文档修订历史

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **版本** | **作者** | **工作描述** | **修订历史** | **修改日期** |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |

目录

[文档修订历史 1](#_Toc522783154)

[CBSS任务调度学习摘录 3](#_Toc522783155)

[1 表结构说明 3](#_Toc522783156)

[1.1 任务定义表(Task-Define) 3](#_Toc522783157)

[1.2 任务计划定义表（Task\_Define\_Plan） 4](#_Toc522783158)

[1.3 任务执行日志（Task\_Plan\_Log） 5](#_Toc522783159)

[2 程序实现说明 6](#_Toc522783160)

[2.1 配置文件设置 6](#_Toc522783161)

[2.1.1 配置文件扫搜路径设置 6](#_Toc522783162)

[2.1.2 配置文件格式实例 7](#_Toc522783163)

[2.2 程序说明 9](#_Toc522783164)

[2.2.1 程序结构图 9](#_Toc522783165)

[2.2.2 主要程序代码说明 9](#_Toc522783166)

CBSS任务调度学习摘录

# 表结构说明



## 任务定义表(Task-Define)

|  |  |  |  |
| --- | --- | --- | --- |
| **Code** | **Name** | **Domain** | **Comment** |
| TASK\_ID | 任务ID | ID域-短-UID |  |
| TASK\_NAME | 任务名称 | 名称域-长-S50 |  |
| PROG\_TYPE | 任务执行方式 | 类型域-C3 | PTP 存储过程  PTL LIB  PTC 执行命令  PTS WEBSERVICE  PTI 内部调用  CODE\_NAME.CODE\_TYPE=“PROG\_TYPE” |
| TASK\_OBJECT | 程序名称 | 描述域-短-TEXT |  |
| INPUT\_PARAM | 输入参数 | 字符串域-超长-Text | 多个参数用逗号分隔 |
| REMARK | REMARK | 字符串域-超长-Text |  |
| STATE | 状态 | 状态域-C3 | 00A 有效  00X 无效 |
| STATE\_DATE | 状态时间 | 时间域-日期时间-DT | 状态变更时间 |
| OPERATER\_ID | 操作者 | ID域-短-UID |  |

## 任务计划定义表（Task\_Define\_Plan）

|  |  |  |  |
| --- | --- | --- | --- |
| **Code** | **Name** | **Domain** | **Comment** |
| TASK\_PLAN\_ID | 任务计划ID | ID域-中-UID |  |
| TASK\_PLAN\_NAME | 任务计划名称 | 名称域-长-S50 |  |
| PLAN\_TYPE | 计划类型 | 编码域-短-S6 | CRON 定时  SIMPLE 间隔  CODE\_NAME.CODE\_TYPE=“PLAN\_TYPE”' |
| INTERVAL\_AMOUNT | 间隔数 | 数量域-短-N5 |  |
| INTERVAL\_UNIT | 间隔单位 | 编码域-短-S6 |  |
| CRON\_EXPRESSION | 定时表达式 | 字符串域-超长-Text |  |
| NEXT\_START\_DATETIME | 下次执行日期时间 | 时间域-日期时间-DT | 若计费类型为间隔，不能为空需要填写，第一次的数据作为第一次执行的日期时间，每次任务执行完更新本字段，值为上次执行完的时间+间隔数得到。  计费类型为定时，第一次生成进去，任务执行完后，生成下一的日期时间。 |
| TASK\_ID | 任务ID | ID域-短-UID |  |
| STATE | 状态 | 状态域-C3 | 状态 |
| STATE\_DATE | 状态时间 | 时间域-日期时间-DT | 状态变更时间 |
| REMARK | 备注 | 字符串域-超长-Text |  |
| OPERATER\_ID | 操作者 | ID域-短-UID |  |
| OPERATE\_DATE | 操作时间 | 时间域-日期时间-DT |  |

