学 号

密 级

哈尔滨工程大学学士学位论文

作业调度平台中心控制系统的设计与实现

院（系）名　称：软件学院

专　业　名　称：软件工程

学　生　姓　名：李佰波

指　导　教　师：马志强

哈尔滨工程大学

2019年6月

**人员社会关系图谱可视化加速算法设计与实现 哈尔滨工程大学**

学 号

密 级

作业调度平台中心控制系统的设计与实现

Design and Implementation of Central Control System for Job Scheduling Platform

**学生姓名：**李佰波

**所在学院：**软件学院

**所在专业：**软件工程

**指导教师：**马志强

**职称：**

**所在单位：**哈尔滨工程大学

**论文提交日期：**2019年6月

**论文答辩日期：**2019年6月

**学位授予单位：**哈尔滨工程大学

**摘 要**

中国数值水池网站创立初期肯定不可能拥有庞大的用户流量和海量数据，而是一步步演变其自身架构满足自身业务，因为系统的做大往往与业务的做大呈正比的。对于一个刚上线的项目，我们往往会将Web服务器、文件服务器和数据库全都部署在同一台物理服务器上。单机结构的优点在于使用简单，配置成本低，数据共享程度高，一致性好，而当业务发展，计算任务激增，用户量不断上升，系统瓶颈便开始暴露出来，需要解决的主要问题是提升系统的并行处理能力，

本课题旨在设计并实现一套面向集群的核心作业调度管理系统，该系统能够完成求解程序的分发，同时提供多种调度策略，能够综合求解器的特点及各节点实际运行状态，合理地进行任务的调度。从而降低单机系统负载，一边支撑更多的用户访问操作以及计算求解任务量， 面向集群的作业调度技术可以将多台独立的服务器通过网络相互连接组合起来，形成一个有效整体对外提供服务，使用集群的意义在于其目的收益高于所付出的实际成本和代价。

本课题研究工作结合集群技术与调度策略算法，为解决系统自身的健壮性，高可用性，高并发以及负载均衡提供了新的技术手段。大量的实验结果证明了本文所提供方法的实用性和有效性。

**关键字：**作业调度；集群；高可用；高并发；负载均衡；

**ABSTRACT**

At the beginning of the establishment of the China Digital Pool website, it is impossible to have huge user traffic and massive data, but to gradually evolve its own architecture to meet its own business, because the system is often bigger than the business. For a project that is just online, we tend to deploy the web server, file server, and database all on the same physical server. The advantage of the stand-alone structure is that it is simple to use, low in configuration cost, high in data sharing, and good in consistency. When the business develops, the number of computing tasks increases, and the number of users continues to rise, the system bottleneck begins to be exposed. The main problem to be solved is to upgrade the system. Parallel processing power,

This project aims to design and implement a cluster-oriented core job scheduling management system. The system can complete the distribution of the solver and provide various scheduling strategies. It can comprehensively solve the characteristics of the solver and the actual running status of each node. Scheduling of tasks. Thereby reducing the load of the single system, while supporting more user access operations and calculating the solution task, the cluster-oriented job scheduling technology can connect multiple independent servers through the network to form an effective overall external service, using the cluster. The significance is that the purpose of the benefits is higher than the actual cost and cost.

This research work combines cluster technology and scheduling strategy algorithms to provide new technical means for solving the system's own robustness, high availability, high concurrency and load balancing. A large number of experimental results prove the practicability and effectiveness of the methods provided in this paper.

**Key words:** job scheduling; cluster; high availability; high concurrency; load balancing;

目 录

[第1章 绪论 1](#_Toc517267133)

[1.1课题背景 1](#_Toc517267134)

[1.2课题目的和意义 1](#_Toc517267135)

[1.3 国内外研究现状 2](#_Toc517267136)

[1.4 主要研究内容 4](#_Toc517267137)

[1.5 论文结构安排 4](#_Toc517267138)

[第2章 系统总体设计与实现 6](#_Toc517267139)

[2.1 系统总体架构设计 6](#_Toc517267140)

[2.2 求解器按需部署总体设计 6](#_Toc517267141)

[2.3求解器自动同步机制设计 8](#_Toc517267142)

[2.4 基于多线程的人员社会关系图谱展示加速算法设计 9](#_Toc517267143)

[2.5 本章小结 10](#_Toc517267144)

[第3章 集群各计算节点资源状态的收集 11](#_Toc517267145)

[3.1 总体实现 11](#_Toc517267146)

[3.2 人员社会关系图谱节点和关系解析 12](#_Toc517267147)

[3.3 人员社会关系图谱节点分层 13](#_Toc517267148)

[3.3.1 基于社团发现算法的图谱聚类 13](#_Toc517267149)

[3.3.2 构造抽象图谱 15](#_Toc517267150)

[3.4人员社会关系图谱节点和关系布局 18](#_Toc517267151)

[3.4.1 KK图谱能量模型 18](#_Toc517267152)

[3.4.2 基于KK图谱算法抽象图谱布局 20](#_Toc517267153)

[3.4.3 基于KK图谱算法子图谱布局 21](#_Toc517267154)

[3.5 本章小结 23](#_Toc517267155)

[第4章 多种调度策略的算法与实现 24](#_Toc517267156)

[4.1 总体实现 24](#_Toc517267157)

[4.2展示策略 25](#_Toc517267158)

[4.3 XML解析 26](#_Toc517267159)

[4.4 节点展示和关系展示 28](#_Toc517267160)

[4.5 本章小结 29](#_Toc517267161)

[第5章 实验结果与分析 31](#_Toc517267162)

[5.1 总体设计 31](#_Toc517267163)

[5.2 基于KK图谱算法的人员社会关系图谱布局加速算法实验 31](#_Toc517267164)

[5.2.1 实验设计 31](#_Toc517267165)

[5.2.2 实验结果 32](#_Toc517267166)

[5.2.3 实验分析 34](#_Toc517267167)

[5.3 基于多线程的人员社会关系图谱展示加速算法实验 35](#_Toc517267168)

[5.3.1 实验设计 35](#_Toc517267169)

[5.3.2 实验结果 35](#_Toc517267170)

[5.3.3 实验分析 36](#_Toc517267171)

[5.4 人员社会关系图谱可视化加速算法应用 36](#_Toc517267172)

[5.4.1 用户广告精准推荐 36](#_Toc517267173)

[5.4.2 社交娱乐 37](#_Toc517267174)

[5.5 本章小结 38](#_Toc517267175)

[结论 40](#_Toc517267176)

[参考文献 41](#_Toc517267177)

[攻读学士学位期间发表的论文和取得的科研成果 43](#_Toc517267178)

[致谢 44](#_Toc517267179)

# 第1章 绪论

## 1.1课题背景

中国数值水池虚拟实验系统时利用先进的水动力学理论模型和精细数值算法，融合了专家的指挥，系统化编制的高效计算软件，经物理实验验证后，结合先进的计算机和互联网条件，为全球的行业用户提供在不同的海洋环境中，进行船舶与海洋结构流体动力响应过程的虚拟实验，满足船舶与海洋工程领域研究，设计及工程应用的要求。随着众多的虚拟实验集成以及用户数的激增，单节点服务器在进行虚拟实验数值计算时会产生性能瓶颈，一台服务器已经不能满足应用的需求，而需要更多的服务器集群来支撑庞大的计算量。开发数值水池作业调度系统将优化求解器计算任务在服务器组之间的分配，消除了服务器之间的负载不均衡，从而提高主系统的反应速度与总体性能。

