XXL Job源码解析

一.XXL JOB项目源码整体概括

1. 源码整体概括说明

这个项目是作为工程开发的同学们很值得学习的一个开源项目。代码整体风格比较好，模块化清晰。代码逻辑遵行Web的MVC架构，采用Spring boot + Mybatis的框架组合来组织代码。

代码总体分为三部分：

一. xxl-job-core: 这是公共服务模块，比如提供RPC远程调度，线程管理等。从业务角度去分析这个模块是没有意义的，很容易一脑雾水，因为这个模块不是独立的服务，它只是为xxl-job-admin和xxl-job-executors-sample提供了功能模块。

二. xxl-job-admin: web交互的后台引擎，这里称为调度中心。主要负责下面几件事情：

1. 负责web端交互：作为Web后台引擎，提供了登录权限管理，任务增删改查操作，执行器组管理，GLUE任务在线编辑，日志管理等

2. 与MySQL数据库交互，把数据持久化。
3. 提供RPC接口，供执行器注册，维持和执行器的心跳。
4. 与quartz交互，把任务调度的事情交给quartz去做。

三. xxl-job-executors-sample。主要做以下两件事情：

1. 执行器初始化，并且主动注册到调度中心那里去。
2. bean的方式注入我们线下编辑好的任务。

整体架构图如下。后续章节会对细节进行展开阐述。

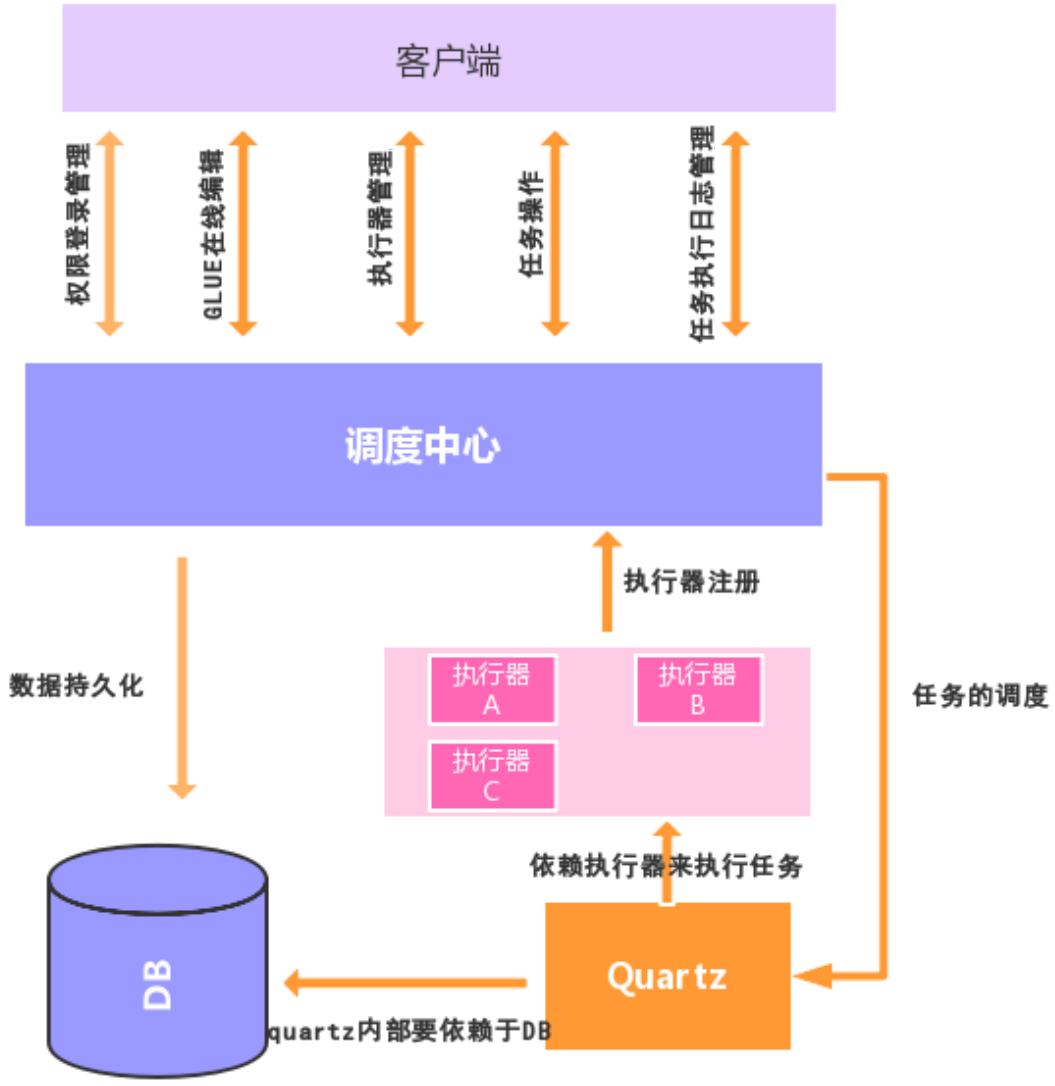

图1. 代码整体逻辑架构图

2.分析该项目源码时一些必须的知识

磨刀不误砍材工，在正式深入分析这个项目之前，有些知识有必要预知下：

1.quartz的用法。
2.freemarker渲染前端界面的原理和用法。
3.java基本功，以及spring boot和mybatis相关框架知识。

2.1 quartz简单介绍

xxl job的任务调度是依赖于quartz的。quartz可用于创建执行数十，数百甚至数十万个作业的简单或复杂的计划; 任务定义为标准Java组件的任务，可以执行任何可以对其进行编程的任何内容。我们先从quartz官网的一个例子说起：

```
// 第一步，定义任务类。这个class必须要实现Job接口的execute方法。
public class HelloJob implements Job {

    private static Logger _log = LoggerFactory.getLogger(HelloJob.class);
    public HelloJob() {
    }
    public void execute(JobExecutionContext context)
        throws JobExecutionException {
        _log.info("Hello World! - " + new Date());
    }

}

//2. 定义任务的执行逻辑，将任务和触发器绑定起来。
public class SimpleExample {

  public void run() throws Exception {

    log.info("------- 初始化----------------------");

    // 首先，我们得到一个scheduler实例
    SchedulerFactory sf = new StdSchedulerFactory();
    Scheduler sched = sf.getScheduler();

    log.info("------- 初始化完成 -----------");

    // computer a time that is on the next round minute
    Date runTime = evenMinuteDate(new Date());

    log.info("------- 调度任务 --------------------");

    // define the job and tie it to our HelloJob class
    JobDetail job = JobBuilder.newJob(HelloJob.class).withIdentity("job1", "gr
oup1").build();
```

```
        // Trigger the job to run on the next round minute
        Trigger trigger = TriggerBuilder.newTrigger().withIdentity("trigger1", "gr
oup1").startAt(runTime).build();

        // 告诉quartz利用trigger触发器来调度job
        sched.scheduleJob(job, trigger);
        log.info(job.getKey() + " will run at: " + runTime);

        // Start up the scheduler (nothing can actually run until the
        // scheduler has been started)
        sched.start();

        log.info("------- 任务已经已经启动了 -----------------");

        // wait long enough so that the scheduler as an opportunity to
        // run the job!
        log.info("------- Waiting 65 seconds... -------------");
        try {
          // wait 65 seconds to show job
          Thread.sleep(65L * 1000L);
          // executing...
        } catch (Exception e) {
          //
        }

        // shut down the scheduler
        log.info("------- 调度关闭 ---------------------");
        sched.shutdown(true);
        log.info("------- 关闭完成 -----------------");
    }

    public static void main(String[] args) throws Exception {

        SimpleExample example = new SimpleExample();
        example.run();

    }
}
```

