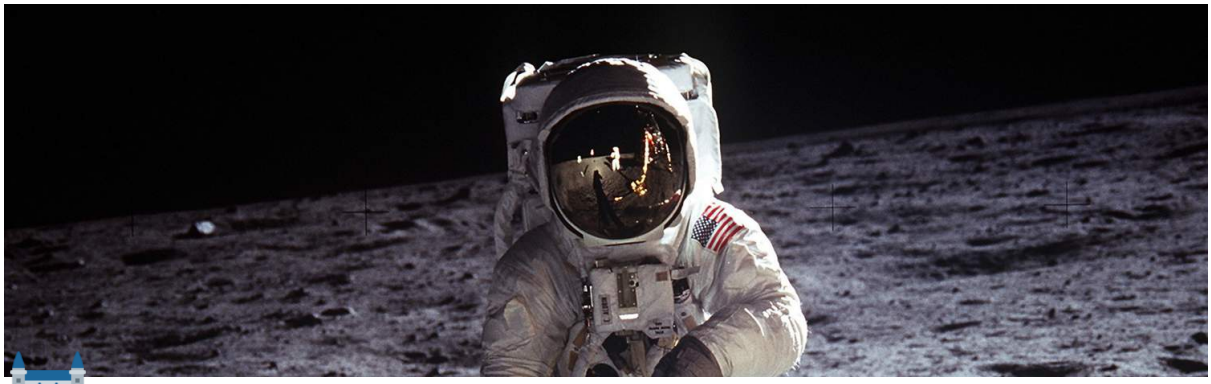


调度服务

1、使用场景

适用于需要执行定时任务的场景下使用，当某个时刻需要执行一些任务时即可启动一个调度任务，并设定对应时间点的触发器，当满足触发条件时就会开始执行指定任务。

2、相关概念

- 调度三要素：一、执行器；二、任务逻辑；三、触发器。整个调度服务都是围绕着这三要素开展的。
- 执行器：首先需要明确的是执行器就是一个物理主机，在 hzero 中表现为一个具有 /v1/scheduler/executor 接口的服务，用于接受服务端发送过来的 RPC 请求。
- 任务逻辑：任务逻辑是执行具体任务的对象，表现为一个实现了 IJobHandler 接口的实体类，也是我们在定义中的 JobHandler 参数。
- 调度任务：首先明确在调度任务中，“调度”是一个动词即一个动作，“任务”是一个名词需要我们定义。因此在调度任务中我们首先需要的是定义一个任务用于关联执行器和执行逻辑以及触发器，还有一些额外的参数设定，当一个“任务”被定义好之后我们才可以对该任务执行“调度”动作，在“调度”动作中具备执行、暂停、终止操作（仅针对任务而言）因此我们可以对这个任务进行执行操作，暂停这个任务执行以及终止这个任务的执行。
- 可执行定义：可执行即是具备执行能力的相关定义，在 hzero 中具备执行能力的首要两个硬性条件（客户端是一个微服务），第一、有一个服务器作为客户端执行具体的任务逻辑；第二、定义一个具体的任务逻辑，因此在可执行定义中就是为了关联执行器和任务逻辑的。需要着重提醒的是任务逻辑必须定义于客户端服务器中而不是服务端。
- 请求定义：当可执行定义完成后我们就具备了执行能力的基本条件，接下在就是请求执行器动作的环节，这里需要注意请求定义并不是真正意义上的发起一个网络请求（尽管最后是通过 RPC 请求让客户端开始执行任务逻辑的）而是为任务逻辑的执行提供需要的参数，任务逻辑（JobHandler）在执行是很多时候会用到一些相关参数，而这些参数的定义就是在请求定义中定义的。因此简单来说请求定义就是对可执行器在执行任务时需要用到的参数进行定义。注意：这里只是定义参数格式。
- 并发请求：并发请求在某种意义上和调度任务是相似的，但是“调度任务”的职能比“并发请求”的更重。“调度任务”中需要关联执行器、执行逻辑和触发器从而构成一个完整的调度过程；而在“并发请求”中执行器和执行逻辑的关联操作在可执行定义中已经被完成了，因此从职能上来看“并发请求”的更轻量一些。除此之外的不同之初还有一点就是由于“请求定义”的作用使得“任务参数”的输入相比于“调度任务”来说“并发请求”更加友好。
- 调度日志：调度日志是对每次执行调度任务结果的记录，其中有两个结果一个是调度结果另一个是客户端执行结果，“调度结果”是 Quatrz 中 Job 的执行结果，当并不是具体的任务逻辑的执行结果；“客户端执行结果”则是具体任务逻辑的执行结果，又由于具体任务逻辑是客户端执行的因此叫做客户端执行结果。

注意：在任务调度的架构设计中具有两条路线实现任务的调度。第一、执行器 ——> 调度任务 ； 第二、执行器管理 ——>可执行定义 —— > 请求定义 —— > 并发请求。因此实现任务的调度我们有两个路线可以选择。

3、实战

3.1、执行器 ——> 调度任务

- 新建执行器：调度服务 ——> 执行器管理 ——> 执行器管理

头行结构行信息中展示的是对已定义执行器的展示，头信息是对执行器的查询定义。

- 配置执行器

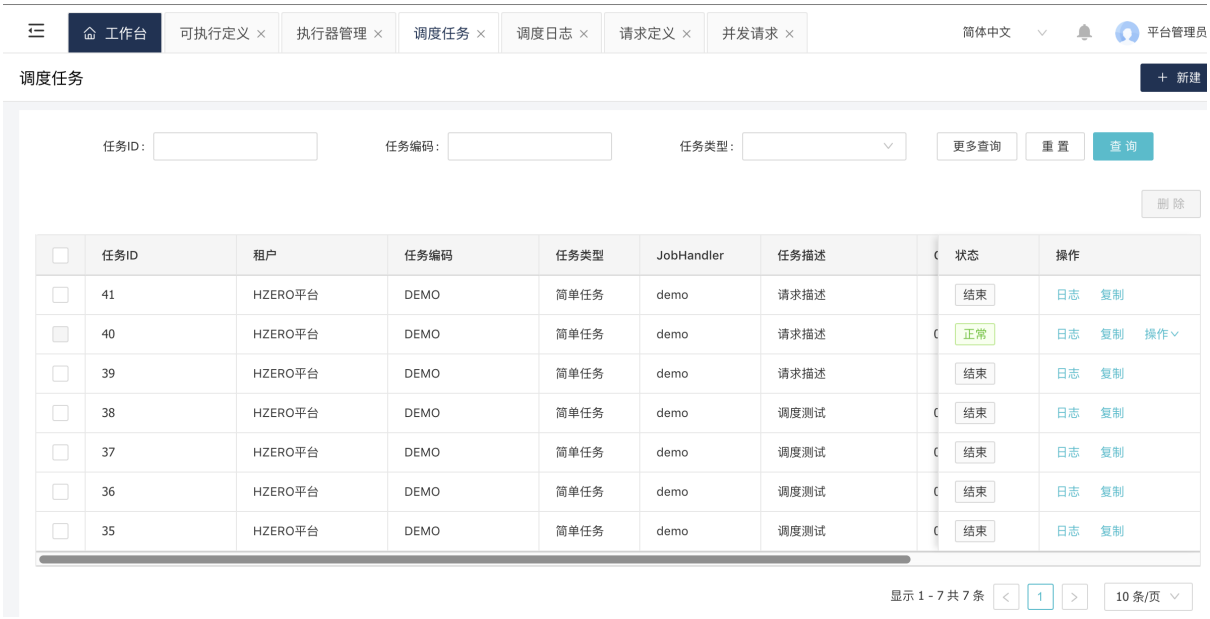
对执行器定义中的执行器组进行配置：权重用于执行策略为执行器权重时使用，权重越大得到执行任务逻辑的机会越多；最大并发量当前执行器可以接受的最大并发量，当最大并发量为 0 时则目标执行器不会得到执行，即不可用。