## 1.2课题目的和意义

数值水池虚拟实验网站创立初期采用的单机节点服务器虽然使用简单，配置成本低，数据共享程度高，一致性好，但是由于随着虚拟试验集成的范围越来越广，用户数量逐渐上升，求解计算任务激增，数据量呈现海量增长，将所有的业务项目如（web服务器，文件服务器，数据库）全部部署服务到一台服务器上，所有的请求业务都由这台服务器处理。显然，当业务增长到一定程度的时候，服务器的硬件会无法满足业务需求。自然而然不可能支撑庞大的用户流量和海量数据，必须演变其自身架构。需要对网站做出如下调整：独立部署，避免不同的系统之间相互争夺共享资源（比如CPU、内存、磁盘等）；Web服务器集群，实现可伸缩性；部署分布式缓存系统，使查询操作尽可能在缓存命中；数据库实施读写分离，实现HA（High Availability）架构。

本课题旨在设计与实现一个面向集群的作业调度管理系统平台，在千万量级的计算任务中，由于存在大量求解任务请求并发访问，需要降低用户请求响应延时，需要合理设计调度策略，解决集群服务器组之间的计算任务合理分配，并对计算任务进行管理。同时完成各类虚拟实验求解器的按需部署及自动同步机制，给出个节点求解器计算状态。实现任务的集群分发，处理异常，保证可靠性。以及维护各类信息的日志，用于后续查询及决策。

在实际应用中，系统管理员可以通过管理页面查询计算集群节点资源状态，并查看每个计算节点的计算任务分配与求解器运行状态，可通过日志系统对求解任务进行回溯，远程一键kill不合理的计算任务，方便数值水池开发人员对虚拟实验集成及网站开发的后续决策。

## 1.3 国内外研究现状

在集群作业调度研究方面，主要分为定时分片类作业调度系统和DAG工作流类作业调度系统。

（1）定时分片类作业调度系统

第一，定时分片类系统的方向，重点定位于任务的分片执行场景，这类系统的代表包括：TBSchedule，SchedulerX，Elastic-job, Saturn。这种功能定位的作业调度系统，其最早的需要来源和出发点往往是做一个分布式的Crontab／Quartz。一开始各个业务方八仙过海，自己玩自己的单机定时任务，然后，随着业务的增长，各种定时任务越来越多，分散管理的代价越来越高。再加上有些业务随着数据量的增长，为了提高运行效率，也需要以分布式的方式在多台机器上并发执行。这时候，分布式分片调度系统也就孕育而生了。这类系统的实际应用场景，往往和日常维护工作或需要定时执行的业务逻辑有一定关联。比如需要定时批量清理一批机器的磁盘空间，需要定时生成一批商品清单，需要定时批量对一批数据建立索引，需要定时对一批用户发送推送通知等等。这类系统的核心目标基本上就是两点：对作业分片逻辑的支持：将一个大的任务拆成多个小任务分配到不同的服务器上执行， 难点在于要做到不漏，不重，保证负载平衡，节点崩溃时自动进行任务迁移等；高可用的精确定时触发要求：因为往往涉及到实际业务流程的及时性和准确性，所以通常需要保证任务触发的强实时和可靠性。所以，负载均衡，弹性扩容，状态同步和失效转移通常是这类调度系统在架构设计时重点考虑的特性。第二，从接入方案和流程上来说，因为要支持分片逻辑，要支持失效转移等，这类调度系统，对所调度的任务通常都是有侵入性要求的。太注重算法的时间复杂度，分析千量级节点的图谱时需要消耗过多的时间。第三，从触发实现逻辑的角度来说，为了在海量任务的情况下，保证严格精确定时触发，这类调度系统有一大半，其定时触发逻辑，实际上是由执行节点自身在本地触发的，也就是说要求作业或守护进程处于运行状态，向服务端注册作业，服务端分配分片信息和定时逻辑给到客户端，但定时的触发，是由客户端库函数封装的如Quartz等定时逻辑来实际执行触发的。这样做的首要目的当然是为了保证触发的精度和效率，降低服务端负载，此外如果服务端短时间内挂掉，只要作业配置保持不变，作业还是能够在客户端正常触发的。也有些系统，比如SchedulerX，是采用服务端触发逻辑的。这对服务端的要求就高了很多，因为这时候，服务端不光要协调分片逻辑，还要维护触发队列。所以服务端触发的系统，首先要保证服务端的高可用，其次

还要保障性能，因此，通常都是采用集群方案

1. DAG工作流类作业调度系统

这一类系统的方向，重点定位于任务的调度依赖关系的正确处理，分片执行的逻辑通常不是系统关注的核心，或者不是系统核心流程的关键组成部分，如果某些任务真的关注分片逻辑，往往交给后端集群（比如MR任务自带分片能力）或者具体类型的任务执行后端去实现。DAG工作流类调度系统所服务的往往是作业繁多，作业之间的流程依赖比较复杂的场景，比如大数据开发平台的离线数仓报表处理业务，从数据采集，清洗，到各个层级的报表的汇总运算，到最后数据导出到外部业务系统，一个完整的业务流程，可能涉及到成百上千个相互交叉依赖关联的作业。所以DAG工作流类调度系统关注的重点，通常会包括：第一：足够丰富和灵活的依赖触发机制：比如时间触发任务，依赖触发任务，混合触发任务；而依赖触发自身，可能还要考虑，多亲依赖，长短周期依赖（比如小时任务依赖天任务，或者反过来），依赖范围判定（比如所依赖任务最后一次成功就可以触发下游，还是过去一个星期的所有任务都成功才可以触发下游），自身历史任务依赖，串并行触发机制等等。第二：作业的计划，变更和执行流水的管理和同步。第三：任务的优先级管理，业务隔离，权限管理等，在定时分片类调度系统中，通常情况下，具体执行端的业务的隔离很多情况下是天然的，注册了特定业务的节点才会去执行特定的任务。然后，加上业务链路一般都比较短，以及强实时性要求，所以对优先级的管理通常要求也不高，基本靠资源隔离来实现资源的可用，不太存在竞争资源的问题，权限管理也同理；而在DAG工作流类调度系统中，往往一大批作业共享资源执行，所以优先级，负载隔离，和权限管控的问题也就突显出来。第三：各种特殊流程的处理，比如暂停任务，重刷历史数据，人工标注失败／成功，临时任务和周期任务的协同等等；这类需求，本质上也是因为业务流程的复杂性带来的，比如业务逻辑变更啦，脚本写错，上游数据有问题，下游系统挂掉等等，而业务之间的网状关联性，导致处理问题时需要考虑的因素很多，也就要求处理的手段要足够灵活强大。第四：完备的监控报警通知机制，最简单的比如，任务失败报警，超时报警，再进一步，流量负载监控，业务进度监控和预测，如果做的再完善一点，还可以包括业务健康度监控分析，性能优化建议和问题诊断专家系统等。

## 1.4 主要研究内容

本课题旨在完成一个面向集群的作业调度中心控制系统的设计与实现，将网站架构从单机节点过渡到集群节点，并保证数值水池虚拟试验计算任务在各集群服务器组中进行合理分配和调度。

作业调度平台中心控制系统是一种反向代理机制。中心控制节点提供反向代理服务，根据客户端计算求解任务的请求，将计算任务和求解器分配到与其关系的一组或多组后端[服务器](https://baike.baidu.com/item/%E6%9C%8D%E5%8A%A1%E5%99%A8)，实现各种虚拟实验服务器簇的求解分离。中心控制节点会监控和管理计算任务，以实现后端虚拟实验服务器簇与中心控制节点反向代理服务器之间的求解器同步以及资源状态的收集。