从上面的demo可以看出quartz的关键API:

- Scheduler – 进行作业调度的主要接口.
- Job – 作业接口，编写自己的作业需要实现，如例子中的HelloJob
- JobDetail – 作业的详细信息，除了包含作业本身，还包含一些额外的数据。

- Trigger – 作业计划的组件–作业何时执行，执行次数，频率等。
- JobBuilder – 建造者模式创建 JobDetail实例.
- TriggerBuilder – 建造者模式创建 Trigger 实例.
- QuartzSchedulerThread 继承Thread 主要的执行任务线程

从上面的几个接口，可以看到quartz设计非常精妙，将作业和触发器分开设计，同时调度器完成对作业的调度。
整个执行过程可以概括如下：

1. 从StdSchedulerFactory获取scheduler
2. 创建JobDetail
3. 创建Trigger
4. scheduler.scheduleJob()将任务和触发器绑定起来

所以quartz的核心元素可以表示为如下图：

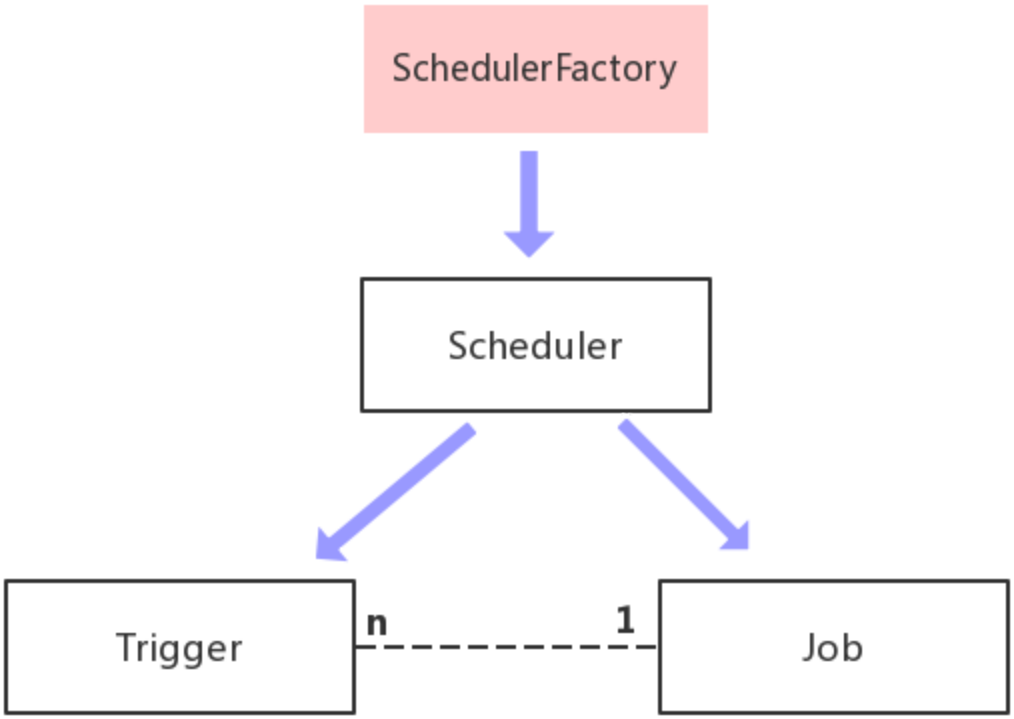

图2. quartz内部核心模块关系图

quartz不是以定时器的方式去执行任务的，而是通过线程池去完成。配置文件quartz.properties配置了线程池相关的参数。在quartz中，有两类线程，Scheduler调度线程和任务执行线程，其中任务执行线程通常使用一个线程池维护一组线程。

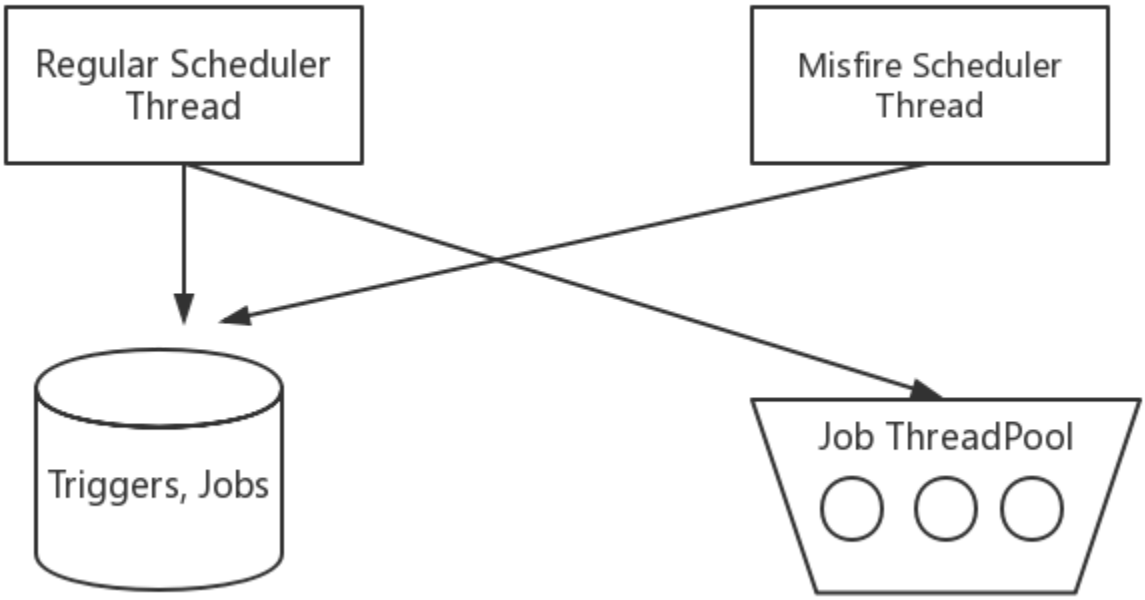

图3. quartz的线程视图

Scheduler调度线程主要有两个：执行常规调度的线程，和执行misfiredtrigger的线程。常规调度线程轮询存储的所有trigger，如果有需要触发的trigger，即到达了下一次触发的时间，则从任务执行线程池获取一个空闲线程，执行与该trigger关联的任务。Misfire线程是扫描所有的trigger，查看是否有misfiredtrigger，如果有的话根据misfire的策略分别处理(fire now 或者 wait for the next fire)。

quartz内部的数据是存入数据库的，总共有12张表。Quartz集群中，独立的Quartz节点并不与另一其的节点或是管理节点通信，而是通过相同的数据库表来感知到另一Quartz应用的。

