过DAP的后台服务将操作集进行代码生成、编译、打包（形成可以在Spark上运行的jar文件）、部署（上传至hdfs或minIO）。DAP集成第三方调度框架（Apache Dolphinscheduler）通过调度框架对操作集进行编排最终生成一个可执行的任务。本文旨在帮助研发人员快速理解DAP的系统运行过程及开发步骤。

## **整体设计**



## **工程结构**

dps 父工程

dps-atomic 原子操作

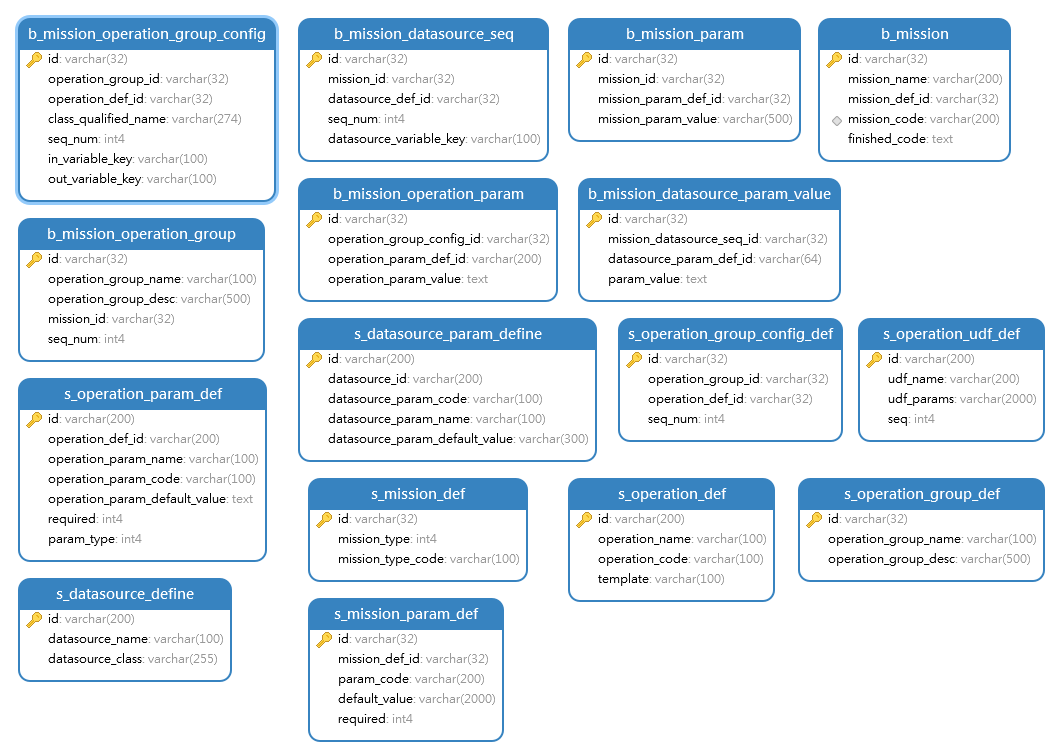
dps-generator 原子操作代码生成器

dps-datasource 数据源适配，数据源分两类：1、流数据源，2、非流数据源

dps-mission 执行器

dps-util 工具类

## **数据库设计**



表名以s\_开头的表为系统表也是原子操作的定义表。

表名以b\_开头的表为业务表也是操作集在执行中的配置表。

具体字段含意见数据库中字段上的注解，其中相关原子操作开发的表重点可关注s\_operation\_def、s\_operation\_param\_def、s\_operation\_udf\_def。

## 分类

原子操作大致分为五类：

1. 数据提取类

dps.atomic.impl.fetch

1. 数据存储类

dps.atomic.impl.store

1. RDD(Resilient Distributed Dataset)操作类

dps.atomic.impl.rdd

1. Dataset操作类

dps.atomic.impl.dataset

1. 定制操作类

dps.atomic.impl.custom

## **类编写约定**

* 原子操作类存放于[分类](#_分类)中描述的5类的包路径下，按开发者提供的功能自行辨别存放位置。
* 开发者所开发的原子操作类必须继承dps.atomic.impl.AbstractAction，并实现scala.Serializable接口，e.g:

**class** RDDString2Map(**override** **val** sparkSession: SparkSession, **override** **val** sparkConf:SparkConf,**override** **val** inputVariableKey: String, **override** **val** outputVariableKey: String, **override** **val** variables: Map[String, Any]) **extends** AbstractAction(sparkSession, sparkConf,inputVariableKey, outputVariableKey, variables) **with** Serializable {

......