实验与应用是作业调度平台中心控制系统有效性的验证。分别使用不同的用户以及虚拟实验计算任务量和服务器集群分别验证计算任务的合理调度和分发以及求解器自动同步机制，通过应用体现其实际价值。

## 1.5 论文结构安排

本篇论文主要分为5章，从面向集群作业调度平台的背景、研究的意义以及研究现状，到作业调度平台中心控制系统的设计与实现，以及多种调度策略算法和求解器同步机制的验证与应用等方面进行论述。

第1章主要论述本课题的背景，研究的意义目的、国内外的对本课题的研究现状以及本课题的主要内容方面做大体的说明。

第2章主要论述求解器按需部署及自动同步机制的实现，简要介绍求解器部署的需求及自动同步机制算法的设计。。

第3章是重点论述集群各计算节点状态收集的实现

第4章重点论述多种调度策略算法的设计与实现，先论述选择合适的调度策略，然后将计算任务进行合理分配。

第5章是实验与应用，验证多种调度策略算法和求解器同步机制

最后是结论、致谢和参考文献。

# 第2章 求解器的按需部署及自动同步机制的设计与实现

## 2.1 系统总体设计

本系统采用Jetty作为内置服务器，基于RESTFUL软件设计架构。众所周知，URL是web应用服务的路径，用户通过浏览器发送过来的任何请求都会被发送到一个指定的URL地址里，然后被响应。首先创建cn.edu.hrbeu.theweb.server.WebServer类，这个类的功能是使用jetty作为嵌入式服务器，把jetty部署到web应用中。因为Jetty可以在java应用中像其他POJO一样被实例化，换句话说，以嵌入式的模式运行Jetty是将http模块放入应用程序中，而非部署程序到HTTP服务器。首先创建一个Server实例，在私有方法configureServer()里添加/配置Connectors，添加/配置Handlers / Contexts 以及 Servlets。然后在构造函数WebServer(int serverPort)中传入serverPort，以serverPort为端口启动和监听Jetty，这样就首先完成了Jetty的嵌入式实现。

Jetty服务器成功后读取etc/settings.json配置文件，该配置文件以json的格式配置了MongoDB数据库地址(DATABASE\_ADDRESS)，数据库名称(DATABASE\_NAME),虚拟实验数据存放的根路径(PROJECT\_BASE\_PATH)，上传文件临时存放路径(UPLOAD\_TEMP\_PATH)，求解器存放根路径(SOLVER\_BASE\_PATH)，身份标识(Role).

cn.edu.hrbeu.theweb.server.HTTP类有同样的静态字段，并且附带有初始化的默认值。cn.edu.hrbeu.theweb.server.HTTP类图如图2.1所示。



图 2.1 cn.edu.hrbeu.theweb.server.HTTP类图

在用户登陆系统验证身份成功后，通过LeaderService.init()启动Leader服务。系统的初始化流程图如下：



图 2.2 系统初始化流程图

（1）读取配置文件

读取etc/settings.json文件，若配置文件为空，则退出并结束。

（2）设置HTTP类的各字段值

根据配置文件中的键值对(key:value)设置相应的HTTP类中各字段值，如果某个字段值为空，则设置为相应的HTTP类字段缺省值。

（3）HTTP.ROLE字段验证

HTTP.ROLE字段的缺省值为"Leader/Agent"，如果

（4）连接用户信息数据库

## 2.2 路由机制与视图渲染

本系统的路由机制就是使处理数据的函数与请求的URL建立映射关系。使请求到来之后，根据cn.edu.hrbeu.theweb.server/目录下的RootResource，StaticResource，SolverFileResource，FileResource，LeaderFeedbackResource，LeaderServiceResource类里的正则表达式及注解条目，去查找与请求相对应的处理方法，从而返回给客户端http页面数据。相应的操作方法都加有@path,@post,@get,@produce注解。@Path注释的值是表示在Java类将被承载的相对URI的路径：例如，/ HelloWorld的,还可以在URI中嵌入变量以生成URI路径模板。例如，可以询问用户的名称，并将其作为URI中的变量传递给应用程序： / helloworld / {username}；@POST注解是请求方法指示符，并对应于类似命名的HTTP方法。使用此请求方法指示符注释的Java方法将处理HTTP POST请求。资源的行为由资源响应的HTTP方法确定；@Produces注释用于指定MIME媒体类型表示的资源可以产生和发送回客户端：例如，“text / plain的”；@Consumes注释用于指定MIME媒体类型表示资源可以消耗这是由客户端发送的；@GET注解是请求方法指示符，并对应于类似命名的HTTP方法，使用此请求方法指示符注释的Java方法将处理HTTP GET请求，资源的行为由资源响应的HTTP方法确定；@PathParam 注释是一个类型参数，可以提取的资源类的使用，URI路径参数从请求URI中提取，参数名称对应于@Path类级别注释中指定的URI路径模板变量名称。@path注解可以加入正则表达式用来匹配请求路径映射。例如：@Path("solver/{id: [a-zA-Z\_0-9]\*}/file/download/{fileLink: .\*}")是用来匹配求解器文件下载请求映射的函数，@Path("solver/{id: [a-zA-Z\_0-9]\*}/file/delete/{fileLink: .\*}")是用来匹配删除求解器文件请求映射的函数，@Path("/css/{subResources:.\*}")是用来匹配静态资源目录下css子目录下的层叠样式表请求映射的函数。系统总的路由机制图与视图渲染如下所示：



图 2.3 路由机制图

首先http请求的根目录 / 可以映射到所有的Resource类（RootResource， StaticResource， SolverFileResource， FileResource，LeaderFeedbackResource，LeaderServiceResource）。

1. RootResource类相当于一个一级路由，主要处理用户信息操作请求，比如用户登陆，用户注销，用户注册。以及doApi(String tt, String op, Document input, LoginedUser lu)函数用来处理系统总的api请求，准发到二级分路由请求进行处理。RootResource类路由机制图如下所示：