到此，我认为quartz的核心要点应该介绍完了。

2.2 freemarker前端渲染模板简介

freemarker是一个java模板引擎。是一种基于模板和要改变的数据，并用来生成输出文本(HTML网页,电子邮件,配置文件,源代码等)的通用工具。类似于JSP,volecity。这里不细说，有这个概念就好了。

2.3 java基本功修炼

xxl job的源码阅读，需要一定的java工程功底。特别要熟悉下spring boot, mybatis框架。

二. xxl-job-admin的源码分析

xxl-job-admin是项目的核心，称为调度中心，也是一个典型的web项目架构。通常对于一个web程序来说，我们分析时，主要是关注两件事情：第一，这个程序在初始化（也就是程序启动的时候）干了哪些事情；第二，程序的Restful接口分析，这个是Web项目最大的主线。下面的分析我们也主要是从这两点分别展开。

1. 调度中心初始化

JVM执行一个java程序时，会经历编译，加载，分配内存和执行等过程。spring boot采用了的bean方式初始化了一些对象，这些对象包括了数据库连接池，前端界面渲染的引擎，配置文件读取，quartz调度引擎，拦截器等等，这些对象一旦初始化，就会从JVM的方法区里实例化到堆内存里面去，可以供进程后续的调用。这里有个和我们业务直接相关的bean初始化，代码如下：

```
<!--classpath:applicationcontext-xxl-job-admin-xml-->
<bean id="xxlJobDynamicScheduler" class="com.xxl.job.admin.core.schedule.XxlJobDynamicScheduler" init-method="init" destory-method="destory">
    <property name="scheduler" ref="quartzScheduler" />
    <property name="accessToken" value="${xxl.job.accessToken" />
</bean>
```

这个XxlJobDynamicScheduler类在初始化化时，执行了init方法。我们来重点分析下这个init方法干了哪些事情。

```
public void init() throws Exception {
  // 1. 调度中心注册守护线程，就是一直守护着执行器的注册，维持着和执行器之间的心跳
  JobRegistryMonitor.getInstance.start();

  // 2. 任务失败处理的守护线程
  JobFailMonitorHelper.getInstance().start();
  // 3. 初始化本地调度中心服务
  NetComServerFactory.putService(AdminBiz.class, XxlJobDynamicScheduler.adminBiz);
  NetComServerFactory.setAcessToken(accessToken);

  // 4.国际化
  initI18n();
  Assert.notNull(scheduler, "quartz scheduler is null");
  logger.info(">>>>>>>> init xxl-job admin success");
}
```

1. JobRegistryMonitor.getInstance.start()是开启了一个单独的线程，这个线程每30s去轮训一下数据库。如果某个执行器的注册信号（也叫作心跳）在近90s内没有写入数据库表XXL_JOB_QRTZ_TRIGGER_REGISTRY，那么调度中心就认为这个执行器已经死掉。然后会更新数

据库表XXL_JOB_QRTZ_TRIGGER_GROUP表，使每个执行器组，只保留活着的执行器。这里的执行器组是根据调度中心来区分的。每个执行器组（这个有可能是一台，也有可能是一个集群）都有一个唯一的appName，执行器向调度中心注册时就是通过这个appName标志来区分是属于哪个执行器组的。

2. JobFailMonitorHelper.getInstance().start()是一个失败任务处理的守护线程。这个线程是每隔10秒执行一下逻辑。数据库表XXL_JOB_QRTZ_TRIGGER_LOG里存着每个任务每次的执行记录，这里面记录着任务的执行状态。如果某条日志记录的处理状态码为500，那么这条执行记录是以失败告终的。那么失败守护线程就会根据这个任务的executorFailRetryCount（失败重试次数）是否大于零（这个参数是前端新增任务时配置的），如果大于零，会去尝试再执行下这个任务。并且相应地在数据库里把该条执行日志里的executorFailRetryCount值减1。最后发出失败告警。
3. 初始化本地的调度中心的服务Map,以及accessToken值。调度中心实例用HaspMap对象存了起来。
4. 国际化。支持中文和英文展示。

所以总的来说，这里主要是初始化了两个守护线程。一个是维持和执行器之间心跳的线程，一个是任务执行失败重试的线程。

2. Web MVC逻辑分析

Controller层是我们理解后台逻辑的入口，com.xxl.job.admin.controller包里面中共包含了六大模块：权限登录模块，调度中心和执行器通信的RPC模块，GLUE任务编辑模块，执行器管理模块，任务操作模块和任务日志管理模块。从用户正常的交互角度分析，这些模块是有先后顺序的。用户首先是通过账户密码登录系统，然后查看调度中心里有没有已经自动注册上的执行器，如果没有，那么需要手动添加执行器。后续就可以创建任务了。任务创建时，GLUE类型的任务可以在线编辑任务逻辑代码的。任务确认创建好了之后，可以手动即席执行，还可以配置cron表达式进行周期调度执行。最后通过日志界面，查看每个任务的执行逻辑。所以本节也会根据这个先后顺序来介绍每个模块的具体逻辑。

2.1. 权限登录模块

程序的配置文件里配置了初始化的username为admin，password为123456。spring boot的xml配置文件里配置了两个拦截器。具体如下：

```
<mvc:interceptors>
    <mvc:interceptor>
      <mvc:mapping path="/**" />
      <bean class="com.xxl.job.admin.controller.interceptor.PermissionIntercep
tor" />
    </mvc:interceptor>
    <mvc:interceptor>
      <mvc:mapping path="/**" />
      <bean class="com.xxl.job.admin.controller.interceptor.CookieInteceptor"
 />
```

```
    </mvc:interceptor>
</mvc:interceptors>
```