## 任务执行日志（Task\_Plan\_Log）

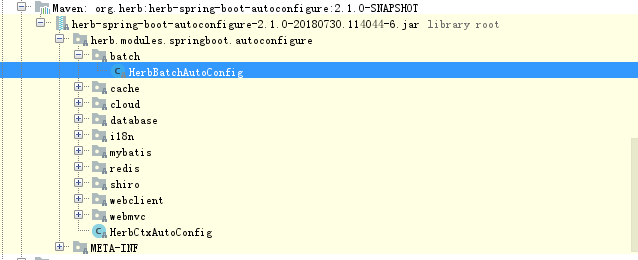
|  |  |  |  |
| --- | --- | --- | --- |
| **Code** | **Name** | **Domain** | **Comment** |
| TASK\_PLAN\_LOG\_ID | 任务计划明细ID | ID域-短-UID |  |
| TASK\_ID | 任务ID | ID域-短-UID |  |
| TASK\_NAME | 任务名称 | 名称域-长-S50 |  |
| TASK\_PLAN\_ID | 任务计划ID | ID域-中-UID |  |
| TASK\_PLAN\_NAME | 任务计划名称 | 名称域-长-S50 |  |
| TASK\_OBJECT | 程序名称 | 描述域-短-TEXT |  |
| INPUT\_PARAM | 输入参数 | 字符串域-超长-Text | 多个参数用逗号分隔 |
| START\_DATETIME | 开始时间 | 时间域-日期时间-DT |  |
| END\_DATETIME | 结束时间 | 时间域-日期时间-DT |  |
| FINISHED\_FLAG | 完成标志 | 布尔域-C1 |  |
| RETRY\_TIMES | 执行次数 | 数量域-短-N5 |  |
| EXEC\_STATE | 执行标志 | 状态域-C3 | 成功：T  失败：F  对于存储过程，约定出参o\_msg =’0‘ 表示成功，o\_msg!='0’表示出错  对于非存储过程调用，约定有捕获异常标识失败，未捕获异常表示成功 |
| EXEC\_MSG | 执行信息 | 字符串域-超长-Text | 对于存储过程，约定出参o\_msg =’0‘ 表示成功，o\_msg!='0’表示出错，该字段记录具体的o\_msg信息  对于非存储过程调用，约定有捕获异常标识失败，未捕获异常表示成功，成功时改字段记录返回值，失败时记录异常信息 |

# 程序实现说明

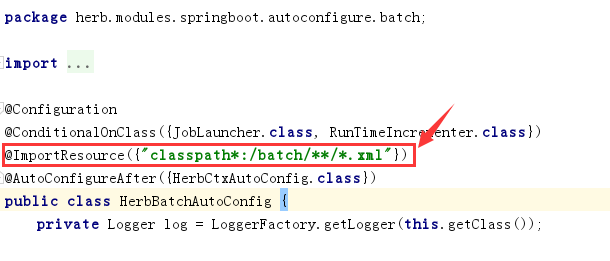
## 配置文件设置

### 配置文件扫搜路径设置

在包org.herb-spring-boot-autoconfig:2.1.0-SNAPSHOT:



具体程序如下：



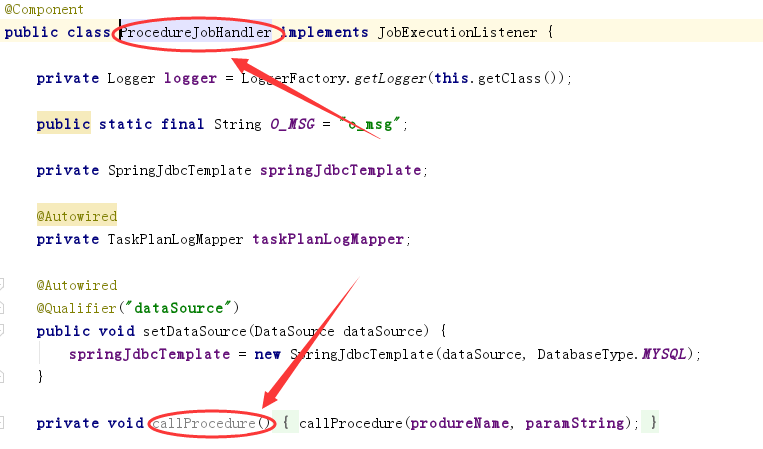
### 配置文件格式实例

|  |
| --- |
| <beans xmlns="http://www.springframework.org/schema/beans"  xmlns:batch="http://www.springframework.org/schema/batch"  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  xsi:schemaLocation="http://www.springframework.org/schema/batch  http://www.springframework.org/schema/batch/spring-batch-3.0.xsd  http://www.springframework.org/schema/beans  http://www.springframework.org/schema/beans/spring-beans-3.2.xsd">  <batch:job id="procedureJob">  <batch:step id="procedureJob.step" parent="procedureJobStep"/>  <batch:listeners>  <batch:listener ref="procedureJobHandler"/>  </batch:listeners>  </batch:job>  <!--退款 -->  <batch:step id="procedureJobStep">  <batch:tasklet transaction-manager="jobTxManager">  <bean class="org.springframework.batch.core.step.tasklet.MethodInvokingTaskletAdapter" scope="step">  <property name="targetObject" ref="procedureJobHandler"/>  <property name="targetMethod" value="callProcedure"/>  </bean>  </batch:tasklet>  </batch:step>  </beans> |

说明：

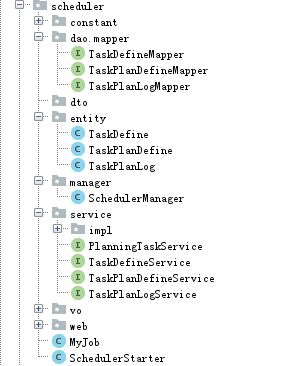
1. <batch:step id="procedureJob.step" parent="procedureJobStep"/> 的“parent“的值与<batch:step id="procedureJobStep">的“step id” 的值一致；
2. <batch:listener ref="procedureJobHandler"/>的“ref”的值是一个类名，与<property name="targetObject" ref="procedureJobHandler"/>的“ref” 的值相同
3. <property name="targetMethod" value="callProcedure"/>的“value”的值是上面<batch:listener ref="procedureJobHandler"/>的“ref”所指类下的一个方法

该类程序示例如下图：



## 程序说明

### 程序结构图



这里主程序是SchedulerStarter,通过来挂接spring-boot框架执行程序。

### 主要程序代码说明

#### SchedulerStarter

|  |
| --- |
| @Component  public class SchedulerStarter implements ApplicationListener<ApplicationReadyEvent> {  private static final Logger logger = LoggerFactory.getLogger(SchedulerStarter.class);  @Autowired  TaskPlanDefineService taskPlanDefineService;  @Autowired  SchedulerManager schedulerManager;  @Autowired  private TaskDefineService taskDefineService;  @Value("${cbss.scheduler.start:false}")  private boolean autoStart;  public void startAllJob() {  List<TaskPlanDefine> taskPlanDefines = taskPlanDefineService.queryValidTaskPlanDefine();  if (0 == taskPlanDefines.size()) {  logger.info("任务计划定义表没有配置有效的定时任务");  return;  }  for (TaskPlanDefine taskPlanDefine : taskPlanDefines) {  TaskDefine taskDefine = taskDefineService.queryByPk(taskPlanDefine.getTaskId());  taskPlanDefine.setInputParam(taskDefine.getInputParam());  taskPlanDefine.setTaskName(taskDefine.getTaskName());  taskPlanDefine.setProgType(taskDefine.getProgType());  taskPlanDefine.setTaskObject(taskDefine.getTaskObject());  }  schedulerManager.startJobs(taskPlanDefines);  }  @Override  public void onApplicationEvent(ApplicationReadyEvent event) {  logger.info("@@@@@@@@@@@@ >> 任务调度自动执行：{} ",autoStart);  if (autoStart) {  startAllJob();  }  }  } |

改程序是通过实现ApplicationListener接口，重载onApplicationEvent来实现启动所有job。核心代码：

|  |
| --- |
| public void onApplicationEvent(ApplicationReadyEvent event) {  logger.info("@@@@@@@@@@@@ >> 任务调度自动执行：{} ",autoStart);  if (autoStart) {  startAllJob();  }  } |