- 排序：排序字段，官方描述表格展示按照从小到大排列（貌似然并*）
- 执行器编码：是执行器的唯一标示，在任务调度中需要通过 LOV 指定执行器，因此具有唯一性
- 执行器名称：对执行器的简单描述，可辨识即可
- 机器地址：这个是比较重要的参数，前面说过执行器就是一个加入注册中心的物理主机，因此这里就是对该服务主机的定义，物理主机的IP地址加上物理主机上跑服务的端口好
- 状态：具有在线和手动下线两种状态。

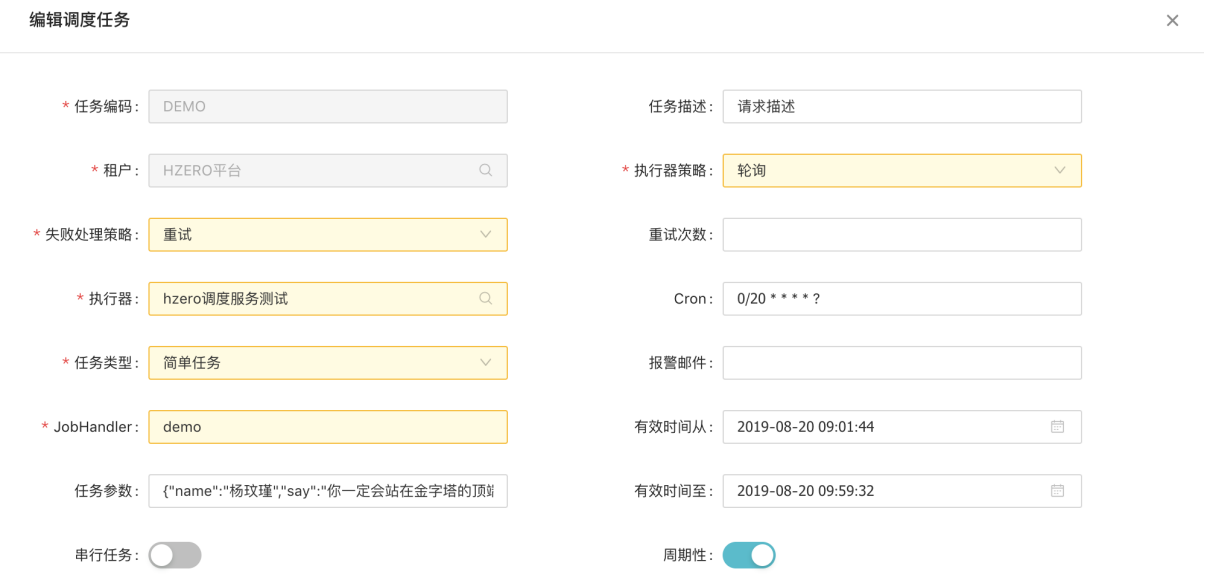
- 定义具体的任务逻辑

```
@JobHandler("demo")
public class DemoJobHandler implements IJobHandler {
    @Override
    public ReturnT execute(Map<String, String> map, SchedulerTool tool) {
        System.out.println(map.get("name")+"说"+map.get("say"));
        return ReturnT.SUCCESS;
    }

    @Override
    public void onCreate(SchedulerTool tool) {
        System.out.println("DemoJobHandler#onCreate()");
    }

    @Override
    public void onException(SchedulerTool tool) {
        System.out.println("DemoJobHandler#onException()");
    }

    @Override
    public void onFinish(SchedulerTool tool, ReturnT returnT) {
        System.out.println("DemoJobHandler#onFinish()");
    }
}
```