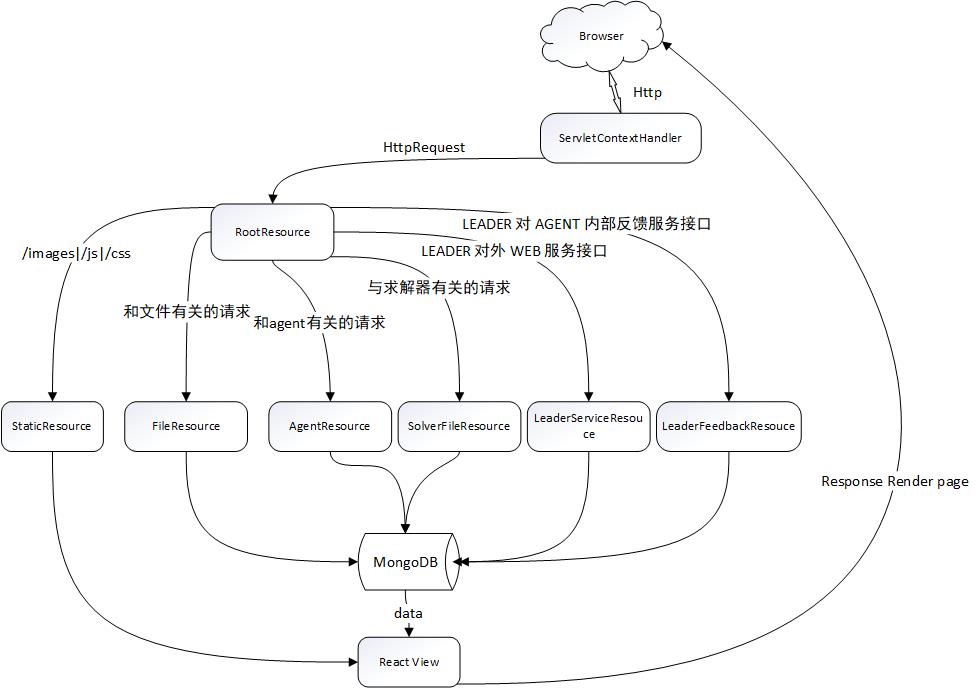


图 2.4 RootResource类路由机制与系统视图渲染图

（2）StaticResource类主要用来处理静态资源请求映射的URL.如javascript脚本，image图片以及css层叠样式表。

（3）SolverFileResource类主要用来处理与求解器文件有关的请求操作，如求解器文件的上传，下载，删除，打开，以及求解器文件属性查询。

（4）FileResource类主要用来处理与文件相关的请求操作。如文件的上传，下载，删除，以及文件属性查询。

（5）LeaderFeedbackResource类主要用来处理中心控制节点LEADER 对 AGENT 内部反馈服务接口，处理AGENT api请求和 AGENT文件同步。

## 2.3 求解器按需部署：

SolverFileResouce类专门用来处理与solver求解器相关的请求，用户请求通过根路径/可以直接跳转到该类。

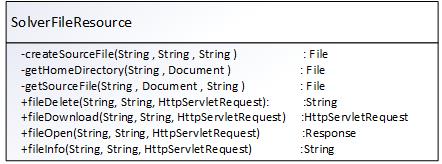


图 2.5 SolverFileResource类图

createSourceFile函数用于创建求解器源文件；getHomeDirectory函数用于获取文件根路径，getSourceFile用于获取求解器源文件；fileDelete函数用于删除求解器文件；fileDownload函数用于处理求解器文件下载请求；fileOpen函数用于处理打开求解器文件请求；fileInfo函数用于获取求解器文件属性。

Solvers类里包含了对求解器的基本操作，包括getRelateAgentList(获取安装该求解器相关的计算节点列表)；solverUpdate（求解器更新）；solveDistribute（求解器发布）；solverFileAppoint（求解器版本文件指定）；solverDelete（求解器删除）；run(执行求解器)；getCommandLine（获取求解器执行参数）；solverFileGet（获取求解器文件）；

listSolverFile（获取求解器列表）。Solvers类还包含了二级路由，从RootResource类的doApi函数一级路由经由与Solver有关的操作可以跳转到该二级路由，该二级路由里包含了如下Solvers类图中所示的所有私有方法，可以过滤请求并跳转到相应的方法进行对Solvers的各种操作。



图 2.6 Solvers类图

管理员用户可以根据前端界面对求解器属性进行修改，例如新建/删除求解器。新建求解器后可以编辑求解器属性，如求解器名称，求解器描述，选择发布模式（包括集群发布和仅本地发布），求解器适合运行的操作系统，版本号，加载器(java, python)等，启动程序，启动参数，排队方式(标准队列，快速队列，不排队)。集群发布表示将该类求解器任务在所有求解器节点上进行发布，根据后续的调度策略选择最优计算节点进行计算；本地发布表示将该类求解器任务放在本地中心控制节点进行求解计算而不进行集群发布，减少了根据调度策略进行选择最优节点进行计算的步骤。求解器的排队方式有三种：标准队列表示求解器在集群发布时进行所有求解任务入队列操作，根据相应的调度策略进行选择最优计算节点的过程。选择快速队列则表示该类求解器任务具有优先级，例如根据以往每类求解器运行时间长短，可以将能够快速得到计算结果的求解器任务选择快速队列，目前该系统支持的快速队列节点为本地节点，省去了调度策略步骤，可以更快的响应用户实验结果的请求。不排队方式也是放在本地节点进行计算求解的，可以对应具有特殊需求的一类求解器任务。

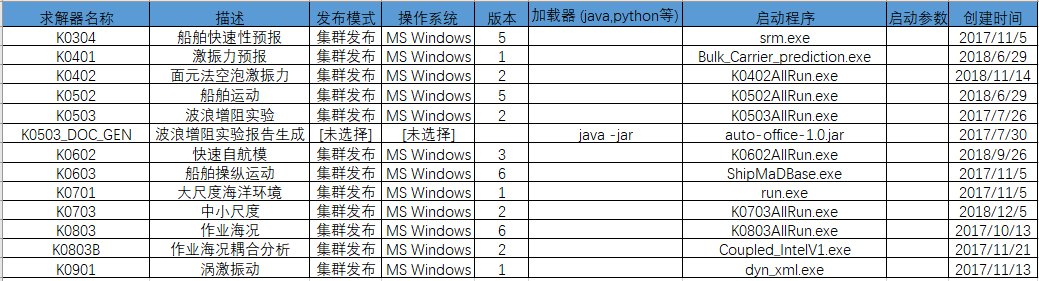


图 2.7求解器按需部署属性图

求解器按需部署属性对应MongoDB的数据库字段如下：

{

"\_id" : "59e041ddf589d45478a2d960",

"name" : "K0803",

"label" : "作业海况",

"executor" : "K0803AllRun.exe",

"user" : "5978ab95f589d41b9cbab06d",

"createTime" : NumberLong(1507869149575),

"file" : "K0803A\_20190408.zip",

"updateTime" : NumberLong(1554697672166),

"os" : "windows",

"mode" : 0,

"queue" : 0,

"version" : 4,

"loader" : "",

"params" : ""

}

在前端界面中，用户编辑好每一个求解器属性后可进入求解器信息详细界面，左侧显示求解器属性和执行参数等已编辑信息，右侧两个表格分别显示求解器执行文件和该类求解器对应的计算任务列表。求解器文件支持上传和刷新操作，目前求解器上传文件仅支持zip格式，求解器执行文件列表将显示所有版本求解器压缩文件，对每个版本的求解器执行文件有两个操作：指定和删除。如果一个版本的求解器执行文件被指定后，后续该类求解计算任务都会根据此版本的求解器进行求解计算。管理员每上传一个求解器执行文件，则新求解器执行文件的版本号加1，对应数据库相应字段version增1。

## 2.4 求解器自动同步机制：

集群中每个计算节点详细界面可以显示该节点已经安装的求解器列表，管理员可以选择在该列表中添加求解器，点击添加求解器后，会显示控制节点求解器中央仓库中所有最新版本的求解器，管理员可以根据复选框添加相应的求解器。之后求解器会根据相应的自动同步机制进行同步。例如计算节点E206，IP地址为192.168.1.216的节点上已经安装求解器列表如下：