所以我们有必要先看这两个拦截器做了哪些事情。因为代码较多，所以这里只展现核心的逻辑出来。

```
public class PermissionInterceptor extends HandlerInterceptorAdapter{
  // 1. 静态代码块
  static {
      String username = XxlJobAdminConfig.getAdminConfig().getLoginUsername()
;
      String password = XxlJobAdminConfig.getAdminConfig().getLoginPassword()
;
      String tokenTmp = DigestUtils.md5Hex(username + "_" + password);
      tokenTmp = new BigInteger(1, tokenTmp.getBytes()).toString(16);
   }

  // 2. 拦截方法，登录方法
  public static boolean login(HttpServletResponse response, String username, S
tring password, boolean ifRemember){
    // 代码略
    // 整体逻辑为：
    // 1. 验证登录时的用户名和密码生成的MD5加密生成的token是否与内存中tokenTmp是否相等，
如果不等，就直接返回false。
    // 2. 如果相等，把token值以Cookie的方式存入response，当前端浏览器收到这个响应时，会
把cookie值自动存入浏览器会话窗口里。
  }

  // 3.登出，注销账户的方法
  public static void logout(HttpServletRequest request, HttpServletResponse res
ponse){
    // 将token值从response里删除
  }

  // 4. 判断客户端是否处于登录状态
  public static boolean ifLogin(HttpServletRequest request) {
    // 判断依据是：
    // 判断请求体request的Cookie里面token值是否和内存里的tokenTmp相等。如果浏览器存着c
ookie值，那么每次请求，浏览器都会自动把cookie带上的。

    // 如果相等，那么处于登录状态；如果不等，那么处于非登录状态。
  }

  // 5. 服务端收到客户端请求时的拦截方法,默认任何请求都会先经过这层拦截器
  @Override
  public boolean preHandler(HttpServletRequest request, HttpServletResponse re
```

```
sponse, Object handler) throws Exception{
        // 逻辑说明
        // 通过ifLogin如果客户端不是处于登录状态，那么需要重定向到登录页。
    }
}
```

```
public class CookieInterceptor extends HandlerInterceptorAdapter {
    // 这是服务端响应客户端请求时执行的拦截，任何请求的响应之后，都会执行这个拦截器。
    @Override
    public void postHandle(HttpServletRequest request, HttpServletResponse respon
se, Object handler, ModelAndView modelAndView) throws Exception {
        // 执行的整体逻辑，把cookies全部以cookieMap对象存入modelAndView中。
    }
}
```

以上是两个拦截器的总体逻辑，这里读者如果有疑问，最好是先去熟悉下拦截器的原理。
下面正式开始Controller的逻辑分析。

2.1.1. 访问首页路由的方法执行体

访问首页代码的执行体：

```
@RequestMapping("/")
public String index(Model model){
    // 1. 请求model数据
    Map<String,Object> dashboardMap = xxlJobService.dashboardInfo();
    model.addAllAttributes(dashboard);
// 2. 返回index.ftl模板
    return "index";
}
```

因为默认所以的请求都会先经过拦截器来判断浏览器会话层是否处于登录状态，所以这个路由的执行体在执行这部分逻辑之前，是要先经过拦截器的preHandler方法的。如果preHandler返回true，才再执行index函数执行体里面的逻辑。这也是spring框架比较隐晦的地方。后续的路由分析将不再强调这个，因为是一样的思想。
dashboardMap是Model的数据。这段逻辑是去统计数据库里的运行报表信息，包括任务数量，调度次数，执行器数量等三部分。然后将这些数据存入dashboardMap。返回前端的界面模板是index.ftl。
freemarker模板引擎会结合model数据和views模板，最后把界面完整地呈现给前端浏览器。
首页的渲染，出了上述逻辑，还有日期分布折线图和成功比例饼型图的绘制。这两个图的绘制所需数据是请求的另一个接口。这个接口的请求，是前端的ajax请求发出的。路由逻辑如下：

```
@RequestMapping("/chartInfo")
@ResponseBody
public ReturnT<Map<String,Object>> chartInfo(Date startDate, Date endDate){
  ReturnT<Map<String, Object>> chartInfo = xxlJobService.chartInfo(startDate,
 endDate);
  return chartInfo;
}
```

这段执行体的主要逻辑是根据开始时间startDate和结束时间endDate去数据库里面捞出这段时间内的统计信息。默认是请求近一个月的数据信息。这里的统计数据是从XXL_JOB_QRTZ_TRIGGER_LOG表里面获得的。最后返回给前端的是json数据格式chartInfo。这里有必要强调下方法体的注解@ResponseBody的作用，如果有这个注解，那么返回给前端的必须是json数据格式。

2.1.2. 访问登录页的方法执行体

用户登录页的controller层代码如下：

```
@RequestMapping("/toLogin")
@PermessionLimit(limit=false)
public String toLogin(Model model, HttpServletRequest request){
  if(PermissionInteceptor.ifLogin(request)){
      return "redirect:/"
   }
   return "login";
}
```

这段代码是先判断客户端是否为登录状态，如果是，那么直接重定向首页，无需用户再进行登录。如果不是处于登录状态，那么久给前端返回login.ftl模板。也就是进入了登录页。

2.1.3. 登录事件的方法执行体

登录事件执行体逻辑如下：

```
@RequestMapping(value="login", method=RequestMethod.POST)
@ResponseBody
@PermessionLimit(limit=false)
public ReturnT<string> loginDo(HttpServletRequest request, HttpServletResponse
 response, String userName, String password, String ifRemember){
  // 1. 先验证是否处于登录状态，如果是，直接返回success
  // 2. 校验用户名和密码的合法性，不能为空值
  // 3. 判断是否要记住密码
  // 4. 执行登录逻辑（这个逻辑很简单，就是判断用户输入的用户名和密码的md5加密值和后台的tmpT
oken是否相等。）如果相等，那么就把token值存入响应体，最后会把cookie存到浏览器。
```

```
    // 5. 判断是否登录成功
}
```

因为这部分的代码相对而言有点长，所以代码没有罗列了。只写出了代码的执行逻辑。这样读者对着代码去读，就会觉得思路很清晰的。注意这个登录请求时post请求，注解里有标注。

2.1.4. 登出事件的方法执行体

代码如下：

```
@RequestMapping(value="logout", method=RequestMethod.POST)
@ResponseBody
@PermssionLimit(limit=false)
public ReturnT<String> logout(HttpServletRequest request, HttpServletResponse r
esponse){
    if(PermissionInterceptor.ifLogin(request)){
        PermissionInterceptor.logout(request, response);
    }
    return ReturnT.SUCCESS;
}
```

先判断客户端是否处于登录状态，如果是，那么就执行PermissionInterceptor.logout函数体。这个函数的逻辑是将response响应体的Cookie删掉了。从而会更新客户端（浏览器）里的cookie为空值。从未就是客户端不符合登录状态。

2.1.5. helper界面请求的方法执行体

这个模块里还内插了一个小的controller，如下：

```
@RequestMapping("/help")
public String help(){
    return "help"
}
```