#### SchedulerManager

|  |
| --- |
| @Component  public class SchedulerManager {  // 通过SchedulerFactory获取一个调度器实例  private static SchedulerFactory schedulerFactory = new StdSchedulerFactory();  private Scheduler scheduler;  public final synchronized Scheduler getScheduler() throws SchedulerException {  if(scheduler==null){  scheduler = schedulerFactory.getScheduler();  }  return scheduler;  }  /\*\*  \* @Description:开始所有定时任务  \*/  public void startJobs(List<TaskPlanDefine> taskPlanDefines) {  // List<TaskPlanDefine> taskPlanDefines = taskPlanDefineService.queryValidTaskPlanDefine();  try {  getScheduler().start();  //定时任务  for (TaskPlanDefine taskPlanDefine : taskPlanDefines) {  addAJob(taskPlanDefine);  }  } catch (SchedulerException e) {  e.printStackTrace();  }  }  /\*\*  \* @Description: 添加一个定时任务  \*/  public void addAJob(TaskPlanDefine taskPlanDefine) {  Trigger trigger = null;  try {  Scheduler sched = getScheduler();  String jobId = taskPlanDefine.getTaskPlanId();  JobKey jobKey = new JobKey(jobId);  TriggerKey triggerKey = new TriggerKey(jobId);  //判断键值是否存在  if(sched.checkExists(triggerKey)||sched.checkExists(triggerKey)){  throw new BizException(MessageFormat.format("jobId:[{0}]在调度中已经存在！", jobId));  }  JobDetail job = JobBuilder.newJob(MyJob.class).withIdentity(jobKey)  .usingJobData("taskPlanName", taskPlanDefine.getTaskPlanName())  .usingJobData("taskName", taskPlanDefine.getTaskName())  .usingJobData("taskType", taskPlanDefine.getPlanType())  .usingJobData("progType", taskPlanDefine.getProgType())  .usingJobData("taskObject", taskPlanDefine.getTaskObject())  .usingJobData("inputParam", taskPlanDefine.getInputParam())  .usingJobData("taskPlanId", taskPlanDefine.getTaskPlanId())  .usingJobData("taskId", taskPlanDefine.getTaskId())  .build();  if (taskPlanDefine.getPlanType().equals(SchedulerConstant.CRON\_CYCLE)) { //定时cron任务  trigger = (CronTrigger) TriggerBuilder.newTrigger().withIdentity(triggerKey).withSchedule(  CronScheduleBuilder.cronSchedule(taskPlanDefine.getCronExpression()))  .usingJobData("taskPlanName", taskPlanDefine.getTaskPlanName())  .usingJobData("taskName", taskPlanDefine.getTaskName())  .usingJobData("taskType", taskPlanDefine.getPlanType())  .usingJobData("progType", taskPlanDefine.getProgType())  .usingJobData("taskObject", taskPlanDefine.getTaskObject())  .usingJobData("inputParam", taskPlanDefine.getInputParam())  .usingJobData("taskPlanId", taskPlanDefine.getTaskPlanId())  .usingJobData("taskId", taskPlanDefine.getTaskId())  .build();  }else if (taskPlanDefine.getPlanType().equals(SchedulerConstant.SIMPLE\_CYCLE)){  Date runTime = DateBuilder.evenSecondDate(taskPlanDefine.getNextStartDatetime());  SimpleScheduleBuilder simpleScheduleBuilder = getSimpBuilder(taskPlanDefine.getIntervalUnit(), taskPlanDefine.getIntervalAmount());  trigger = (SimpleTrigger) TriggerBuilder.newTrigger().withIdentity(triggerKey)  .startAt(runTime).withSchedule(simpleScheduleBuilder)  .usingJobData("taskPlanName", taskPlanDefine.getTaskPlanName())  .usingJobData("taskName", taskPlanDefine.getTaskName())  .usingJobData("taskType", taskPlanDefine.getPlanType())  .usingJobData("progType", taskPlanDefine.getProgType())  .usingJobData("taskObject", taskPlanDefine.getTaskObject())  .usingJobData("inputParam", taskPlanDefine.getInputParam())  .usingJobData("taskPlanId", taskPlanDefine.getTaskPlanId())  .usingJobData("taskId", taskPlanDefine.getTaskId())  .build();  } else{  throw new BizException(MessageFormat.format("计划类型:[{0}]错误！", taskPlanDefine.getPlanType()));  }  sched.scheduleJob(job, trigger);  } catch (Exception e) {  throw new RuntimeException(e);  }  }  /\*\*  \* @param jobId  \* @Description: 移除一个任务  \*/  public void removeJob(String jobId) {  try {  Scheduler sched = getScheduler();  if(!(sched.checkExists(JobKey.jobKey(jobId)) && sched.checkExists(TriggerKey.triggerKey(jobId)))){  throw new BizException(MessageFormat.format("jobId:[{0}]在调度中不存在！", jobId));  };  sched.deleteJob(JobKey.jobKey(jobId));  sched.unscheduleJob(TriggerKey.triggerKey(jobId));  } catch (Exception e) {  throw new RuntimeException(e);  }  }  /\*\*  \* @param jobId  \* @Description: 暂停一个任务  \*/  public void pauseJob(String jobId) {  try {  Scheduler sched = getScheduler();  if(!