- 调度任务：调度服务 ——> 调度任务 ——> 调度任务



同样是简单的头行结构用于对调度任务的展示和查询。



- 任务编码：任务的编码标识，没有唯一性约束。
- 任务描述：对任务的描述信息，可辨识即可。
- 租户：当前任务的所属租户。
- 执行器策略：

  轮询：循环遍历执行器中定义的所有客户端来执行任务

  执行器权重：根据执行器定义中对执行器配置的权重值来将任务按比例的分到对应的执行器上。

  任务-执行器权重：和执行器权重不同的是任务-执行器权重中的权重值不是在执行器管理中配置的而是在选择了任务-执行器策略后会在下方重新为执行器组指定权重。

- 失败处理策略：

  忽略：无视调度结果，调度失败不进行任何操作，仅记录日志。

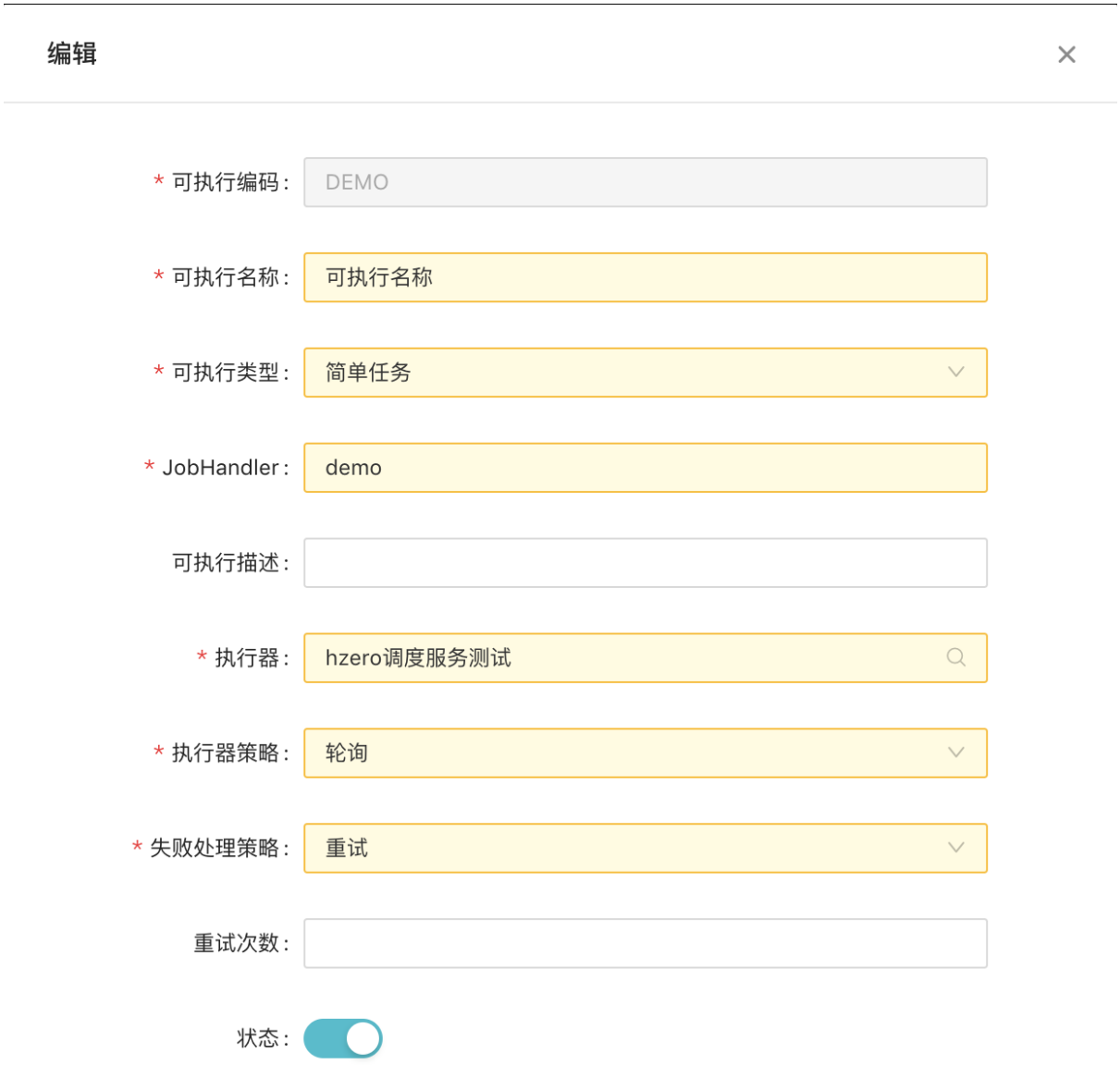
  转移：当选择的执行器调度失败，自动根据执行器策略选择下一个执行器执行任务。

  重试：当选择的执行器调度失败，5秒后再次尝试连接该执行器，默认重试次数为3次。
- 执行器：LOV 选择执行器，比较重要三要素之一。
- Cron:Cron表达式是触发器的体现，比价重要是三要素之一。
- 任务类型：目前只支持简单任务，运行模式为简单任务时，JobHandler为必输。
- 报警邮件：当调度任务出现异常时会通过邮件的形式通知，需要结合 Message 服务。
- JobHandler：具体的任务逻辑与后端编写的 @JobHandler("demo") 相对应，比较重要三要素之一。
- 有效时间从：任务的开启时间。
- 任务参数：以 JSON 的形式向任务逻辑中传递参数。
- 有效期至：任务的结束时间。
- 串行任务：是否允许该任务并发执行。
- 周期性：标识任务为周期任务还是瞬时任务。

- 执行任务

3.2、执行器管理 ——>可执行定义 —— > 请求定义 —— > 并发请求

- 可执行定义



- 可执行编码：可执行定义的唯一标示。
- 可执行名称：对可执行定义的简单描述，可辨识即可。
- 可执行类型：仅支持简单任务。
- JobHandler：具体任务逻辑的定义和后端的 @JobHandler("demo") 对应，比较重要三要素之一。
- 可执行描述：对可执行的描述信息。
- 执行器：LOV 选择执行器管理中定义的执行器。
- 执行器策略：

  轮询：循环遍历执行器中定义的所有客户端来执行任务

  执行器权重：根据执行器定义中对执行器配置的权重值来将任务按比例的分到对应的执行器上。

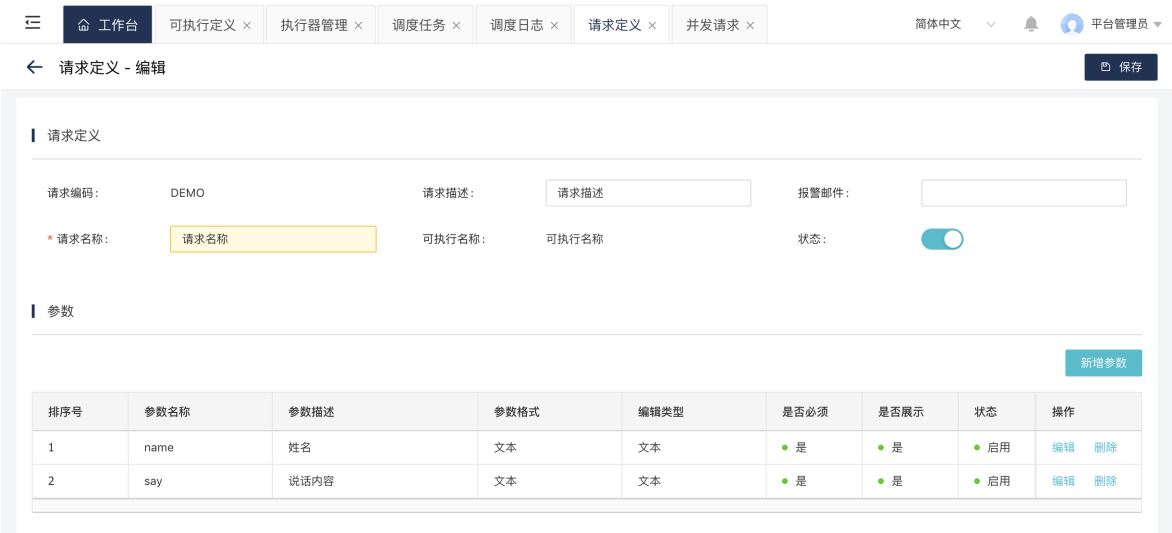
  任务-执行器权重：和执行器权重不同的是任务-执行器权重中的权重值不是在执行器管理中配置的而是在选择了任务-执行器策略后会在下方重新为执行器组指定权重。
- 失败处理策略：

  忽略：无视调度结果，调度失败不进行任何操作，仅记录日志。

  转移：当选择的执行器调度失败，自动根据执行器策略选择下一个执行器执行任务。

重试：当选择的执行器调度失败，5秒后再次尝试连接该执行器，默认重试次数为3次。

- 请求定义



- 请求编码：请求定义中的唯一标示，和调度任务总的任务编码一样。
- 请求描述：对请求的简单描述，可辨识即可。
- 报警邮件：当调度任务出现异常时会通过邮件的形式通知，需要结合 Message 服务。
- 请求名称：对请求的简单命名，可辨识即可。
- 可执行名称：指定执行本次请求的执行器，和可执行定义中相呼应。
- 参数：定义任务逻辑执行中需要用到的参数。