}

* 重写方法doIt，doIt方法是DAP执行器执行时调用的方法，也是唯一调用方法

**def** doIt(params: Map[String, String]): Any = {

......

｝

* 重写方法define，define方法是对原子操作进行[定义](#_定义)，由原子操作工具类进行扫描并进行数据库数据初始化所执行的方法

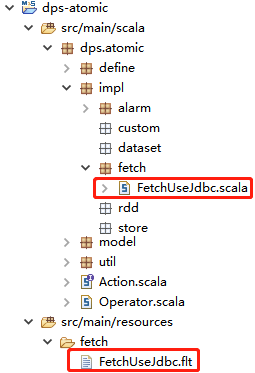
**def** define(): AtomOperationDefine = {

......

}

## **模板**

每个原子操作都有其对应的.ftl模板文件，e.g:



模板内容与其原子操作的代码几乎一样，操作集执行时并非执行dps-atomic工程中的原子操作而是执行由代码生成器生成在dps-mission中的代码。

原子操作Scala类内容示例：

**package** dps.atomic.impl.fetch

**import** java.text.SimpleDateFormat

**import** java.util.concurrent.TimeUnit

**import** java.util.concurrent.TimeUnit.{DAYS, HOURS, MINUTES, SECONDS}

**import** java.util.{Calendar, Date}

**import** com.fasterxml.jackson.databind.ObjectMapper

**import** com.fasterxml.jackson.module.scala.experimental.ScalaObjectMapper

**import** dps.atomic.define.{AtomOperationDefine, AtomOperationParamDefine}

**import** org.apache.spark.sql.SparkSession

**import** scala.collection.mutable.Map

**import** org.apache.spark.SparkConf

**import** dps.atomic.impl.AbstractAction

**import** dps.atomic.define.AtomOperationUdf

**import** dps.atomic.define.AtomOperationHasUdfDefine

**class** FetchUseJdbc(**override** **val** sparkSession: SparkSession, **override** **val** sparkConf:SparkConf,**override** **val** inputVariableKey: String, **override** **val** outputVariableKey: String, **override** **val** variables: Map[String, Any]) **extends** AbstractAction(sparkSession, sparkConf,inputVariableKey, outputVariableKey, variables) **with** Serializable {

**def** doIt(params: Map[String, String]): Any = {

**val** startTime = getTime(params.get("startTime").get)

**val** interval = Integer.valueOf(params.get("interval").getOrElse("0"))

**val** timeunit = TimeUnit.valueOf(params.get("timeunit").getOrElse("HOURS"))

**val** sql = s"(select \* from dual where d>='${getTimeWithHour(startTime)}' and d<'${getTime(getTimeWithHour(startTime), interval, timeunit)}')"

**val** om = **new** ObjectMapper **with** ScalaObjectMapper

**val** dsParams = om.readValue(params.get(params.get("dsParamsKey").get).get,classOf[Map[String,String]])

**val** dataset = sparkSession.sqlContext.read.format("jdbc")

.option("url", dsParams.get("url").get)

.option("driver", dsParams.get("driver").get)

.option("dbtable", sql)

.option("user", dsParams.get("user").get)

.option("password", dsParams.get("password").get)

.load();

dataset.createOrReplaceTempView(params.get("viewName").get)

variables.put(outputVariableKey, dataset)

}

**def** getTime(paramValue: String): String = {

**if** (paramValue == **null** || "".equals(paramValue.trim())) {

**val** sdf = **new** SimpleDateFormat("yyyy-MM-dd HH:mm:ss")

**return** sdf.format(**new** Date())

}

**return** paramValue;

}

**def** getTime(startTime: String, interval: Int, unit: TimeUnit): String = {

**if** (startTime == **null** || "".equals(startTime)) {

**return** getTime(startTime)

}

**val** sdf = **new** SimpleDateFormat("yyyy-MM-dd HH:mm:ss")

**val** date = sdf.parse(startTime)

**val** calendar = Calendar.getInstance

calendar.setTime(date)

**val** endTime = unit **match** {

**case** DAYS => {

calendar.add(Calendar.DAY\_OF\_MONTH, interval)

calendar.getTime

}

**case** HOURS => {

calendar.add(Calendar.HOUR, interval)

calendar.getTime

}

**case** MINUTES => {

calendar.add(Calendar.MINUTE, interval)

calendar.getTime

}

**case** SECONDS => {

calendar.add(Calendar.SECOND, interval)

calendar.getTime

}

**case** \_ => {

calendar.getTime

}

}

sdf.format(endTime)