这个逻辑没有做任何事情，就是返回静态页help.ftl。

到此调度中心的权限登录逻辑分析完了，这也是一个基本的Web应用产品都要做的事情。这里的思想甚至都可以行成模块。

2.2. 调度中心和执行器的RPC通信模块

说起调度中心和执行器之间的RPC通信模块，这里有必要先提下执行器。后续章节会具体讲解执行器的逻辑，但是这里要先提下执行器和调度中心的简单的交互逻辑。执行器自动注册时，主要是向调度中心发起RPC请求，所以这里调度中心供一个rpc接口供执行器过来注册。rpc框架需要解决的一个问题是：像调用本地接口一样调用远程的接口。于是如何组装数据报文，经过网络传输发送至服务提供方，屏蔽远程接口调用的细节，便是动态代理需要做的工作。

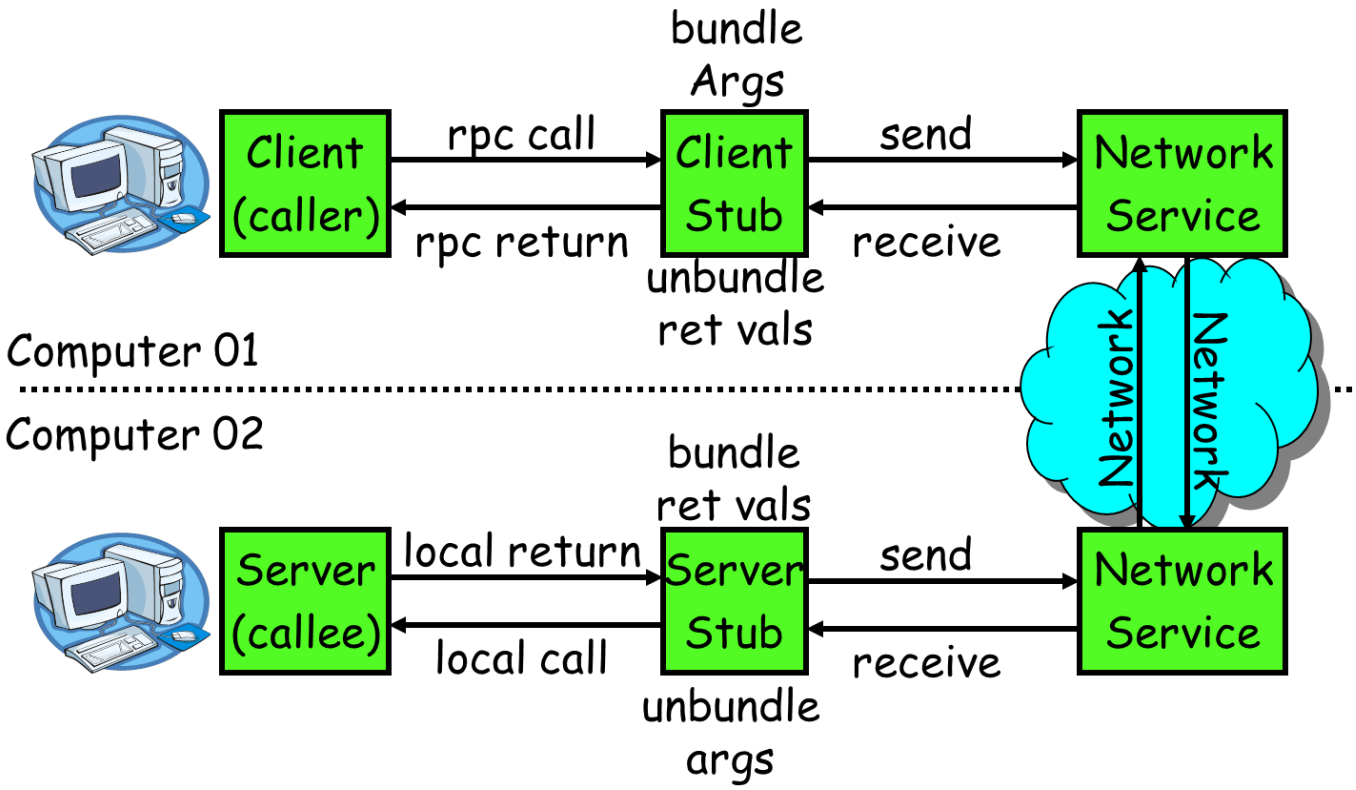

图4. RPC过程示意图

如上图所示，假设Computer1在调用sayHi()方法，对于Computer1而言调用sayHi()方法就像调用本地方法一样，调用 ->返回。但从后续调用可以看出Computer1调用的是Computer2中的sayHi()方法，RPC屏蔽了底层的实现细节，让调用者无需关注网络通信，数据传输等细节。

RPC的整体执行过程可以描述为：

1）服务消费方（client）调用以本地调用方式调用服务；

2）client stub接收到调用后负责将方法、参数等组装成能够进行网络传输的消息体；

3）client stub找到服务地址，并将消息发送到服务端；

4）server stub收到消息后进行解码；

5）server stub根据解码结果调用本地的服务；

6）本地服务执行并将结果返回给server stub；

7）server stub将返回结果打包成消息并发送至消费方；

8）client stub接收到消息，并进行解码；

9）服务消费方得到最终结果

调度中心提供这个RPC接口的代码逻辑为：

```
private RpcResponse doInvoke(HttpServletRequest request) {
        try {
```

```
            // 反序列化请求体
            byte[] requestBytes = HttpClientUtil.readBytes(request);
            if (requestBytes == null || requestBytes.length==0) {
                RpcResponse rpcResponse = new RpcResponse();
                rpcResponse.setError("RpcRequest byte[] is null");
                return rpcResponse;
            }
            RpcRequest rpcRequest = (RpcRequest) HessianSerializer.deserialize
(requestBytes, RpcRequest.class);

            // 触发
            RpcResponse rpcResponse = NetComServerFactory.invokeService(rpcR
equest, null);
            return rpcResponse;
        } catch (Exception e) {
            logger.error(e.getMessage(), e);

            RpcResponse rpcResponse = new RpcResponse();
            rpcResponse.setError("Server-error:" + e.getMessage());
            return rpcResponse;
        }
    }

@RequestMapping(AdminBiz.MAPPING)
@PermessionLimit(limit=false)
public void api(HttpServletRequest request, HttpServletResponse response) throw
s IOException {

//触发，动态代理执行逻辑
RpcResponse rpcResponse = doInvoke(requst);

//序列化
byte[] responseBytes = HessianSerializer.serialize(rpcResponse);

response.setContentType("text/html;charset=utf-8");
response.setStatus(HttpServletResponse.SC_OK);

OutputStream out = response.getOutputStream();
out.write(responseBytes);
out.flush();

}
```