  参数名称：相当于 JSON 中的key

  参数描述：对参数的简单描述，可辨识即可。

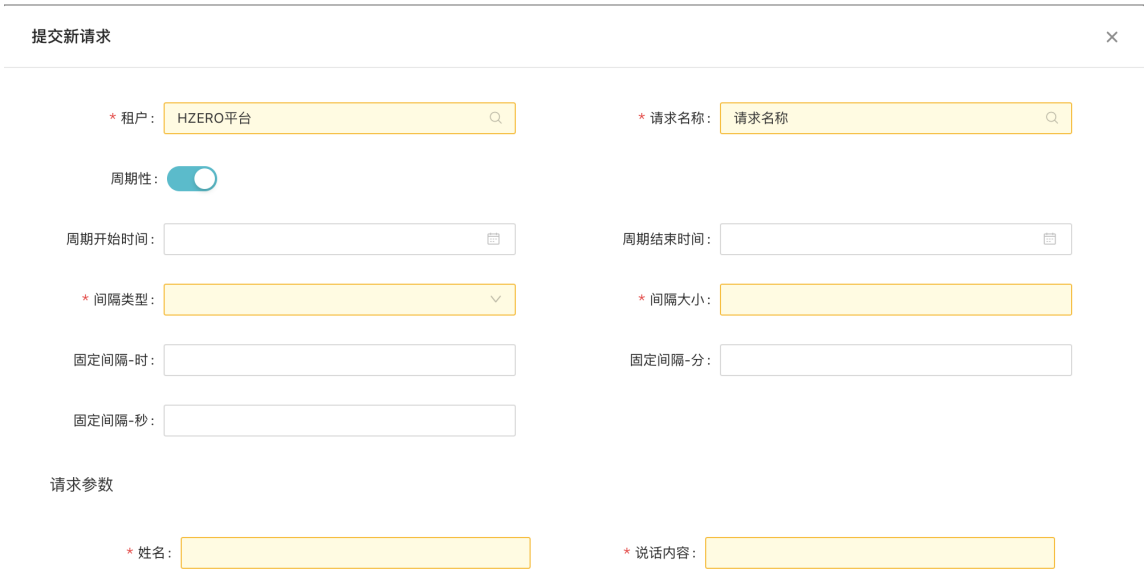
  参数格式：支持数字、日期、文本

  编辑类型：参数输入的组件类型。支持 LOV、文本、勾选框、下拉框、日期选择框、数字框类型。

  是否必须：参数在任务逻辑中是否为必须

  是否展示：如果选择否，则在下个环节（并发请求）中就不会显示该参数的 UI 组件

- 并发请求



- 租户：发起请求的组合
- 请求名称：LOV 选择上个环节中的请求定义
- 周期性：到目前止发现在 执行器管理 ——>可执行定义 —— > 请求定义 —— > 并发请求 这个路线中一直没有出现触发器的定义，其实这里就是对触发器和请求定义的绑定，而请求定义中具备了执行器和任务逻辑的绑定，因此到目前位置执行器、任务逻辑和触发器都确定了，而触发器这里并没有使用 Cron 表达式，而是通过周期性指定来作为触发器，只有当周期性被开启后周期性组件和请求参数之间的组件才会被显示用于定义一个具有周期性的触发器。后端会根据所选内容自动生成 Corn 表达式。

  周期开启时间：调度任务开启时间

  周期结束时间：调度任务结束时间

  间隔类型：间隔类型包括 天、时、分、秒

  间隔大小：间隔的数量级和间隔类型搭配使用

- 请求参数：这里的请求参数是在请求定义环节中定义好的，包括请求参数的 UI 组件和参数名称。

- 执行调度任务 —— 执行、暂停、终止

4、源码分析

- JobInfoController#createJob()

```
@ApiOperation("任务创建")
    @Permission(
        level = ResourceLevel.ORGANIZATION
    )
    @PostMapping
    public ResponseEntity<JobInfo> createJob(@PathVariable Long organizationId, @RequestBody JobInfo jobInfo) {
        jobInfo.setTenantId(organizationId);
        this.validObject(jobInfo, new Class[0]);
        return Results.success(this.jobInfoService.createJob(jobInfo));
    }
```

当我们新建调度任务时将前端输入的信息封装成 JobInfo 用 POST 方法请求 /v1/{organizationId}/job-info 接口，从而进入该方法。在该方法中先对租户 ID 进行设定并对 jobInfo 对象进行校验以后，返回了 jobInfoService.createJob() 的执行结果。

- JobInfoServiceImpl#createJob()

```
@Transactional(
        rollbackFor = {Exception.class}
    )
    public JobInfo createJob(JobInfo jobInfo) {
        if (Objects.equals(jobInfo.getCycleFlag(), Flag.NO)) {
            jobInfo.setJobCron((String)null);
        }

        jobInfo.validate();
        ValidUtil.isJSONValid(jobInfo.getJobParam());
        this.jobInfoRepository.insertSelective(jobInfo);
        this.jobService.addJob(jobInfo);
        return jobInfo;
    }
```

首先对 jobInfo 的周期性进行判断，如果是非周期性的则 Corn 表达式为 null，然后对 jobInfo 进行校验（jobInfo.validate();），以及对 jobParam 参数进行判断确保是 JSON 格式的（ValidUtil.isJSONValid(jobInfo.getJobParam())），当校验通过后对 jobInfo 进行数据库持久化操作（jobInfoRepository.insertSelective(jobInfo)）。随后调用 jobService 添加 job (this.jobService.addJob(jobInfo))，这里可能会有点蒙，想不通为什么会有两个 JobService，其中 JobInfoServiceImpl 是我们通常说的 Service 层用于执行持久化的操作逻辑的；而 JobServiceImpl 则是执行 Quartz 相关操作的服务，例如添加一个 job 、构建一个触发器等工作。

- JobServiceImpl#addJob()

```
public void addJob(JobInfo jobInfo) {
        Executor executor = (Executor)this.executorRepository.selectByPrimaryKey(jobInfo.getExecutorId());
        Assert.notNull(executor, "hsdr.error.executor_not_find");
        List<String> address = StringUtils.isNotBlank(executor.getAddressList()) ? Arrays.asList(executor.getAddressList().s
        ((List)address).forEach((item) -> {
            ExecutorConfig.addCache(this.redisHelper, this.configRepository, jobInfo.getExecutorId(), item, jobInfo.getJobId
        });

        try {
            JobDetail jobDetail = JobBuilder.newJob(MyJob.class).withDescription(jobInfo.getDescription()).withIdentity(Stri
            Trigger trigger = this.buildTrigger(jobInfo);
            Assert.notNull(trigger, "hsdr.error.create_job");
            this.scheduler.scheduleJob(jobDetail, trigger);
            this.scheduler.start();
            logger.debug("-----------------  add job success, jobId : {}  tenantId: {}", jobInfo.getJobId(), jobInfo.getTena
        } catch (Exception var6) {
            throw new CommonException("hsdr.error.quartz.add", new Object[0]);
        }
    }
```

首先从数据库中查询是否有 jobInfo 中指定的执行器，然后将执行器组中的地址依次加入到缓存中，值得注意的是在加入缓存的过程中首先需要对地址的最大并发量进行判断。

- ExecutorConfig.addCache()

```
private static String getCacheKey(Long executorId) {
        return "hsdr:executor-config:" + executorId;
    }

    public static boolean addCache(RedisHelper redisHelper, ExecutorConfigRepository repository, Long executorId, String addres
        List<String> data = getCache(redisHelper, executorId, address);
        ExecutorConfig config = repository.selectByUnique(executorId, address);
        if (config == null) {
            config = new ExecutorConfig();
        }

        Integer maxConcurrent = config.getMaxConcurrent();
        if (maxConcurrent != null && maxConcurrent <= data.size()) {
            return false;
        } else {
            if (!data.contains(String.valueOf(jobId))) {
                data.add(String.valueOf(jobId));
                clearCache(redisHelper, executorId, address);
                redisHelper.hshPut(getCacheKey(executorId), address, redisHelper.toJson(data));
            }
```

```
            return true;
        }
    }
public static List<String> getCache(RedisHelper redisHelper, Long executorId, String address) {
        String data = redisHelper.hshGet(getCacheKey(executorId), address);
        return (List)(StringUtils.isNotBlank(data) ? (List)redisHelper.fromJson(data, List.class) : new ArrayList());
    }
```