}

**def** getTimeWithHour(paramValue: String): String = {

**if** (paramValue == **null** || "".equals(paramValue.trim())) {

**val** sdf = **new** SimpleDateFormat()

sdf.applyPattern("yyyy-MM-dd HH:mm:ss")

**val** calendar = Calendar.getInstance

calendar.set(Calendar.MINUTE, 0)

calendar.set(Calendar.SECOND, 0)

calendar.set(Calendar.MILLISECOND, 0)

**return** sdf.format(calendar.getTime)

}

**return** paramValue

}

**override** **def** define: AtomOperationDefine = {

**val** params = Map(

"dsParamsKey" -> **new** AtomOperationParamDefine("ds.params.key", "dsKey", **true**, stringType),//JDBC

"sql" -> **new** AtomOperationParamDefine("sql", "select \* from dual", **true**, sqlType),

"viewName" -> **new** AtomOperationParamDefine("view.name", "dual", **true**, stringType),

"startTime" -> **new** AtomOperationParamDefine("start.time", "yyyy-MM-dd HH:mm:ss", **false**, stringType),

"interval" -> **new** AtomOperationParamDefine("interval", "1", **false**, integerType),

"timeunit" -> **new** AtomOperationParamDefine("time.unit", ",DAYS,HOURS,MINUTES,SECONDS", **false**, listType))

**val** udfs = Seq(**new** AtomOperationUdf("getTime",Seq(classOf[String].getName)),

**new** AtomOperationUdf("getTime",Seq(classOf[String].getName,classOf[Int].getName,classOf[TimeUnit].getName)),

**new** AtomOperationUdf("getTimeWithHour",Seq(classOf[String].getName))

)

**val** atomOperation = **new** AtomOperationHasUdfDefine("fetch.use.jdbc", "fetchUseJdbc", "fetch/FetchUseJdbc.flt", params.toMap,udfs)

atomOperation.id = "fetch\_use\_jdbc"

**return** atomOperation

}

}

原子操作模板内容示例：

package ${packagePath}

import java.text.SimpleDateFormat

import java.util.concurrent.TimeUnit

import java.util.concurrent.TimeUnit.{DAYS, HOURS, MINUTES, SECONDS}

import java.util.{Calendar, Date}

import com.fasterxml.jackson.databind.ObjectMapper

import com.fasterxml.jackson.module.scala.experimental.ScalaObjectMapper

import dps.atomic.define.{AtomOperationDefine, AtomOperationParamDefine}

import org.apache.spark.sql.SparkSession

import dps.atomic.impl.AbstractAction

import scala.collection.mutable.Map

import org.apache.spark.SparkConf

class ${className}(override val sparkSession: SparkSession, override val sparkConf:SparkConf,override val inputVariableKey: String, override val outputVariableKey: String, override val variables: Map[String, Any]) extends AbstractAction(sparkSession, sparkConf,inputVariableKey, outputVariableKey, variables) with Serializable {

def doIt(params: Map[String, String]): Any = {

val startTime = getTime(params.get("startTime").get)

val interval = Integer.valueOf(params.get("interval").getOrElse("0"))

val timeunit = TimeUnit.valueOf(params.get("timeunit").getOrElse("HOURS"))

val sql = s"""(${sql}) as ${className}\_view"""

val dataset = sparkSession.sqlContext.read.format("jdbc")

.option("url", params.get("url").get)

.option("driver", params.get("driver").get)

.option("dbtable", sql)

.option("user", params.get("user").get)

.option("password", params.get("password").get)

.load();

dataset.createOrReplaceTempView(params.get("viewName").get)

variables.put(outputVariableKey, dataset)

}

def getTime(paramValue: String): String = {

if (paramValue == null || "".equals(paramValue.trim())) {

val sdf = new SimpleDateFormat("yyyy-MM-dd HH:mm:ss")

return sdf.format(new Date())

}

return paramValue;

}

def getTime(startTime: String, interval: Int, unit: TimeUnit): String = {

if (startTime == null || "".equals(startTime)) {

return getTime(startTime)