(sched.checkExists(JobKey.jobKey(jobId)) && sched.checkExists(TriggerKey.triggerKey(jobId)))){  throw new BizException(MessageFormat.format("jobId:[{0}]在调度中不存在！", jobId));  };  sched.pauseJob(JobKey.jobKey(jobId));  } catch (Exception e) {  throw new RuntimeException(e);  }  }  /\*\*  \* @param jobId  \* @Description: 恢复一个任务  \*/  public void resumeJob(String jobId) {  try {  Scheduler sched = getScheduler();  if(!sched.checkExists(TriggerKey.triggerKey(jobId))){  throw new BizException(MessageFormat.format("jobId:[{0}]在调度中不存在！", jobId));  };  sched.resumeTrigger(TriggerKey.triggerKey(jobId));  } catch (Exception e) {  throw new RuntimeException(e);  }  }  /\*\*job执行一次  \* @param jobId  \* @Description: j  \*/  public void runOnceJob(String jobId) {  try {  Scheduler sched = getScheduler();  if(!(sched.checkExists(JobKey.jobKey(jobId)) )){  throw new BizException(MessageFormat.format("jobId:[{0}]在调度中不存在！", jobId));  }  sched.triggerJob(JobKey.jobKey(jobId));  } catch (Exception e) {  throw new RuntimeException(e);  }  }  /\*\*  \* @param jobId  \* @Description: 修改简单周期任务时间  \*/  public void modSimpTime(String jobId, Date nextTime, String intervalUnit, int intervalNum) {  try {  Scheduler sched = getScheduler();  TriggerKey triggerKey = TriggerKey.triggerKey(jobId);  //判断键值是否存在  if (!(sched.checkExists(triggerKey) && sched.checkExists(triggerKey))){  throw new BizException(MessageFormat.format("jobId:[{0}]在调度中不存在！", jobId));  };  SimpleTrigger trigger = (SimpleTrigger) sched.getTrigger(triggerKey);  Date runTime = DateBuilder.evenSecondDate(nextTime);  SimpleScheduleBuilder simpleScheduleBuilder = getSimpBuilder(intervalUnit, intervalNum);  trigger = trigger.getTriggerBuilder().withIdentity(triggerKey)  .startAt(runTime).withSchedule(simpleScheduleBuilder)  .build();  //按新的trigger重新设置job执行  sched.rescheduleJob(triggerKey, trigger);  } catch (Exception e) {  throw new RuntimeException(e);  }  }  /\*\*  \* @param jobId  \* @Description: 修改周期任务时间  \*/  public void modCronTime(String jobId, String cronTime) {  try {  Scheduler sched = getScheduler();  TriggerKey triggerKey = TriggerKey.triggerKey(jobId);  //判断键值是否存在  if (!(sched.checkExists(triggerKey) && sched.checkExists(triggerKey))){  throw new BizException(MessageFormat.format("jobId:[{0}]在调度中不存在！", jobId));  };  CronTrigger trigger = (CronTrigger) sched.getTrigger(triggerKey);  //表达式调度构建器  CronScheduleBuilder scheduleBuilder = CronScheduleBuilder.cronSchedule(cronTime);  //按新的cronExpression表达式重新构建trigger  trigger = trigger.getTriggerBuilder().withIdentity(triggerKey)  .withSchedule(scheduleBuilder).build();  //按新的trigger重新设置job执行  sched.rescheduleJob(triggerKey, trigger);  } catch (Exception e) {  throw new RuntimeException(e);  }  }  /\*\*  \* @Description:关闭所有定时任务  \*/  public void shutdownJobs() {  try {  Scheduler sched = getScheduler();  if (!sched.isShutdown()) {  sched.shutdown();  }  } catch (Exception e) {  throw new RuntimeException(e);  }  }  /\*\*  \* @Description: 获得简单时间周期  \*/  public SimpleScheduleBuilder getSimpBuilder(String intervalUnit, int intervalNum) {  SimpleScheduleBuilder simpleScheduleBuilder;  switch (intervalUnit) {  case SchedulerConstant.INTERVAL\_UNIT\_SECOND:  simpleScheduleBuilder = SimpleScheduleBuilder.simpleSchedule().withIntervalInSeconds(intervalNum)  .repeatForever();  break;  case SchedulerConstant.INTERVAL\_UNIT\_MINUTE:  simpleScheduleBuilder = SimpleScheduleBuilder.simpleSchedule().withIntervalInMinutes(intervalNum)  .repeatForever();  break;  case SchedulerConstant.INTERVAL\_UNIT\_HOUR:  simpleScheduleBuilder = SimpleScheduleBuilder.simpleSchedule().withIntervalInHours(intervalNum)  .repeatForever();  break;  case SchedulerConstant.INTERVAL\_UNIT\_DAY:  simpleScheduleBuilder = SimpleScheduleBuilder.simpleSchedule().withIntervalInHours(intervalNum \* 24)  .repeatForever();  break;  default:  simpleScheduleBuilder = SimpleScheduleBuilder.simpleSchedule().withIntervalInHours(intervalNum)  .repeatForever();  }  return simpleScheduleBuilder;  }  /\*\*  \* 获取所有计划中的任务  \* \*\*/  public