再加入缓存的逻辑中首先会使用 "hsdr:executor-config:"+“执行器ID”作为 key向 Redis 缓存中查询 “address”字段的值，这里的 address 变量其实就是执行器中定义的IP地址，如果执行器配置中的最大并发量（max_concurrent 字段）没有设定或者小于了当前的并发数量，则返回 false 表示加入缓存失败。这里需要注意两个地方，第一、执行器配置和执行器不是同一个东西在数据库中表现为两张表，执行器（hsdr_executor表）记录的是我们在执行器管理界面中定义的信息；而执行器配置（hsdr_executor_config表）是专门针对执行器管理中的 address_list 字段设立的一张表用于记录 IP地址信息。第二、这里有个最大并发量的设定，同一个IP我们可以在多个执行器中定义最大并发量那么到底以那个为准尼？其实这里的最大并发量只针对与当前的执行器，是这个意思也不是这个意思，首先从源码中可以看到每次对IP进行缓存以后都会添加一条记录中到缓存中表现为一个并发量，那么加入现在有 A、B两个调度任务指定的执行器组（不是同一个执行器）中都有同一个IP 客户端，那么A调度任务先在缓存中占用一个并发量，当B调度任务为该 IP 执行 addJob() 时这个IP的并发量已经就是 1 了，如果 调度任务 B 指定的执行器中的最大并发量设定为 1 的话，则 B 调度任务就会被拒绝，应该该 IP 的客户端并发量已满。也就是说执行器中定义的最大并发量指的是如果当前系统中该 IP 的并发量不超过设定值的话就可以使用该 IP 执行调度任务，这里要理解好当前系统中该 IP 的并发量，一个系统中可能不知一个调度任务，一个客户端也不可能只执行一个调度任务，当其他调度任务使用了该客户端时就相当于为该客户端增加了一个并发量。

再回到 JobServiceImpl#addJob() 中在判断完并发量后通过 this.scheduler.scheduleJob(jobDetail, trigger) 对触发器和 Job 进行绑定（详细：见 Quartz 官方文档）并开启调度（scheduler.start()）。这里的触发器是通过 buildTrigger(jobInfo) 方法创建的

- JobServiceImpl#buildTrigger()

```
private Trigger buildTrigger(JobInfo jobInfo) {
        TriggerKey triggerKey = TriggerKey.triggerKey(String.valueOf(jobInfo.getJobId()), String.valueOf(jobInfo.getTenantId()))
        Trigger trigger = null;
        if (Objects.equals(jobInfo.getCycleFlag(), Flag.YES)) {
            String cron = jobInfo.getJobCron();
            if (StringUtils.isBlank(cron)) {
                cron = "0 0 0 1 * ? 2100";
            }

            CronScheduleBuilder cronScheduleBuilder = CronScheduleBuilder.cronSchedule(cron).withMisfireHandlingInstructionDoNo
            if (jobInfo.getStartDate() != null && jobInfo.getEndDate() != null) {
                trigger = TriggerBuilder.newTrigger().withIdentity(triggerKey).withSchedule(cronScheduleBuilder).startAt(jobInf
            }

            if (jobInfo.getStartDate() != null && jobInfo.getEndDate() == null) {
                trigger = TriggerBuilder.newTrigger().withIdentity(triggerKey).withSchedule(cronScheduleBuilder).startAt(jobInf
            }

            long currentTime;
            if (jobInfo.getStartDate() == null && jobInfo.getEndDate() != null) {
                currentTime = System.currentTimeMillis() + 30000L;
                trigger = TriggerBuilder.newTrigger().withIdentity(triggerKey).withSchedule(cronScheduleBuilder).startAt(new Da
            }

            if (jobInfo.getStartDate() == null && jobInfo.getEndDate() == null) {
                currentTime = System.currentTimeMillis() + 30000L;
                trigger = TriggerBuilder.newTrigger().withIdentity(triggerKey).withSchedule(cronScheduleBuilder).startAt(new Da
            }
        } else if (jobInfo.getStartDate() == null) {
            trigger = TriggerBuilder.newTrigger().withIdentity(triggerKey).withSchedule(SimpleScheduleBuilder.simpleSchedule().
        } else {
            trigger = TriggerBuilder.newTrigger().withIdentity(triggerKey).withSchedule(SimpleScheduleBuilder.simpleSchedule().
        }

        return trigger;
    }
```

通过判断周期性标示，如果是周期性的则获取 Corn 表达式接下来对 Corn 表达式作判空处理，如果为空则给一个默认的表达式 （cron = "0 0 0 1 * ? 2100"）接下来通过 Corn 表达式创建了一个 CronTrigger 类型的触发器（详情见：Quartz 官方文档 https://www.w3cschool.cn/quartz_doc/quartz_doc-lwuv2d2a.html）然后更具开始时间和截止时间对触发器进行配置，这里根据开启时间和截止时间组成了四种情况进行了对应的设置。如果是非周期性的则创建一个 SimpleTrigger 类型的触发器并设定开启时间，最后将触发器返回。

再回到 JobServiceImpl#addJob() 中，触发器的创建流程说完了，到了该说 JobDetail的时候了（JobBuilder.newJob(MyJob.class).withDescription(jobInfo.getDescription()).withIdentity(String.valueOf(jobInfo.getJobId()),

String.valueOf(jobInfo.getTenantId())).usingJobData(this.getMap(jobInfo)).build()） 可以说 JobDetail 的创建才是重头戏，我们仔细看这条创建语句，其中 newJob() 的参数是 MyJob.class 并不是我们实现了 IJobHandler 接口的那个东西，当然这也说得通 newJob() 方法的参数必须是 Job 类型的而 Job 是 Quartz 官方定义的接口，这样也就说得通了我们之前说的 @JobHandler("demo") 是任务逻辑但却不是 Job 。也就是说我们的触发器绑定的并不是 @JobHandler("demo") 而是 MyJob 这个类，只是每次触发时传入的参数都不一样而已，因此所有的秘密就都在 MyJob 这个类中了。