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **求解器名称** | **版本** | **描述** | **操作系统** | **执行状态** |
| [K0304](http://202.118.184.201:10001/#/solvers/59fee826f589d42b7864def2/) | 5 | 船舶快速性预报 | MS Windows | 同步成功 |
| [K0401](http://202.118.184.201:10001/#/solvers/5b35f2bdf589d4116459bd5e/) | 1 | 激振力预报 | MS Windows | 同步成功 |
| [K0602](http://202.118.184.201:10001/#/solvers/5bab4cabf589d4116459bd5f/) | 3 | 快速自航模 | MS Windows | 同步成功 |
| [K0603](http://202.118.184.201:10001/#/solvers/59feeb4cf589d42b7864def3/) | 6 | 船舶操纵运动 | MS Windows | 同步成功 |
| [K0701](http://202.118.184.201:10001/#/solvers/59ff00b7f589d42b7864def4/) | 1 | 大尺度海洋环境 | MS Windows | 同步成功 |
| [K0901](http://202.118.184.201:10001/#/solvers/5a097e9bf589d41c54cfff68/) | 1 | 涡激振动 | MS Windows | 同步成功 |

表2-1 E206计算节点已安装求解器列表

如果管理员需要在某个计算节点上添加相应的求解器，当复选未安装求解器列表中的求解器并点击确认时，相应的计算节点会向中心控制节点发出特定的http下载请求，从控制节点求解器中央仓库目录下载选中的求解器到计算节点本地求解器目录。接着在该计算节点相关联的求解器列表就可以看到刚刚安装的求解器了。求解器的同步以及启动状态在DEF类里定义，如下表所示：

|  |  |
| --- | --- |
| 状态 | 含义 |
| **\_SOLVER\_SYN\_STATUS\_CAN\_NOT\_SYN = 0** | 无法同步 |
| **\_SOLVER\_SYN\_STATUS\_WAITING = 1** | 等待启动 |
| **\_SOLVER\_SYN\_STATUS\_STARTED = 2** | 启动 |
| **\_SOLVER\_SYN\_STATUS\_FINISHED = 3** | 已同步成功 |
| **\_SOLVER\_SYN\_LEADER\_FILE\_LOST = -1** | 无法连接上级 |
| **\_SOLVER\_SYN\_STATUS\_FILE\_CAN\_NOT\_DOWNLOAD = -2** | 上级无法下载文件 |
| **\_SOLVER\_SYN\_STATUS\_FILE\_CAN\_NOT\_UNZIP = -3** | 文件无法解压 |

表2-2 求解器同步及启动状态

## 2.5 本章小结

本章论述作业调度平台中心控制系统的路由设计与视图渲染，其中路由设计分为一级路由与二级路由，视图渲染是通过相应的路由并与数据库交互后用React组件渲染相应的结果数据前端界面进行展示。求解器的按需部署及自动同步机制，求解器的按需部署包括求解器属性的编辑，选择求解器运行的操作系统，发布模式，排队方式等。求解器的自动同步机制是当管理员在某个计算节点添加未安装求解器时，会自动向中心控制节点请求同步该求解器最新版本的可执行文件。

# 第3章 管理收集计算节点资源状态与系统日志信息的维护

## 3.1 总体实现

作业调度平台中心控制节点要实现对集群计算节点的管理，以及集群任务的分发就必须清楚的知道每个计算节点的资源状态。在此前提下，当新的求解任务请求触发时，中心控制节点会根据当前的集群状态以及求解器属性配置进行求解任务调度策略的选择与执行。集群状态最重要的指标包括每个计算节点的cpu利用率，内存利用率以及当前节点所执行的作业数量等。

以计算节点E208为例，在agents表结构中定义了如下字段：

{

"\_id" : "5c9099d5f589d423d064a4cf",

"name" : "E208",

"ip" : "192.168.1.218",

"os" : "windows",

"port" : 10001,

"path" : "Y:/TANK/projects",

"maxLoads" : 4,

"updateTime" : NumberLong(1554791827884),

"status" : 1,

"loads" : 3,

"solvers" : [

"59fee826f589d42b7864def2",

"5b35f2bdf589d4116459bd5e",

"5beb788ff589d4312cc24eaf",

"5b35e93af589d4116459bd5d",

"5978ac60f589d41b9cbab06e",

"5bab4cabf589d4116459bd5f",

"59feeb4cf589d42b7864def3",

"59ff00b7f589d42b7864def4",

"5c0739cff589d4312cc24eb0",

"59e041ddf589d45478a2d960",

"5a1391ddf589d417b0735bb9",

"5a097e9bf589d41c54cfff68"

],

"suspend" : false

}

\_id表示计算节点编号，name节点名称，ip地址，ip端口，os表示操作系统，path表示该计算节点存储求解任务计算结果的根路径。maxloads表示该计算节点设置的最大负载即缺省并行计算的最大作业量，updateTime代表该节点资源状态更新的时间戳。status代表节点的在线状态，在cn.edu.hrbeu.theweb.logic.DEF类里设置了三个枚举值；loads表示当前节点负载即正在运行的作业数量。solvers字段是一个json数组，为该计算节点已经安装的所有求解器的\_id集合，以下时计算节点status字段属性。

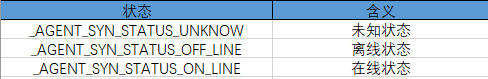


图 3.1: 计算节点status状态图

# 第4章 集群任务分发与多种任务调度策略的实现

## 4.1 总体实现

当服务器停留在单机架构时，随着业务的增加，访问量的提高和数据的不断堆积，系统会变得越来越慢，硬件设备的更新换代是一个必不可少的工作。在达到同等性能的条件下，采用计算机集群比采用同等能力的计算机所花的代价要小很多，能降低成本；采用传统服务器的用户如果需要大幅度扩展系统的能力，就必须购买昂贵的服务器，将单机架构过渡到集群技术，只需要将新的服务器加入集群即可，相比而言提高了系统的扩展性。作业调度平台中心控制系统相当于一个代理负载均衡器，整个系统则为负载均衡集群。每个计算节点都可以承担一定的处理负载，并且可以由中心控制节点实现处理负载在计算节点之间的动态分配。作业调度平台中心控制节点应用服务要求分布式集群检查每个计算节点检查当前的负载，并确定哪些节点可以接受新的作业，以实现集群的任务分发。

LeaderTaskSender类是专门用来处理集群任务分发。

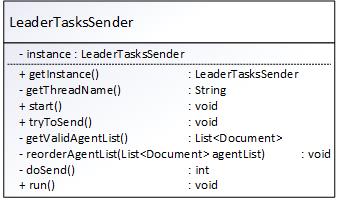


图 3.3: LeaderTaskSender类图

LeaderTaskSender继承自java.lang.Thread类，实现方式使用了单例设计模式，LeaderTaskSender类只有一个实例，该类由getInstance方法自行创建该实例，向整个系统公开这个实例接口。

LeaderTaskSender的启动在创建该对象实例instance时进行。

|  |
| --- |
| public static LeaderTasksSender getInstance() {  if (instance == null) {  instance = new LeaderTasksSender();  instance.start();  }  return instance;  } |

表4-1: LeaderTaskSender的实例启动

Thread的run方法是不抛出任何检查型异常(checked exception)的,但是它自身却可能因为一个异常而被终止，导致这个线程的终结。最麻烦的是，在线程中抛出的异常即使使用try...catch也无法截获，因此可能导致一些问题出现，比如异常的时候无法回收一些系统资源，或者没有关闭当前的连接等等。 LeaderTaskSender在初始化静态块中为所有线程加入了异常处理，设置了线程异常的打印和处理信息。

|  |
| --- |
| static {  Thread.setDefaultUncaughtExceptionHandler(new Thread.UncaughtExceptionHandler() {  @Override  public void uncaughtException(Thread t, Throwable e) {  if (t instanceof LeaderTasksSender) {  System.err.println(new SimpleDateFormat("HH:mm:ss\_SSS> ").format(new Date()) +  "LoggerThread " + ((LeaderTasksSender) t).getThreadName() +  " died! Because " + E.formatThrowable(e));  System.exit(9);  } else {  System.err.println(new SimpleDateFormat("HH:mm:ss\_SSS> ").format(new Date()) +  "Thread " + t.getName() + " died! Because " + E.formatThrowable(e));  }  }  });  } |