这段代码主要执行逻辑在这一行：RpcResponse rpcResponse = doInvoke(request)。执行逻辑总结为如下流程：

1. 反序列化请求体数据。
2. 反射出服务器里的执行类方法和参数，并且执行服务端（调度中心）的方法。这里要强调下在执行主体方法之前，程序先判断了远程rpc请求者的系统时间和当前调度中心的系统时间的差值，如果时间差超过3分钟，那么就响应错误，这个时候就需要先做时间同步操作。
3. 对执行方法的结果进行序列化。返回给客户端（执行器）。

这是服务端提供的rpc服务接口，然后具体客户端那边怎么调用，后续在分析执行器模块时再重点分析下。

2.3. GLUE任务编辑模块

在线编辑任务代码主要针对GLUE类型的任务。如果是bean类型的任务是不提供编辑修改任务代码的。该模块主要提供的功能有回溯GLUE类任务的历史版本，以及更新任务的脚本内容等操作。

2.3.1 请求加载任务的GLUE 代码

该接口的controller代码如下：

```
@RequestMapping
public String index(Model model, int jobId){
    // 1. 通过jobId获取任务信息。
  XxlJobInfo = xxlJobInfoDao.loadById(jobId);

    // 2. 通过jobId获取GLUE历史版本信息。
  List<XxlJobLogGlue> jobLogGlues = xxlJobLogGlueDao.findByJobId(jobId);
  // 3. 验证任务是否存在
  if(jobInfo == null){
      throw new RuntimeException(I18nUtil.getString("jobinfo_glue_jobid_invali
d");
  }
  // 4. 验证任务是否是GlueType，如果不是抛出异常。
  if(GlueTypeEnum.BEAN == GlueTypeEnum.match(jobInfo.getGlueType())){
    throw new RuntimeException(I18nUtil.getString("jobinfo_glue_gluetype_unval
id"));
  }

  // 5. 将数据和模板返回
  model.addAttribute("GlueTypeEnum", GlueTypeEnum.values());
  model.addAttribute("jobInfo",jobInfo);
  model.addAttribute("jobLogGlues",jobLogGlues);
  return "jobcode/jobcode.index";
}
```

这个逻辑在代码注释里写比较清晰了，这里一次性返回了该任务的历史所有GLUE版本，通过前端去操作可以编辑任务版本的脚本内容 。

2.3.2 保存编辑的GLUE内容

任务在线编辑好之后，可以点击保存按钮，将新的逻辑脚本更新到数据库。

```
@RequestMapping("/save")
@ResponseBody
public ReturnT<String> save(Model model, int id, String glueSource, String glu
eRemark){
  // 逻辑有点长，所以这里只列出执行逻辑：
  // 概括起来，就是通过任务id，glue内容以及对应的glueRemark（版本）去更新数据库，将该版本的
脚本内容保存起来
}
```

这个接口要接受前端参数：任务id，脚本内容，以及对应得版本号，去执行修改版本的脚本内容。这里的版本号也就是该任务的最新版本号了，后续任务执行时是读取最新的版本号内容去执行的。

2.4. 执行器管理模块

同一个执行器集群内AppName（xxl.job.executor.appname）需要保持一致，AppName执行器心跳注册分组依据。一个执行器组内可以只有一个执行器，也可以有多个执行器。当有多个执行器时，这些执行器的地址是使用逗号相连的方式存入数据库表的。

2.4.1 获取执行器列表

controller层代码逻辑：

```
@RequestMapping
public String index(Model model) {

    // job group (executor)
    List<XxlJobGroup> list = xxlJobGroupDao.findAll();

    model.addAttribute("list", list);
    return "jobgroup/jobgroup.index";
}
```

这个接口是去数据库里表XXL_JOB_QRTZ_TRIGGER_GROUP里插入该执行器组的所有执行器。将所以的执行器查询出来。然后把列表返回给前端界面。

2.4.2 添加新的执行器

添加执行器的接口逻辑如下：

```
@RequestMapping("/save")
@ResponseBody
public ReturnT<String> save(XxlJobGroup xxlJobGroup){

    // 1. 校验：appName检验，执行器名称校验，如果是手动注册，那么要验证执行器address的合法性
    if (xxlJobGroup.getAppName()==null || StringUtils.isBlank(xxlJobGroup.getAppName())) {
        return new ReturnT<String>(500, (I18nUtil.getString("system_please_input")+"AppName") );
    }
    if (xxlJobGroup.getAppName().length()<4 || xxlJobGroup.getAppName().length()>64) {
        return new ReturnT<String>(500, I18nUtil.getString("jobgroup_field_appName_length") );
    }
    if (xxlJobGroup.getTitle()==null || StringUtils.isBlank(xxlJobGroup.getTitle())) {
        return new ReturnT<String>(500, (I18nUtil.getString("system_please_input") + I18nUtil.getString("jobgroup_field_title")) );
    }
   // 这里特别是对执行器的address列表进行了校验
    if (xxlJobGroup.getAddressType()!=0) {
        if (StringUtils.isBlank(xxlJobGroup.getAddressList())) {
            return new ReturnT<String>(500, I18nUtil.getString("jobgroup_field_addressType_limit") );
        }
        String[] addresss = xxlJobGroup.getAddressList().split(",");
        for (String item: addresss) {
            if (StringUtils.isBlank(item)) {
                return new ReturnT<String>(500, I18nUtil.getString("jobgroup_field_registryList_unvalid") );
            }
        }
    }

    // 2. 将新的执行器信息添加到数据库
    int ret = xxlJobGroupDao.save(xxlJobGroup);
    return (ret>0)?ReturnT.SUCCESS:ReturnT.FAIL;
}
```

这里的逻辑是比较清晰的。对新添加的执行器的相关数据校验，然后就数据存到数据库里面去。同一执行器组的appName要一样。

2.4.3 更新执行器信息

代码如下：

```
@RequestMapping("/update")
@ResponseBody
public ReturnT<String> update(XxlJobGroup xxlJobGroup){
    // 1.校验数据，和2.4.2接口的校验方式类似.
    // 更新数据库
}
```

这里的第一步校验数据整体是和2.4.2接口类似的。但是增加了对注册方式的处理，执行器的注册方式有两种：手动注册和自动注册。如果是自动注册，就通过注册时的appName参数去获取数据库表已经存在的执行器组内的执行器address列表,有点类似于复制了一个执行器组。如果是手动注册,那么只要校验填写的addressList是否符合规范。