- MyJob#execute()

```
public void execute(JobExecutionContext context) {
        String triggerName = context.getTrigger().getKey().getName();
        Long jobId = Long.valueOf(context.getJobDetail().getKey().getName());
        logger.info(" Scheduler Job Start.  JobId : {} ", jobId);
        JobDataMap jobDataMap = context.getJobDetail().getJobDataMap();
        JobLog jobLog = new JobLog();
        Long executorId = jobDataMap.getLongValue("executorId");
        Integer serial = StringUtils.isBlank(String.valueOf(jobDataMap.get("serial"))) ? Flag.YES : Integer.valueOf(String.valu
        if (Objects.equals(serial, Flag.YES) && !JobLock.addLock(jobId)) {
            logger.info(" Scheduler Job Locked.  JobId : {} ", jobId);
        } else {
            try {
                String jobCode = jobDataMap.getString("jobCode");
                Long tenantId = jobDataMap.getLongValue("tenantId");
                String jobType = jobDataMap.getString("jobType");
                String jobHandler = jobDataMap.getString("jobHandler");
                String paramStr = jobDataMap.getString("param");
                this.jwtToken = jobDataMap.getString("jwtToken");
                this.failStrategy = jobDataMap.getString("failStrategy");
                this.executorStrategy = jobDataMap.getString("executorStrategy");
                if (jobDataMap.containsKey("retryNumber")) {
                    this.retryNumber = Integer.parseInt(String.valueOf(jobDataMap.get("retryNumber")));
                }

                List<String> errorList = new ArrayList();
                //重点一、根据执行器策略选择出执行器IP
                String address = this.urlService.getUrl(this.executorStrategy, executorId, jobId, errorList);
                if (StringUtils.isNotBlank(address)) {
                    //构建 JobLog 对象准备持久化调度日志
                    jobLog.setJobId(jobId).setTenantId(tenantId).setExecutorId(executorId).setStartTime(new Date()).setJobResul
                    this.jobLogRepository.insertSelective(jobLog);
                    JobDataDTO jobDataDTO = (new JobDataDTO()).setJobId(jobId).setLogId(jobLog.getLogId()).setJobCode(jobCode).
                    //重点二、发送请求通知客户端执行任务
                    this.send(address, jobLog, jobDataDTO, errorList);
                } else {
                    jobLog.setJobId(jobId).setTenantId(tenantId).setExecutorId(executorId).setStartTime(new Date()).setJobResul
                    this.jobLogRepository.insertSelective(jobLog);
                }
            } catch (Exception var16) {
                logger.error(ExceptionUtils.getMessage(var16));
                if (jobLog.getLogId() != null) {
                    //如果执行过程中出现异常则修改数据库中的 job_result 字段为 FAILURE
                    this.jobLogRepository.updateByPrimaryKey(jobLog.setJobResult("FAILURE").setClientResult((String)null).setMe
                    this.jobService.sendEmail(jobLog.getLogId());
                }
            }

            if (context.getNextFireTime() == null && Objects.equals(String.valueOf(jobId), triggerName)) {
                this.clearJobInfo(executorId, jobId);
            }

        }
    }
```

execute() 方法是 Job 执行的入口，因此当触发器触发时执行的都是execute() 方法。在该方法中首先是获取 jobDataMap 对象然后从该对象中依次获取相关参数。在这个方法中有两个地方值得我们关注，第一、address = this.urlService.getUrl(this.executorStrategy, executorId, jobId, errorList) 这里是根据执行器策略选择出合适的IP地址作为客户端因此执行策略的相关逻辑都在这里了；第二、this.send(address, jobLog, jobDataDTO, errorList) 我们之前分析了 MyJob 并不是真正的执行调度任务逻辑的地方，因此 MyJob 是如何通知客户端执行调度逻辑的就隐藏于此。

- UrlServiceImpl#getUrl()

```
public String getUrl(String executorStrategy, Long executorId, Long jobId, List<String> errorUrls) {
        byte var6 = -1;
        switch(executorStrategy.hashCode()) {
        case -1738262920:
            if (executorStrategy.equals("WEIGHT")) {
                var6 = 1;
            }
            break;
```

```
        case 320077731:
            if (executorStrategy.equals("POLLING")) {
                var6 = 0;
            }
            break;
        case 1614461274:
            if (executorStrategy.equals("JOB_WEIGHT")) {
                var6 = 2;
            }
        }

        switch(var6) {
        case 0:
            return this.polling(executorId, jobId, errorUrls);
        case 1:
            return this.weight(executorId, jobId, errorUrls);
        case 2:
            return this.jobWeight(executorId, jobId, errorUrls);
        default:
            return "";
        }
    }
```

首先判断执行器策略类型，如果是“权重”策略（WEIGHT）执行 weigth() 方法“轮询”策略（POLLINTG）执行 polling() 方法“任务-权重”策略执行 jobWeight() 方法。

- UrlServiceImpl#weight()

```
private String weight(Long executorId, Long jobId, List<String> errorUrls) {
        Executor executor = (Executor)this.executorRepository.selectByPrimaryKey(executorId);
        List<String> addressList = this.usableUrl(executor, errorUrls);
        if (CollectionUtils.isEmpty(addressList)) {
            return "";
        } else {
            List<ExecutorConfig> configList = new ArrayList();
            //定义权重总和
            Integer weightSum = 0;
            Iterator var8 = addressList.iterator();
            while(var8.hasNext()) {
                String address = (String)var8.next();
                //根据执行器ID和IP地址获取当前执行器下该IP的配置信息
                ExecutorConfig config = this.configRepository.selectByUnique(executorId, address);
                if (config == null) {
                    config = (new ExecutorConfig()).setWeight(1).setAddress(address);
                }
                //统计权重总和后将该IP的配置信息加入到 config
                weightSum = weightSum + config.getWeight();
                configList.add(config);
            }
            //调用 calculateWeight() 计算出权重
            String address = this.calculateWeight(configList, weightSum);
            if (this.usable(address, jobId, executorId, errorUrls)) {
                return address;
            } else {
                return this.weight(executorId, jobId, errorUrls);
            }
        }
    }
private String calculateWeight(List<ExecutorConfig> list, Integer weightSum) {
        //在权重总和中获取一个随机整数
        Integer n = (new Random()).nextInt(weightSum);
        Integer m = 0;
        String address = "";

        ExecutorConfig config;
        /**
         * 在这个循环中 m = m + config.getWeight() 就像把执行器的权重当作线段拼接到一起组成一个长度为权重总和的线段，然后随机数落到
         * 那个线段，则那个线段代表的执行器获选，
         */
        for(Iterator var6 = list.iterator(); var6.hasNext(); m = m + config.getWeight()) {
            config = (ExecutorConfig)var6.next();
            //这里需要注意的是，此时的 m 并没有包含当前 config 的权重，因此 m <= n 将随机数限定在了当前 config 所代表线段的起始位置以后
            //n < m + config.getWeight() 将随机数限定在了当前 config 所代表线段的重点位置之前，因此两个条件同时满足即代表着当亲啊执行器代表的线段之间
            //当然这里有先后顺序之分也并不是完全意义上的权重
            if (m <= n && n < m + config.getWeight()) {
                address = config.getAddress();
                break;
            }
        }

        if (StringUtils.isBlank(address)) {
            address = ((ExecutorConfig)list.get(0)).getAddress();
        }
```

```
        return address;
    }
```

- UrlServiceImpl#polling()

```
private String polling(Long executorId, Long jobId, List<String> errorUrls) {
        Executor executor = (Executor)this.executorRepository.selectByPrimaryKey(executorId);
        //根据执行器对象获取对应的IP地址列表
        List<String> urlList = this.usableUrl(executor, errorUrls);
        if (CollectionUtils.isEmpty(urlList)) {
            return "";
        } else {
            try {
                //以"hsdr:job-polling:" + executorId + ":" + jobId 为 key 从 Redis 缓存中获取当前执行器组轮询的索引
                Integer index = this.getCache(executorId, jobId);
                //根据索引值寻找下一个轮询索引
                index = (index + 1) % urlList.size();
                //将新的索引值重新刷新会缓存中
                this.refreshCache(executorId, jobId, index);
                //根据索引值获取对应位置上的 URL
                String address = (String)urlList.get(index);
                return this.usable(address, jobId, executorId, errorUrls) ? address : this.polling(executorId, jobId, errorUrls
            } catch (Exception var8) {
                logger.error(var8.toString());
                return "";
            }
        }
    }
private Integer getCache(Long executorId, Long jobId) {
        String key = "hsdr:job-polling:" + executorId + ":" + jobId;
        String result = this.redisHelper.strGet(key);
        return result == null ? -1 : Integer.valueOf(result);
    }
```