表4-2: LeaderTaskSender类异常处理

在进行集群任务分发时，必要的基础条件时获取当前有效的计算节点资源状态。getValidAgentList方法首先构建求解器映射表，求解器映射表是一个Map<String, Document>数据结构。其次获取活跃的计算节点列表，接着使用DB.\_list(Document where, Document order, Document select)非分页方式获取获取所有在线计算列表，此处的参数where是查询条件，值为new Document(DEF.\_STATUS, DEF.\_AGENT\_SYN\_STATUS\_ON\_LINE), 即要查询的条件为{“status” : 1}的所有计算节点，参数order值为NULL表示不排序，select的参数值表示查询结果投影的字段，此处为DEF.\_LOADS, DEF.\_MAX\_LOADS，DEF.\_SOLVERS， DEF.\_PATH。将这些查询信息添加到agenList里。

接着遍历该agentList, 首先获取agentID, loads, maxLoads字段，比较计算节点当前负载量loads和最大设置负载量maxLoads。如果当前作业负载量小于最大负载量，首先获取该计算节点的所有求解器ID列表solverIds, 判断该agent是否配置了solver, 如果配置了solver则从求解器映射表里获取该solver的相关属性solverItem, 并将其添加到solverList里。如果solverList不为空，则将其添加到activeAgentlist中。如果activeAgentList中活跃的计算节点数大于2，则根据每个节点的剩余负载量（最大负载量-当前负载量）对activeAgenList活跃计算节点列表进行一个排序，返回活跃计算节点列表剩余负载量的一个升序版本。

在DEF类中对Task求解任务的每一个状态进行标识。

|  |  |
| --- | --- |
| 状态 | 含义 |
| **\_TASK\_RUN\_STATUS\_CREATE = 0** | 未启动 |
| **\_TASK\_RUN\_STATUS\_PENDING = 1** | 等待启动 |
| **\_TASK\_RUN\_STATUS\_HAS\_= 2** | 已启动 |
| **\_TASK\_RUN\_STATUS\_IS\_RUNNING = 3** | 执行中 |
| **\_TASK\_RUN\_STATUS\_IS\_KILLED = 4** | 强制停止 |
| **\_TASK\_RUN\_STATUS\_HAS\_FINISHED = 5** | 执行结束 |
| **\_TASK\_RUN\_STATUS\_RECORD\_LOST = -1** | 任务不存在 |
| **\_TASK\_RUN\_STATUS\_FAILED\_STARTED = -2** | 启动失败 |
| **\_TASK\_RUN\_STATUS\_EXCEPTION\_STOP = -3** | 意外停止 |
| **\_TASK\_RUN\_STATUS\_IS\_CANCLED = -4** | 取消 |
| **\_TASK\_RUN\_STATUS\_SOLVER\_NOT\_EXIST = -5** | 求解器不存在 |
| **\_TASK\_RUN\_STATUS\_SOLVER\_NOT\_DIST = -6** | 求解器未发布 |
| **\_TASK\_RUN\_STATUS\_AGENT\_INTERNAL\_ERROR = -7** | 计算节点内部错误(重启) |
| **\_TASK\_RUN\_STATUS\_TASK\_CANNOT\_CREATE = -8** | 任务无法创建 |

表4-24: Task属性标识

doSend()方法是处理集群任务分发的函数。

第一步：由getUndistTaskList()方法先获取未分配的task列表存储在taskList中。并设置计数器count,初始值设置为0,代表将要分发的任务个数。

|  |
| --- |
| List<Document> taskList = this.getUndistTaskList();  if (taskList == null || taskList.isEmpty()) {  return 0;  }  int count = 0; |

表4-3: 获取未分配的Tak列表

getUndistTaskList()方法原型如下：首先设置查询条件为undistTaskWhere, 即以任务状态为\_TASK\_RUN\_STATUS\_PENDING = 1 等待启动的任务为过滤条件。排序方式为按照\_ID降序。在获取未分配Task列表时对线程对象进行加锁，保证数据的可靠性。

|  |
| --- |
| private final Document undistTaskWhere = new Document(DEF.\_STATUS, DEF.\_TASK\_RUN\_STATUS\_PENDING)  .append(DEF.\_VERSION, DEF.\_NOT\_EXISTS)  .append(DEF.\_AGENT, DEF.\_NOT\_EXISTS);  private final Document undistTaskOrder = Docat.\_\_order\_down(DEF.\_ID);  private final Document undistTaskSelect = null;  private List<Document> undistTaskList;  private boolean reload = true;  private List<Document> getUndistTaskList() {  synchronized (this) {  if (this.reload) {  this.reload = false;  undistTaskList = DB.task.\_\_list(undistTaskWhere, // 可分配但未分配  undistTaskOrder, undistTaskSelect);  }  return undistTaskList;  }  } |

表4-4: getUndiskTaskList方法原型

第二步：构建求解器映射表。首先根据undistTaskList未分配Task列表，获取所有相关task对应求解器Solver的编号\_ID.存储在以List<string>为数据结构的solverIds变量中。其次以DEF.\_VERSION, DEF.\_LOADER, DEF.\_EXECUTOR字段构建求解器映射表solverMap。

|  |
| --- |
| List<String> solverIds = Docat.\_\_get\_value\_list(undistTaskList, DEF.\_SOLVER, true);  Map<String, Document> solverMap = DB.solver.\_\_map(new Document(DEF.\_ID, DEF.\_IN(solverIds)), null,  Docat.\_\_select(DEF.\_VERSION,  DEF.\_LOADER,  DEF.\_EXECUTOR)); |

表4-5: 构建求解器映射表

第三步：填充求解器映射表，找到每一个求解器可以分配的节点列表。

1. 首先先找到solver在每个agent上的分配情况。项目中的dist数据库表是专门用来连接solver和agent的。比如名称为E209，\_ID为5c921bd7f589d423d0df9046的计算节点可以映射一个或多个求解器。dist数据库表的结构如下：

|  |
| --- |
| {  "\_id" : "5c921bd7f589d423d0df9046",  "agent" : "5c90d3c7f589d41918717c08",  "solver" : "59fee826f589d42b7864def2",  "version" : 4,  "status" : 3,  "createTime" : NumberLong(1553079255314)  } |

表4-6: dist数据表结构

Status为3,表示\_ID为59fee826f589d42b7864def2的solver求解器在该节点已经同步成功。

1. 获取初级比较指标loads。在distinct数据表中获取agentID字段的不重复值：List agentIds = DB.dist.\_\_distinct(DEF.\_AGENT, where, null);如果agentIds不为空，则从agent表中获取以DEF.\_LOADS, DEF.MAX\_LOADS, DEF.PATH为集合的agent列表agentList。如果agentList不为空，则遍历agentList。从agentList取出每一个成员项agent,获取相应的agentID, loads, maxLoads。

|  |
| --- |
| if (agentIds != null && !agentIds.isEmpty()) {  List<Document> agentList = DB.agent.\_\_list(  new Document(DEF.\_ID, DEF.\_IN(agentIds)).  append(DEF.\_SUSPEND, false),  null,  Docat.\_\_select(DEF.\_LOADS, DEF.\_MAX\_LOADS, DEF.\_PATH));  if (agentList != null && !agentList.isEmpty()) {  for (int j = agentList.size() - 1; j >= 0; j--) {  Document agentItem = agentList.get(j);  String agentId = Docat.getString(agentItem, DEF.\_ID);  int loads = Docat.getInteger(agentItem, DEF.\_LOADS, 0);  int maxLoads = Docat.getInteger(agentItem, DEF.\_MAX\_LOADS, 0);  if (loads >= maxLoads) {  agentList.remove(j);  }  }  } |