List<PlanningTaskVO> getPlanningTask() {  try {  Scheduler sched = getScheduler();  GroupMatcher<TriggerKey> matcher =GroupMatcher.anyTriggerGroup();  Set<TriggerKey> triggerKeys = sched.getTriggerKeys(matcher);  List<PlanningTaskVO> planningTaskVOs =new ArrayList<PlanningTaskVO>();  for(TriggerKey triggerKey : triggerKeys){  Trigger trigger = sched.getTrigger(triggerKey);  if(trigger.getJobDataMap().getString("taskName")==null){  //这种是立即运行的，不用显示  continue;  }  PlanningTaskVO planningTaskVO = new PlanningTaskVO();  planningTaskVO.setJobKey(trigger.getJobKey());  planningTaskVO.setTriggerKey(trigger.getKey());  planningTaskVO.setTaskPlanName(trigger.getJobDataMap().getString("taskName"));  planningTaskVO.setTaskName(trigger.getJobDataMap().getString("taskName"));  planningTaskVO.setInputParam(trigger.getJobDataMap().getString("inputParam"));  planningTaskVO.setTaskObject(trigger.getJobDataMap().getString("taskObject"));  planningTaskVO.setProgType(trigger.getJobDataMap().getString("progType"));  String execState = sched.getTriggerState(trigger.getKey()).name();  planningTaskVO.setExecState(execState);  planningTaskVO.setNextFireTime(trigger.getNextFireTime());  planningTaskVO.setTaskPlanId(triggerKey.getName());  if (trigger instanceof CronTrigger) {  CronTrigger cronTrigger = (CronTrigger) trigger;  String cronExpression = cronTrigger.getCronExpression();  planningTaskVO.setPlanTimeExpress(cronExpression);  planningTaskVO.setPlanType(SchedulerConstant.CRON\_CYCLE);  }else {  SimpleTrigger simpleTrigger = (SimpleTrigger) trigger;  Long intervals = simpleTrigger.getRepeatInterval();  planningTaskVO.setPlanTimeExpress(DateTimeUtil.formatDuration(intervals));  planningTaskVO.setPlanType(SchedulerConstant.SIMPLE\_CYCLE);  }  planningTaskVOs.add(planningTaskVO);  }  return planningTaskVOs;  } catch (Exception e) {  throw new RuntimeException(e);  }  }  /\*\*  \* 获取所有运行中的任务  \* \*\*/  public List<RunningTaskVO> getRunningTask() {  try {  Scheduler sched = getScheduler();  List<JobExecutionContext> executingJobs = sched.getCurrentlyExecutingJobs();  List<RunningTaskVO> runningTaskVOs = new LinkedList<RunningTaskVO>();  for(JobExecutionContext executingJob : executingJobs){  RunningTaskVO runningTaskVO = new RunningTaskVO();  Trigger trigger = executingJob.getTrigger();  JobDetail job = executingJob.getJobDetail();  runningTaskVO.setJobKey(trigger.getJobKey());  runningTaskVO.setTriggerKey(trigger.getKey());  if(trigger.getJobDataMap().getString("taskName")==null){  runningTaskVO.setTaskPlanName(job.getJobDataMap().getString("taskName"));  runningTaskVO.setTaskName(job.getJobDataMap().getString("taskName"));  }else{  runningTaskVO.setTaskPlanName(trigger.getJobDataMap().getString("taskName"));  runningTaskVO.setTaskName(job.getJobDataMap().getString("taskName"));  }  // runningTaskVO.setTaskName(trigger.getJobDataMap().getString("taskName"));  runningTaskVO.setInputParam(trigger.getJobDataMap().getString("inputParam"));  runningTaskVO.setTaskObject(trigger.getJobDataMap().getString("taskObject"));  runningTaskVO.setProgType(trigger.getJobDataMap().getString("progType"));  String execState = sched.getTriggerState(trigger.getKey()).name();  runningTaskVO.setExecState(execState);  runningTaskVO.setNextFireTime(trigger.getNextFireTime());  runningTaskVO.setStartTime(trigger.getStartTime());  runningTaskVO.setTaskPlanId(trigger.getKey().getName());  if(trigger.getJobDataMap().getString("taskName")==null){  runningTaskVO.setPlanType(SchedulerConstant.ONCE\_CYCLE);  }else if(trigger instanceof CronTrigger) {  CronTrigger cronTrigger = (CronTrigger) trigger;  String cronExpression = cronTrigger.getCronExpression();  runningTaskVO.setPlanTimeExpress(cronExpression);  runningTaskVO.setPlanType(SchedulerConstant.CRON\_CYCLE);  }else {  SimpleTrigger simpleTrigger = (SimpleTrigger) trigger;  Long intervals = simpleTrigger.getRepeatInterval();  runningTaskVO.setPlanTimeExpress(DateTimeUtil.formatDuration(intervals));  runningTaskVO.setPlanType(SchedulerConstant.SIMPLE\_CYCLE);  }  runningTaskVOs.add(runningTaskVO);  }  return runningTaskVOs;  } catch (Exception e) {  throw new RuntimeException(e);  }  }  } |