2.4.4 删除执行器

逻辑代码：

```
@RequestMapping("/remove")
@ResponseBody
public ReturnT<Stirng> remove(int id){
  // 1. 如果该执行器正在执行任务，那么是不允许删除该任务的
  int count = xxlJobInfoDao.pageListCount(0,10,id,null,null);
  if(count > 0){
     return new ReturnT<500,I18nUtil.getString("jobgroup_del_limit_0");
   }
  // 2.如果执行器组里只剩下一个执行器了,那么这个操作是不允许再删除执行器了.
  List<XxlJobGroup> allList = xxlJobGroupDao.findAll();
  if (allList.size() == 1) {
     return new ReturnT<String>(500, I18nUtil.getString("jobgroup_del_limit_1")
 );
  }

  int ret = xxlJobGroupDao.remove(id);
  return (ret>0)?ReturnT.SUCCESS:ReturnT.FAIL;
}
```

这里主要是在执行数据库删除执行器之前，先做两步验证，首先是如果有任务和该执行器绑定，那么这个执行器是不能删除的，然后如果执行器组仅剩下最后一个执行器了，那么执行器也是不能删除的。

2.5 任务操作模块

任务操作模块主要是针对任务的增删改查等操作以任务的执行，杀死，暂停,恢复执行等操作。但这一块的功能我在测试过程中，发现存在着争议,下面就一一分析下这些接口。

2.5.1 获取任务列表

Controller的代码如下:

```
@RequestMapping
public String index(Model model, @RequestParam(required = false, defaultValue
 = "-1") int jobGroup) {

    // 枚举-字典
    model.addAttribute("ExecutorRouteStrategyEnum", ExecutorRouteStrategyEnum.
values()); // 路由策略-列表
    model.addAttribute("GlueTypeEnum", GlueTypeEnum.values());
                // Glue类型-字典
    model.addAttribute("ExecutorBlockStrategyEnum", ExecutorBlockStrategyEnum
.values());     // 阻塞处理策略-字典

    // 任务组
    List<XxlJobGroup> jobGroupList =  xxlJobGroupDao.findAll();
    model.addAttribute("JobGroupList", jobGroupList);
    model.addAttribute("jobGroup", jobGroup);

    return "jobinfo/jobinfo.index";
}


@RequestMapping("/pageList")
@ResponseBody
public Map<String, Object> pageList(@RequestParam(required = false, defaultVal
ue = "0") int start,
        @RequestParam(required = false, defaultValue = "10") int length,
        int jobGroup, String jobDesc, String executorHandler, String filterTime
) {
    return xxlJobService.pageList(start, length, jobGroup, jobDesc, executorHa
ndler, filterTime);
}
```

index函数是列出路由策略，GLUEType类型，执行器阻塞类型，执行器组等列表信息，这些信息最终是渲染到jobinfo.index.ftl中去，形成前端界面任务管理界面里的条件选择。就是用户可以选择条件（比如路由策略等）去查询符合条件的任务。界面初始化时发起了一个ajax请求，这个ajax请求就是请求了任务列表api。也就是代码里的/pageList路由。这个路由就是分页查询任务列表了。

2.5.2 新增任务

新增任务的Controller：

```
@RequestMapping("/add")
@ResponseBody
public ReturnT<String> add(XxlJobInfo jobInfo) {
    return xxlJobService.add(jobInfo);
}
```

整体逻辑体现在xxlJobService.add(jobInfo)里，首先是检验参数的合理性，特别是对子任务的id进行了校验，就是要保证子任务是已经存在的任务；然后把新增的任务存到存到数据库表xxl_job_qrtz_trigger_info里面去；最后，调用quartz的scheduler.scheduleJob()接口，把任务的cron调度交给quartz去管理，将任务和执行器绑定起来。这里整体逻辑支持了事务回滚，如果最后一步失败，那么会把表xxl_job_qrtz_trigger_info里刚刚插入的任务给删掉。

2.5.3 更新任务

代码如下：

```
@RequestMapping("/update")
@ResponseBody
public ReturnT<String> update(XxlJobInfo jobInfo) {
   return xxlJobService.update(jobInfo);
}
```

xxlJobService.update的逻辑和add的逻辑类似，首先也是校验数据，然后是更新数据库表xxl_job_qrtz_trigger_info，最后调用quartz的scheduler.rescheduleJob()接口重新调度该任务。

2.5.4 删除任务

```
    @RequestMapping("/remove")
    @ResponseBody
    public ReturnT<String> remove(int id) {
        return xxlJobService.remove(id);
    }
```

删除任务首先是调用quartz的scheduler.unscheduleJob()接口，相当于是卸载任务；然后是去数据库表该任务相关的东西先删除掉，比如任务信息，任务日志信息，还有任务的GLUE历史版本信息，这些全部要删除掉。

2.5.5 暂停任务

```
@RequestMapping("/pause")
@ResponseBody
public ReturnT<String> pause(int id) {
    return xxlJobService.pause(id);
}
```

暂停任务其实是调用quartz的schedule.pauseTrigger()接口，不过这个接口在测试的过程是由问题的，就是对某个正在执行的任务执行这个暂停操作，其实没效果，因为无法将一个正在执行的任务进程杀死掉。这是个bug。

2.5.6 恢复任务

```
@RequestMapping("/resume")
@ResponseBody
public ReturnT<String> resume(int id) {
    return xxlJobService.resume(id);
}
```

当任务暂停之后，然后再把其恢复继续执行，这个接口实际上执行的是quartz的scheduler.resumeTrigger()接口，但是在真实的测试环节中，这个接口也是没有效果。感觉也是个bug。

2.5.7 触发任务

```
@RequestMapping("/trigger")
@ResponseBody
//@PermessionLimit(limit = false)
public ReturnT<String> triggerJob(int id, String executorParam) {
    // force cover job param
    if (executorParam == null) {
        executorParam = "";
    }
    // 触发core依赖模块里的事件，触发类型是手动，
    JobTriggerPoolHelper.trigger(id, TriggerTypeEnum.MANUAL, -1, null, executor
Param);
    return ReturnT.SUCCESS;
}
```

触发任务执行的整体逻辑是执行JobTriggerPoolHelper.trigger函数，这里尤其要注意该函数的第二个参数，代表了触发类型，这是手动触发。其背后逻辑是新创建一个任务扔到触发线程池里面去执行，触发线程池是一个coreSize=32，maxSize=256的线程池。所以任务不是立马执行的,而是线程池的方式执行的,任务只是添加到了阻塞队列里去了。
调度中心会根据任务的路由策略去选择哪个执行器来执行任务，比如执行器组的第一个执行器,执行器组的最后一个执行器,或者随机获取一个执行器等等，但是如果该任务是分片广播的方式去执行任务，情况要注意下。每个任务是和一个执行器组绑定在一起的，调度会把任务下发到该执行器组的每一个执行器，让每一个执行器都执行该任务。