表4-7: 获取初级比较指标loads

1. 获取准确比较指标loads。如果当前遍历agent节点的作业负载loads大于最大负载阈值maxLoads,则将该agent从agentList中移除。否则获取准确负载后再比较一次，从Task数据表中统计该agent的所有作业负载，包括正在运行，等待启动和已经启动的任务。如果获取的准确负载loads大于最大负载阈值maxloads, 则从agentList中移除该节点。否则，计算该节点剩余负载，即leftloads = maxLoads – loads。并将该字段项写入agent数据表中。

|  |
| --- |
| else {  // loads只是初级指标，获取准确负载后再比较一次  loads = DB.task.\_\_count(new Document(DEF.\_AGENT, agentId).  append(DEF.\_STATUS, DEF.\_IN(DEF.\_TASK\_RUN\_STATUS\_IS\_RUNNING,  DEF.\_TASK\_RUN\_STATUS\_PENDING,  DEF.\_TASK\_RUN\_STATUS\_HAS\_STARTED)));  if (loads >= maxLoads) {  agentList.remove(j);  } else {  int leftLoads = maxLoads - loads;  canLoads += leftLoads;  agentItem.append(DEF.\_LEFT\_LOADS, leftLoads);  }  } |

表4-8: 获取准确比较指标loads

1. 将可用的agent列表写进solver表字段中，并且将可用的agent按照leftLoads降序排列，目的是将任务分配到最空闲的agent中。

|  |
| --- |
| if (size > 0) {  // 将可用的agent放到solver中  solver.append(DEF.\_AGENTS, agentList);  // 如果需要按照leftLoads降序排列，目的是将任务分配到最空闲的agent中  this.reorderAgentList(agentList);  } |

表4-9: 对agent按照leftLoads降序排列

reorderAgentList是对agent按照leftLoads剩余负载进行排序的函数，其原型如下

|  |
| --- |
| private void reorderAgentList(List<Document> agentList) {  if (agentList.size() >= 2) {  agentList.sort(new Comparator() {  @Override  public int compare(Object o1, Object o2) {  int l1 = Docat.getInteger((Document) o1, DEF.\_LEFT\_LOADS);  int l2 = Docat.getInteger((Document) o2, DEF.\_LEFT\_LOADS);  if (l1 > l2) {  return -1;  } else if (l1 == l2) {  return 0;  } else {  return 1;  }  }  });  }  } |

第四步：将Task按顺序进行分配。

1. 从task数据表中获取任务状态为\_TASK\_RUN\_STATUS\_PENDING即等待启动的任务taskIntem。如果taskItem为空，则该task在分配前已经被取消，打印日志信息并且从taskList中移除该task。

|  |
| --- |
| for (int i = taskList.size() - 1; i >= 0; i--) {  Document taskItem = DB.task.\_\_get(  New Document(DEF.\_STATUS, DEF.\_TASK\_RUN\_STATUS\_PENDING).  append(DEF.\_ID, taskList.get(i).get(DEF.\_ID)),  null,  null  );  if (taskItem == null) {  M.trace("TASK分配前已被取消 => ", Docat.\_\_doc\_to\_json(taskList.get(i)));  taskList.remove(i);  continue;  } |

表4-10: 获取等待启动的任务Task

1. 从task数据表中的每一项taskItem取得taskId, solverId。然后根据该solverId从求解器映射表solverMap中获取该solver的映射信息solverItem。如果solverItem为空，则更新task表status字段为\_TASK\_RUN\_STATUS\_SOLVER\_NOT\_EXIST即求解器不存在，并追加日志信息字段DEF.\_MESSAGE为“求解器已经删除”, 添加相应的时间戳。

|  |
| --- |
| String taskId = Docat.getString(taskItem, DEF.\_ID);  String solverId = Docat.getString(taskItem, DEF.\_SOLVER);  Document solverItem = solverMap.get(solverId);  if (solverItem == null) {  DB.task.\_\_update\_by\_id(taskId,  new Document(DEF.\_STATUS,  DEF.\_TASK\_RUN\_STATUS\_SOLVER\_NOT\_EXIST)  .append(DEF.\_MESSAGE, "求解器被删除"),  System.currentTimeMillis(), DEF.\_ID);  taskList.remove(i);  } |

表4-11: 获取该task关联的求解器映射信息

1. 如果求解器不为空，首先从该solver求解器映射表中获取该求解器可分配的计算节点列表agentList。如果agentList不为空则逐项进行任务分配，否则留待以后处理。遍历agentList的每一项，获取agentId以及剩余负载leftLoads, 如果leftLoads大于0，则中心控制节点将发送任务运行消息到该计算节点启动求解任务计算，如果任务启动成功，则从任务列表taskList中移除该任务，更新agent表中leftLoads为leftLods – 1，最后对agentList中的所有节点进行重新排序.计算器count增1，doSend（）函数返回成功分配的任务个数。

|  |
| --- |
| else {  List<Document> agentList = Docat.getList(solverItem, DEF.\_AGENTS);  if (agentList != null && !agentList.isEmpty()) {  // 不空则逐项分配，否则留待以后处理  for (Document agentItem: agentList) {  String agentId = Docat.getString(agentItem, DEF.\_ID);  int leftLoads = Docat.getInteger(agentItem, DEF.\_LEFT\_LOADS, 0);  if (leftLoads > 0) {  int c = NextAgentKeeper.get(agentId).  sendTaskRunMessage(taskItem, solverItem, agentItem);  if (c > 0) {  taskList.remove(i);  agentItem.append(DEF.\_LEFT\_LOADS, leftLoads - 1);  this.reorderAgentList(agentList);  count++;  break;  }  }  }  }  }  return count; |

表4-12: 任务分配与后处理

NextAgentKeeper类是中心控制节点用于保持对下级节点的状态处理的类，也继承自java.lang.Thread类。

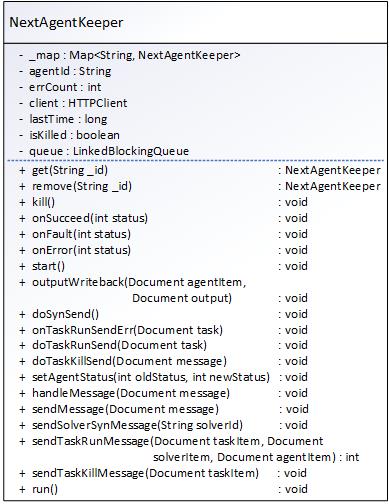


图 3.2: NextAgentKeeper类图

其私有成员里有以Map<String, NextAgentKeeper>为数据结构的\_map。Key为\_id，Value为NextAgentKeeper实例。 agentId表示计算节点id, errCount表示计算节点上运行错误的任务量计数器。lastTime表示结束时间戳，isKilled为布尔变量表示中心控制节点对该下级节点管理的线程是否结束。

queue是LinkedBlockingQueue<Document>阻塞队列。

构造方法，每个线程实例绑定一个agentId。

|  |
| --- |
| public NextAgentKeeper(String agentId) {  this.agentId = agentId;  } |