该程序主要说明如下：

1. 通过SchedulerFactory获取一个调度器实例，并且有且只有一个实例，通过程序public final synchronized Scheduler getScheduler() 实现
2. 调度实例执行过程是：启动调度-》初始化JOB-》运行JOB

* 启动调度

getScheduler().start();

* 初始化JOB

JobDetail job = JobBuilder.newJob0

trigger = (CronTrigger) TriggerBuilder.newTrigger().withIdentity(triggerKey).withSchedule

或者

trigger = (SimpleTrigger) TriggerBuilder.newTrigger().withIdentity(triggerKey)

.startAt(runTime).withSchedule(simpleScheduleBuilder)

* 运行JOB

sched.scheduleJob(job, trigger)

1. 调度其它操作：

* 移除JOB

sched.deleteJob(JobKey.jobKey(jobId));

sched.unscheduleJob(TriggerKey.triggerKey(jobId));

* 暂停JOB

sched.pauseJob(JobKey.jobKey(jobId));

* 恢复JOB

sched.resumeTrigger(TriggerKey.triggerKey(jobId));

* 执行一次JOB

sched.triggerJob(JobKey.jobKey(jobId));

* 重置JOB

sched.rescheduleJob(triggerKey, trigger);

* 关闭所有JOB

sched.shutdown();