}

val sdf = new SimpleDateFormat("yyyy-MM-dd HH:mm:ss")

val date = sdf.parse(startTime)

val calendar = Calendar.getInstance

calendar.setTime(date)

val endTime = unit match {

case DAYS => {

calendar.add(Calendar.DAY\_OF\_MONTH, interval)

calendar.getTime

}

case HOURS => {

calendar.add(Calendar.HOUR, interval)

calendar.getTime

}

case MINUTES => {

calendar.add(Calendar.MINUTE, interval)

calendar.getTime

}

case SECONDS => {

calendar.add(Calendar.SECOND, interval)

calendar.getTime

}

case \_ => {

calendar.getTime

}

}

sdf.format(endTime)

}

def getTimeWithHour(paramValue: String): String = {

if (paramValue == null || "".equals(paramValue.trim())) {

val sdf = new SimpleDateFormat()

sdf.applyPattern("yyyy-MM-dd HH:mm:ss")

val calendar = Calendar.getInstance

calendar.set(Calendar.MINUTE, 0)

calendar.set(Calendar.SECOND, 0)

calendar.set(Calendar.MILLISECOND, 0)

return sdf.format(calendar.getTime)

}

return paramValue

}

}

模板部分代码中高亮部分的表达式最终会由代码生成器将此部分的类型替换为配置中的值，以上类示例及模板示例中重点对比doIt方法中第4行val sql = ......

## **定义**

开发者在开发好了原子操作后，必须重写define方法，e.g:

**override** **def** define: AtomOperationDefine = {

**val** params = Map(

"dsParamsKey" -> **new** AtomOperationParamDefine("ds.params.key", "dsKey", **true**, stringType),//JDBC

"sql" -> **new** AtomOperationParamDefine("sql", "select \* from dual", **true**, sqlType),

"viewName" -> **new** AtomOperationParamDefine("view.name", "dual", **true**, stringType),

"startTime" -> **new** AtomOperationParamDefine("start.time", "yyyy-MM-dd HH:mm:ss", **false**, stringType),

"interval" -> **new** AtomOperationParamDefine("interval", "1", **false**, integerType),

"timeunit" -> **new** AtomOperationParamDefine("time.unit", ",DAYS,HOURS,MINUTES,SECONDS", **false**, listType))

**val** udfs = Seq(**new** AtomOperationUdf("getTime",Seq(classOf[String].getName)),

**new** AtomOperationUdf("getTime",Seq(classOf[String].getName,classOf[Int].getName,classOf[TimeUnit].getName)),

**new** AtomOperationUdf("getTimeWithHour",Seq(classOf[String].getName))

)

**val** atomOperation = **new** AtomOperationHasUdfDefine("fetch.use.jdbc", "fetchUseJdbc", "fetch/FetchUseJdbc.ftl", params.toMap,udfs)

atomOperation.id = "fetch\_use\_jdbc"

**return** atomOperation

}

以上代码可结合[数据库](#_数据库设计)理解各字段的作用，原子操作的ID由定义指定（atomOperation.id = "fetch\_use\_jdbc"），而非数据库生成UUID或自增序号。

代码段中 params 的定义表示该原子操作在执行的过程中需要使用到的参数，如无需参数在new AtomOperationDefine 的构造函数中传入一个空Map即可

代码段中 udfs 的定义表示该原子操作提供了哪些可供调用的函数，如有可供调用的函数使用new AtomOperationHasUdfDefine进行构造，如没有可供调用的函数则直接使用new AtomOperationDefine进行构造

AtomOperationParamDefine对应的参数作用：

* val operationParamName:String——原子操作的参数国际化key
* val operationParamDefaultValue:String——默认值
* val required:Boolean——是否必填
* val operationParamType:String——参数类型（1:数字类型; 2:字符类型; 3:列表类型; 4:文本类型->sql; 5:文本类型->scala; 6:文本类型->javascript）

AtomOperationDefine及 AtomOperationHasUdfDefine对应的参数作用

* val operationName: String——原子操作名的国际化key
* val operationCode: String——原子操作的编码，用于组合在业务中生成的ID
* val template: String——对应的模板文件
* val operationParams: Map[String, AtomOperationParamDefine]——原子操作所需要使用到的参数
* val udfs: Seq[AtomOperationUdf]——原子操作可供调用的函数定义

## **原子操作发布**

dps-atomic工程下dps.atomic.util.AtomOperationUtil可对定义好的原子操作进行数据库数据初始化的操作，该工具类会扫描dps.atomic.impl及其子包路径下所有继承dps.atomic.impl.AbstractAction的实现类，并根据其defin方法中的定义进行数据写入，以备界面上进行配置