2.6. 任务日志管理模块

2.6.1 日志管理界面初始化

日志管理界面的初始化是请求了两个接口：

```
@RequestMapping
public String index(Model model, @RequestParam(required = false, defaultValue
= "0") Integer jobId){
    List<XxlJobGroup> jobGroupList = xxlJobGroupDao.findAll();
    model.addAttribute("JobGroupList", jobGroupList);

    if(jobId > 0){
        XxlJobInfo jobInfo = xxlJobInfoDao.loadById(jobId);
        model.addAttribute("jobInfo",jobInfo);
    }
    return "joblog/joblog.index";
}
```

这是日志管理界面初始化时首先要加载到所有的执行器，执行器列表是作为前端的筛选条件的。返回joblog.index.ftl模板。如果jobId大于0,那么就展现该任务的日志信息。随之，前端通过ajax请求了pageList接口，加载日志列表：

```
@RequestMapping("/pageList")
@ResponseBody
public Map<String,Object> pageList(@RequestParam(required = false, defaultValu
e = "0") int start,
        @RequstParam(required = false, defaultValue="10") int length, int jobG
roup, int jobId,
       int logStatus, String filterTime){
    Date triggerrTimeStart = null;
    Date triggerTimeEnd = null;
    if(StringUtils.isNotBlank(filterTime)){
```

```
        String[] temp = filterTime.split("-");
        if(temp != null && temp.length == 2){
            try {
                triggerTimeStart = DateUtils.parseDate(temp[0], new String[]{"yyyy-M
M-dd HH:mm:ss"});
                triggerTimeEnd = DateUtils.parseDate(temp[1], new String[]{"yyyy-MM-
dd HH:mm:ss"});
            } catch(ParseException e){ }
        }
    }
    List<XxlJobLog> list = xxlJobLogDao.pageList(start, length, jobGroup, jobId
, triggerTimeStart, triggerTimeEnd,logStatus);
    int list_count = xxlJobLogDao.pageListCount(start, length, jobGroup, jobId,
 triggerTimeStart, triggerTimeEnd,logStatus);

    Map<String, Object> maps = new HashMap<String, Object>();
    maps.put("recordsTotal", list_count);
    maps.put("recordsFiltered", list_count);
    maps.put("data",list);
        return maps;

}
```

这个接口就是分页查询日志列表了，根据查询条件去查询符合条件的日志。

2.6.2 通过执行器ID查询该执行器下面的级联任务

接口代码：

```
@RequestMapping("/getJobsByGroup")
@ResponseBody
public ReturnT<List<XxlJobInfo>> getJobsByGroup(int jobGroup){
    List<XxlJobInfo> list = xxlJobInfoDao.getJobsByGroup(jobGroup);
    return new ReturnT<List<XxlJobInfo>>(list);
}
```

这个接口提供了通过执行器去加载该执行器下面的任务。这是在条件筛选时触发的，级联条件筛选。

2.6.3 清除日志

接口代码：

```
@RequestMapping("/clearLog")
@ResponseBody
public ReturnT<String> clearLog(int jobGroup, int jobId, int type){
```

```
    // 按照时间返回，清除日志
}
```

这个接口含义就是根据清理方式（type)来对日志清晰清除，比如清理一个月之前的日志，清晰一周之前的日志。一般来说，越早的日志对我们的价值越小，用户主要关注近期的日志。

备注：代码里还有其他的接口：/logKill, /logDetailPage, /logDetailCat。这三个接口，貌似没有看到它们的用处，暂时不做分析。

三 .执行器代码分析

执行器主要是做两部分工作，第一，负责任务的执行。第二，负责初始化线下编辑好的bean任务。执行器模块是以bean的方式来初始化对象的，这里先看xml配置文件。

```
<bean id="xxlJobExecutor" class="com.xxl.job.core.executor.XxlJobExecutor" i
nit-method="start" destory-method="destory" >
    <property name="adminAddresses" value="${value.job.admin.addresses}" />
    <property name="appName" value="${xxl.job.executor.appname" />
    <property name="ip" value="${xxl.job.executor.ip}" />
    <property name="port" value="${xxl.job.executor.port}" />
    <property name="accessToken" value="${xxl.job.accessToken" />
    <property name="logPath" value="${xxl.job.executor.logpath}" />
    <property name="logRetentionDays" value="${xxl.job.executor.logretentionda
ys}" />
</bean>
```

这个bean初始化时执行了init-method方法.所以XxlJobExecutor的start方法是执行器服务初始化时做的处理。 重点关注下这部分初始化逻辑。

```
public void start() throws Exception {
    //1. 初始化调度中心的本地列表
    initAdminBizList(adminAddresses,accessToken);

    //2. 初始化执行器的任务库
    initJobHandlerRepository(applicationContext);

    // 3. 初始化日志路径
     XxlJobFileAppender.initLogPath(logPath);

    // 4. 初始化执行服务器
     initExecutorServer(port,ip, appName, accessToken);
```

```
        // 5. 日志清除线程
        JobLogFileCleanThread.getInstance().start(logRetionDays);
    }
```

这部分程序初始化逻辑主要做了5件事情，下面我们具体依次对这5件事情进行分析。

1. 初始化调度中心的本地列表。这里是为每一个调度中心的地址创建了一个RPC实例，以供执行器远程调用调度中心的方法。就将这些RPC实例存到本地。执行器和调度中心之间通信采用了Hessian框架进行数据加密。
2. 执行器部署的时候是内置有bean的任务的，这些任务具有一个特点，那就是实现了IjobHandler接口，并且带有JbHandler注解。第二步就是获取到这些任务实例，并存到本地库中。
3. 初始化日志路径。
4. 初始化执行服务器。
5. 日志清除线程。这里开启了一个独立线程，每天执行一次去轮询数据库的日志表，将时间超过logRentetionDays的日志删掉，以减轻数据库的压力。

在这里要重点强调下调度中心和执行器之间的RPC交互。一方面，执行器通过rpc去调度中心注册，以及维持心跳；另一方面，当调度中心要执行任务时，是远程调用执行器的方法。RPC要求远程类和本地创建的代理类要集成共同的接口，所以RPC模块放在了xxl-job-core公共库。这样执行器模块和调度中心模块都需要依赖。

四.XXL Job需要改进的地方

1. 对任务的操作还是有点问题，比如说，任务暂停，任务杀死，任务恢复执行，实际测试中，这些操作没有效果。
2. 用户在新创建bean类型任务时，对已经存在的bean类型任务名称无感知。会影响jobHandler参数的填写。然后对子任务的Id也没有感知，这个也影响交互。

五.总结说明

本次源码分析是针对xxl job的1.9.2版本。github最新的版本改动较大，项目全面迁移到了spring boot，减少了很多配置项。特别是rpc模块，作者利用了自己开源的一个xxl-rpc-core框架，这个框架的网络模型是基于NIO，性能比BIO要好。