#### MyJob

|  |
| --- |
| public class MyJob implements Job {  TaskPlanLogService taskPlanLogService= SpringContextUtil.getApplicationContext().getBean(TaskPlanLogService.class);  SimpleDateFormat sdf = new SimpleDateFormat("yyyy-MM-dd hh:mm:ss");  @Override  //把要执行的操作，写在execute方法中  public void execute(JobExecutionContext context) throws JobExecutionException {  JobKey jobKey = context.getJobDetail().getKey();  JobDataMap dataMap = context.getJobDetail().getJobDataMap();  String eMsg ="";  String execFlag ="F";  String progType = dataMap.getString("progType");  String taskObject = dataMap.getString("taskObject");  //TODO 先注释掉，后面实际运行时开起来  TaskPlanLog taskPlanLog = insertTaskLog(dataMap);  String taskPlanLogId = taskPlanLog.getTaskPlanLogId();  //执行JOB  JobParameters jobParameters ;  org.springframework.batch.core.Job batchJob; //存储过程的话带存储过程名字作为参数  if(SchedulerConstant.PROG\_TYPE\_PTP.equals(progType)){  String inputParam = dataMap.getString("inputParam");  batchJob = SpringContextUtil.getApplicationContext().getBean("procedureJob", org.springframework.batch.core.Job.class);  jobParameters = new JobParametersBuilder()  .addString("taskPlanLogId",taskPlanLogId)  .addString("procedure",taskObject)  .addString("inParams",inputParam)  .toJobParameters();  }else{  batchJob = SpringContextUtil.getApplicationContext().getBean(taskObject, org.springframework.batch.core.Job.class);  jobParameters = new JobParametersBuilder()  .addString("taskPlanLogId",taskPlanLogId)  .addString("taskJobName",taskObject)  .toJobParameters();  }  SimpleJobLauncher jobLauncher = SpringContextUtil.getApplicationContext().getBean(SimpleJobLauncher.class);  try {  jobLauncher.run(batchJob,jobParameters);  // System.out.println("执行完成");  // eMsg = "执行完成";  // execFlag =SchedulerConstant.PLAN\_LOG\_STATE\_SUC;  } catch (Exception e) {  eMsg = e.getMessage();  execFlag =SchedulerConstant.PLAN\_LOG\_STATE\_FAL;  }finally {  //TODO 先注释掉，后面实际运行时开起来  /\*  taskPlanLog.setExecMsg(eMsg);  taskPlanLog.setExecState(execFlag);  updateTaskLog(taskPlanLog);  \*/  }  }  public TaskPlanLog insertTaskLog(JobDataMap dataMap) {  String taskObject= dataMap.getString("taskObject");  String taskName = dataMap.getString("taskName");  //记录日志  System.out.println(sdf.format(DateTimeUtil.nowDate())+"任务计划 : " + taskName + " 程序名 : " + taskObject + "开始执行");  TaskPlanLog taskPlanLog = new TaskPlanLog();  taskPlanLog.setTaskPlanLogId(UUID.randomUUID().toString());  taskPlanLog.setTaskName(dataMap.getString("taskName"));  taskPlanLog.setTaskPlanName(dataMap.getString("taskPlanName"));  taskPlanLog.setTaskId(dataMap.getString("taskId"));  taskPlanLog.setStartDatetime(DateTimeUtil.nowDate());  taskPlanLog.setInputParam(dataMap.getString("inputParam"));  taskPlanLog.setTaskObject(taskObject);  taskPlanLog.setTaskPlanId(dataMap.getString("taskPlanId"));  taskPlanLog.setExecMsg("开始执行");  taskPlanLog.setExecState(SchedulerConstant.PLAN\_LOG\_STATE\_STR);  Reflections.invokeMethod(taskPlanLogService,"createLog",new Class[]{TaskPlanLog.class},new Object[]{taskPlanLog});  return taskPlanLog;  }  public TaskPlanLog updateTaskLog(TaskPlanLog taskPlanLog) {  Date now = DateTimeUtil.nowDate();  taskPlanLog.setEndDatetime(now);  System.out.println(sdf.format(now)+"任务计划 : " + taskPlanLog.getTaskPlanName() +  " 程序名 : " + taskPlanLog.getTaskObject() + taskPlanLog.getExecMsg());  Reflections.invokeMethod(taskPlanLogService,"updateLog",new Class[]{TaskPlanLog.class},new Object[]{taskPlanLog});  return taskPlanLog;  }  } |

主要程序：

1. 生成JOB

batchJob = SpringContextUtil.getApplicationContext().getBean(taskObject, org.springframework.batch.core.Job.class);

jobParameters = new JobParametersBuilder()

.addString("taskPlanLogId",taskPlanLogId)

.addString("taskJobName",taskObject)

.toJobParameters();

1. 得到任务发射器

SimpleJobLauncher jobLauncher = SpringContextUtil.getApplicationContext().getBean(SimpleJobLauncher.class);

1. 启动发射器

jobLauncher.run(batchJob,jobParameters);