- UrlServiceImpl#jobWeight()

```
private String jobWeight(Long executorId, Long jobId, List<String> errorUrls) {
        Executor executor = (Executor)this.executorRepository.selectByPrimaryKey(executorId);
        List<String> addressList = this.usableUrl(executor, errorUrls);
        JobInfo jobInfo = (JobInfo)this.jobInfoRepository.selectByPrimaryKey(jobId);
        if (!CollectionUtils.isEmpty(addressList) && jobInfo != null) {
            try {
                Map<String, Object> map = (Map)this.objectMapper.readValue(jobInfo.getStrategyParam(), new TypeReference<Map<St
                });
                if (map.containsKey("jobWeight")) {
                    Map jobWeight = (Map)map.get("jobWeight");
                    List<ExecutorConfig> configList = new ArrayList();
                    Integer weightSum = 0;
                    Iterator var11 = addressList.iterator();

                    while(var11.hasNext()) {
                        String item = (String)var11.next();
                        if (jobWeight.containsKey(item)) {
                            //根据IP获取对应的权重值
                            Integer weight = (Integer)jobWeight.get(item);
                            //构建 ExecutorConfig 对象并加入到 configList 列表中
                            configList.add((new ExecutorConfig()).setAddress(item).setWeight(weight));
                            weightSum = weightSum + weight;
                        } else {
                            ExecutorConfig config = this.configRepository.selectByUnique(executorId, item);
                            if (config == null) {
                                config = (new ExecutorConfig()).setWeight(1).setAddress(item);
                            }

                            configList.add(config);
                            weightSum = weightSum + config.getWeight();
                        }
                    }
                    //计算权重值和 weight() 一样了
                    String address = this.calculateWeight(configList, weightSum);
                    if (this.usable(address, jobId, executorId, errorUrls)) {
                        return address;
                    } else {
                        return this.jobWeight(executorId, jobId, errorUrls);
                    }
                } else {
                    return this.weight(executorId, jobId, errorUrls);
                }
            } catch (Exception var14) {
                logger.error(var14.toString());
```

```
                return "";
            }
        } else {
            return "";
        }
    }
```

执行器策略看完可以回到 MyJob#execute() 中的第二点了

- MyJob#send()

```
private void send(String address, JobLog jobLog, JobDataDTO jobDataDTO, List<String> errorList) {
        String data;
        try {
            data = this.objectMapper.writeValueAsString(jobDataDTO);
        } catch (Exception var7) {
            throw new CommonException("hsdr.error.parameter_error", new Object[0]);
        }

        if (StringUtils.isNotBlank(address)) {
            String url = "http://" + address + "/v1/scheduler/executor";
            RpcClient.asyncPost(url, data, this.jwtToken, new Callback() {
                public void onFailure(Call call, IOException e) {
                    MyJob.this.failureStrategy(address, jobLog, jobDataDTO, errorList, e);
                }

                public void onResponse(Call call, Response response) throws IOException {
                    response.close();
                }
            });
        } else {
            jobLog.setJobResult("FAILURE").setClientResult((String)null).setMessageHeader("No normal executor found.").setMessa
            this.jobLogRepository.updateByPrimaryKey(jobLog);
        }

    }
```

可以看到首先根据IP地址构建了一个 URL 值得注意的是这个 URL 的接口是/v1/scheduler/executor 然后通过 RPC 客户端（RpcClient）发起了一个异步的 POST 请求（asyncPost()）那么到这里就解释了我们之前说的客户端具备 /v1/scheduler/executor 接口才能实现调度任务。那么我们转过来查找实现 /v1/scheduler/executor 接口的 Controller 对象，通过端点打印我们找到了该接口的 Controller 类。

- JobExecuteController#runJob()

```
@RestController
public class JobExecuteController {
    @Autowired
    private JobExecuteService jobExecuteService;
    @Permission(
        permissionWithin = true
    )
    @PostMapping({"/v1/scheduler/executor"})
    public ResponseEntity<String> runJob(@RequestBody JobDataDTO jobDataDTO) {
        return Results.success(this.jobExecuteService.jobExecute(jobDataDTO));
    }
}
```

该方法中调用的是 jobExecuteService 对象的 jobExecute() 方法。

- JobExecuteServiceImpl#jobExecute()

```
@Service
public class JobExecuteServiceImpl implements JobExecuteService {
    public String jobExecute(JobDataDTO jobDataDTO) {
        String jobHandler = jobDataDTO.getJobHandler();
        Object handler = JobRegistry.getJobHandler(jobHandler);
        Long logId = jobDataDTO.getLogId();
        Long jobId = jobDataDTO.getJobId();
        JobLogDTO logDTO = (new JobLogDTO()).setLogId(logId).setJobId(jobId);
        if (handler == null) {
            this.jobLogBackService.updateLog(logDTO.setClientResult("FAILURE").setMessageHeader("No jobHandler").setMessage("Th
            return "SUCCESS";
        } else {
            IJobHandler iJobHandler = null;
            Thread thread = Thread.currentThread();
            SchedulerTool tool = new SchedulerTool(Scheduler.REDIS_DB, logId, jobDataDTO);
```

```
            String var12;
            try {
                if (handler instanceof IJobHandlerAllowStop) {
                    iJobHandler = (IJobHandlerAllowStop)handler;
                    ThreadRegistry.addJobHandler(thread.getId(), iJobHandler);
                    ThreadRegistry.addThread(thread, jobId);
                    this.jobRun(iJobHandler, jobDataDTO, logDTO, tool);
                } else {
                    IJobHandler iJobHandler = (IJobHandler)handler;
                    this.jobRun(iJobHandler, jobDataDTO, logDTO, tool);
                }

                return "SUCCESS";
            } catch (Exception var16) {
                logger.error(ExceptionUtils.getMessage(var16));
                iJobHandler.onException(tool);
                JobProgress progress = tool.getJobProgress();
                this.jobLogBackService.updateLog(logDTO.setClientResult("FAILURE").setMessageHeader(var16.getMessage()).setMess
                var12 = "SUCCESS";
            } finally {
                ThreadRegistry.deleteThread(thread);
                ThreadRegistry.deleteJobHandler(thread.getId());
            }

            return var12;
        }
    }
    //涵盖了 JobHandler 的生命周期以及 JobLog 的修改
    private void jobRun(IJobHandler iJobHandler, JobDataDTO jobDataDTO, JobLogDTO logDTO, SchedulerTool tool) throws IOExceptio
        String url = "";
        //指定 JobHandler 的 onCreate()方法
        iJobHandler.onCreate(tool);
        tool.updateProgress(0, "Job init.");
        Map<String, String> map = StringUtils.isNotBlank(jobDataDTO.getParam()) ? (Map)this.objectMapper.readValue(jobDataDTO.g
        }) : new HashMap(16);
        //任务逻辑的执行位置
        ReturnT result = iJobHandler.execute((Map)map, tool);
        tool.clearProgress(jobDataDTO.getLogId());
        if (tool.getByteArrayOutputStream().size() > 0) {
            String suffix = StringUtils.isNotBlank(tool.getFileSuffix()) ? tool.getFileSuffix() : ".txt";
            url = this.fileClient.uploadFile(jobDataDTO.getTenantId(), "hsdr", "log", jobDataDTO.getJobCode() + suffix, tool.ge
        }

        tool.closeLogger();
        iJobHandler.onFinish(tool, result);
        switch(result) {
        //根据执行结果修改 jobLog
        case SUCCESS:
            this.jobLogBackService.updateLog(logDTO.setClientResult("SUCCESS").setLogUrl(url));
            break;
        case FAILURE:
            this.jobLogBackService.updateLog(logDTO.setClientResult("WARNING").setLogUrl(url));
            break;
        default:
            this.jobLogBackService.updateLog(logDTO.setClientResult("FAILURE").setMessageHeader("empty.").setMessage("empty Res
        }

    }
}
```

在这段代码中首先从JobRegistry（Object handler = JobRegistry.getJobHandler(jobHandler)）中获取 JobHander，如果不存在指定的 JobHander 则更新 JobLog (jobLogBackService.updateLog(logDTO.setClientResult("FAILURE").setMessageHeader("No jobHandler").setMessage("The same jobHandler was not found.")))修改 JobLog 的失败信息从而便于在浏览器端查看，这也就是为什么我们的调度日志中的错误信息显示 “The same jobHandler was not found” 字样。然后对 handler 进行类型判断作一些设置后返回 “SUCCESS” 给服务端，从这我们可以看到 JobHandler 支持两种类型分别是 IJobHandlerAllowStop 和 IJobHandler 前者是后者的子类相比之下更加灵活，也就是说我们在写 JobHandler 时可以有选择的决定用哪一个接口来实现我们的任务逻辑。那么到这里还有一个问题需要我们去探索，我们知道 JobHandler 是通过 @JobHandler 注解标示的，而 JobRegistry 是如何找到的呢？（JobRegistry.getJobHandler(jobHandler)）

- JobRegistry

```
public class JobRegistry {
    private static Map<String, Object> jobMap = new ConcurrentHashMap();

    private JobRegistry() {
    }

    public static void addJobHandler(String jobCode, Object handler) {
        jobMap.put(jobCode, handler);
    }
```

```
    public static Object getJobHandler(String jobHandler) {
        return jobMap.get(jobHandler);
    }
}
```

JobRegistry 显得十分简单，其中仅仅维护了一个 Map 对象，当然我们可以确定的是其中的 key 就是我们 @JobHandler("demo") 注解中的属性值"name"，而 value 就是对应的 JobHandler 对象。因此我们更应该探索的是什么时候调用的 addJobHandler()方法。

- ExecutorInit#scanJobHandler()

```
@Component
public class ExecutorInit implements CommandLineRunner {
    private void scanJobHandler() {
        Map<String, Object> map = ApplicationContextHelper.getContext().getBeansWithAnnotation(JobHandler.class);
        Iterator var2 = map.values().iterator();

        while(var2.hasNext()) {
            Object service = var2.next();
            if (service instanceof IJobHandler) {
                JobHandler jobHandler = (JobHandler)service.getClass().getAnnotation(JobHandler.class);
                if (ObjectUtils.isEmpty(jobHandler)) {
                    logger.debug("could not get target bean , jobHandler : {}", service);
                } else {
                    JobRegistry.addJobHandler(jobHandler.value(), service);
                }
            }
        }

    }
}
```

通过 debug 的方式找到了调用 JobRegistry.addJobHandler() 的地方，在该方法中首先通过 ApplicationContext 对象从从其中获取 标注有 @JobHandler 注解的 Bean 对象，然后通过 Class 信息获取 @JobHandler 对象，最后通过 value 属性值作为 key 将对象依次加入到 JobRegistry 中的 Map 对象中。到这里整个流程就算基本走通了。最后剩下的失败策略在 MyJob 中也可以找到。

- MyJob#failureStrategy()

```
private void failureStrategy(String address, JobLog jobLog, JobDataDTO jobDataDTO, List<String> errorList, IOException e) {
        String var6 = this.failStrategy;
        byte var7 = -1;
        switch(var6.hashCode()) {
        case 77867656:
            if (var6.equals("RETRY")) {
                var7 = 0;
            }
            break;
        case 2063509483:
            if (var6.equals("TRANSFER")) {
                var7 = 1;
            }
        }

        switch(var7) {
        case 0:
            this.retry = this.retry + 1;
            if (this.retry < this.retryNumber) {
                try {
                    TimeUnit.SECONDS.sleep(5L);
                } catch (Exception var9) {
                    throw new CommonException(var9, new Object[0]);
                }

                this.send(address, jobLog, jobDataDTO, errorList);
            } else {
                jobLog.setMessageHeader(e.getMessage()).setMessage(ExceptionUtils.getMessage(e));
                this.failCallback(this.jobLogRepository, this.jobService, jobLog);
            }
            break;
        case 1:
            errorList.add(address);
            String newAddress = this.urlService.getUrl(this.executorStrategy, jobLog.getExecutorId(), jobLog.getJobId(), errorL
            jobLog.setAddress(newAddress);
            this.jobLogRepository.updateByPrimaryKey(jobLog);
            this.send(newAddress, jobLog, jobDataDTO, errorList);
            break;
        default:
            jobLog.setMessageHeader(e.getMessage()).setMessage(ExceptionUtils.getMessage(e));
```

```
            this.failCallback(this.jobLogRepository, this.jobService, jobLog);
        }

    }
```

源码分析到这里可以止步了，更详细的内容按照逻辑看源码。

5、总结

首先在查找 Maven 包下的源码时会找到两个包。第一、Maven:org.hzero.boot:hzero-boot-scheduler-0.11.0.RELEASE.jar 第二、Maven:org.hzero.boot:hzero-scheduler-saas-0.11.0.RELEASE.jar 第一个包是用于客户端的第二个是用于服务端的，由于客户端和服务端在同一个工程中所以 Maven 中会出现两个包且都在源码分析中用到。

最后对各包中涉及到的对象职责进行简单总结：

- 客户端
- JobExecuteController ： 接受服务端发送过来的请求
- JobExecuteServiceImpl：具体任务逻辑的执行位置，包括执行一个 JobHandler 、停止一个 JobHandler 操作等
- JobRegistry：维护 JobHandler 实例

- 服务端
- JobInfoController：提供操作 Job 的接口
- JobInfoServiceImpl：操作 Job 的逻辑实现，是传统意义上的 Service 层
- JobServiceImpl：是对 Quartz 封装的 Service，主要用于操作 Quartz 中的 Job 和触发器等
- UrlServiceImpl：定义了执行器策略的实现

在调度任务中的大致流程是，服务端根据 Corn 为 MyJob 当定一个触发器，当到达触发条件后 MyJob 得到执行然后根据参数向选中的执行器（客户端）发送一个请求，执行器收到请求后会从 JobRegistry 中找出 jobHandler 字段对应的对象，从而进行相应的动作。