表4-13: NextAgentKeeper构造方法

onSucceed(int status)成功状态处理函数，更新时间戳，设置错误计数器errCount为0，更新该线程实例相关联的计算节点agent状态status为\_AGENT\_SYN\_STATUS\_ON\_LINE在线状态。

|  |
| --- |
| private void onSucceed(int status) {  this.lastTime = System.currentTimeMillis();  this.errCount = 0;  this.setAgentStatus(status, DEF.\_AGENT\_SYN\_STATUS\_ON\_LINE);  } |

表4-14: onSucceed函数原型

onFault(int status)未知错误处理函数，更新时间戳，并更新该线程实例相关联的计算节点agent状态status为\_AGENT\_SYN\_STATUS\_UNKNOW未知状态。

|  |
| --- |
| private void onFault(int status) { // 不该出错的  this.lastTime = System.currentTimeMillis();  this.setAgentStatus(status, DEF.\_AGENT\_SYN\_STATUS\_UNKNOW);  } |

表4-15: onFault函数原型

onError(int status)错误处理函数，更新时间戳，更新错误处理计数器errCount增1，如果errCount大于3，则设置该线程实例相关联的计算节点agent状态status为\_AGENT\_SYN\_STATUS\_OFF\_LINE离线状态。

|  |
| --- |
| private void onError(int status) {  this.lastTime = System.currentTimeMillis();  this.errCount++;  if (this.errCount >= 3) {  this.setAgentStatus(status, DEF.\_AGENT\_SYN\_STATUS\_OFF\_LINE);  }  } |

表4-16: onError函数原型

onTaskRunSendErr(Document task)任务分配失败处理函数，将以该task为参数的任务其中一些字段进行重置。

|  |
| --- |
| private void onTaskRunSendErr(Document task) {  String taskId = Docat.getString(task, DEF.\_ID);  Document value = new Document();  value.append(DEF.\_AGENT, 0);  value.append(DEF.\_VERSION, 0);  value.append(DEF.\_DIST\_TIME, 0);  DB.task.\_\_unset\_one(new Document(DEF.\_ID, taskId), value, -1, null);  } |

表4-16: onTaskRunSendErr函数原型

NextAgentKeeper中的路由函数为handleMessage(Document message)。当message中action字段为“syn”时，调用doSynSend()方法同步求解器和状态信息；action字段为“taskRun”时，调用doTaskRunSend()方法，中心控制节点方法发送任务启动命令到相应线程实例关联的计算节点上。action字段为“kill”时, 调用doTaskKillSend()方法,中心控制节点方法发送任务结束命令到相应线程实例关联的计算节点上。否则，打印消息NextHostKeeper 不支持下发消息类型。

|  |
| --- |
| private void handleMessage(Document message) {  String action = Docat.getString(message, DEF.\_ACTION, "syn");  if ("syn".equals(action)) {  M.trace("NextHostKeeper handleMessage => ",  this.agentId, " ", Docat.\_\_doc\_to\_json(message));  this.doSynSend();  } else if ("taskRun".equals(action)) {  M.trace("NextHostKeeper handleMessage => ",  this.agentId, " ",  Docat.\_\_doc\_to\_json(message));  this.doTaskRunSend(Docat.getDocument(message, DEF.\_TASK));  } else if ("taskKill".equals(action)) {  M.trace("NextHostKeeper handleMessage => ",  this.agentId, " ",  Docat.\_\_doc\_to\_json(message));  this.doTaskKillSend(message);  } else {  M.trace("NextHostKeeper 不支持下发消息类型 => ",  this.agentId, " ",  Docat.\_\_doc\_to\_json(message));  }  } |

表4-17: handleMessage路由函数原型

sendSolverSynMessage(String solverId)函数发送求解器同步消息。同步以solverId为编号的求解器。

|  |
| --- |
| public void sendSolverSynMessage(String solverId) {  Document message = new Document(DEF.\_ACTION, "syn").  append(DEF.\_SOLVER, solverId);  this.sendMessage(message);  } |

表4-18: sendSolverSynMessage函数原型

sendTaskKillMessage(Document taskItem)函数发送结束任务消息，参数为taskItem。

|  |
| --- |
| public void sendTaskKillMessage(Document taskItem) {  Document message = new Document(DEF.\_ACTION, "taskKill").append(DEF.\_ID, Docat.getString(taskItem, DEF.\_ID));  this.sendMessage(message);  } |

表4-19: sendTaskKillMessage函数原型

sendMessage与sendSolverMessage和sendTaskKillMessage方法的调用过程如下：



图4.3: sendMessage函数的引用关系

sendMessage(Document message)传递Document为类型的message，对阻塞队列queue进行加锁。将该消息message加入队列并唤醒所有线程并打印日志信息。

|  |
| --- |
| private void sendMessage(Document message) {  synchronized (queue) {  this.queue.add(message);  this.queue.notifyAll();  M.trace("NextHostKeeper recever message => ",  this.agentId, " ",  Docat.\_\_doc\_to\_json(message));  }  } |

表4-20: sendMessage函数原型

sendTaskRunMessage(Document taskItem, Document solverItem, Document agentItem)函数有以Document为类型的三个参数taskItem， solverItem， agentItem。首先根据taskItem获取taskId, 接着根据solverItem获取求解器版本号version。设置查询条件where，以及更新值update对task表记录进行更新。添加”taskRun”命令到阻塞队列queue中，并唤醒其它所有线程。

|  |
| --- |
| public int sendTaskRunMessage(Document taskItem, Document solverItem, Document agentItem) {  String taskId = Docat.getString(taskItem, DEF.\_ID);  int version = Docat.getInteger(solverItem, DEF.\_VERSION);  Document where = new Document(DEF.\_ID, taskId)  .append(DEF.\_STATUS, DEF.\_TASK\_RUN\_STATUS\_PENDING)  .append(DEF.\_VERSION, DEF.\_NOT\_EXISTS)  .append(DEF.\_AGENT, DEF.\_NOT\_EXISTS);  Document update = new Document(DEF.\_VERSION, version)  .append(DEF.\_AGENT, agentId)  .append(DEF.\_DIST\_TIME, System.currentTimeMillis())  .append(DEF.\_PATH, Docat.getString(agentItem, DEF.\_PATH))  .append(DEF.\_LOADER, Docat.getString(solverItem, DEF.\_LOADER))  .append(DEF.\_EXECUTOR, Docat.getString(solverItem, DEF.\_EXECUTOR));  synchronized (queue) {  Document task = DB.task.\_\_update\_one(where, update, null,  System.currentTimeMillis(),  null);  if (task != null) {  Document message = new Document(DEF.\_ACTION, "taskRun").  append(DEF.\_TASK, task);  this.queue.add(message);  this.queue.notifyAll();  return 1;  }  }  return 0;  } |

表4-21: sendTaskRunMessage函数原型

setAgentStatus(int oldStatus, int newStatus)方法设置agent节点的状态。

|  |
| --- |
| private void setAgentStatus(int oldStatus, int newStatus) {  if (oldStatus != newStatus) {  DB.agent.\_\_update\_by\_id(this.agentId,  new Document(DEF.\_STATUS, newStatus),  -1, DEF.\_ID);  }  } |

表4-22: setAgentStatus函数原型

NextAgentKeeper线程实例在get(String \_id)方法中启动线程。

|  |
| --- |
| public static NextAgentKeeper get(String \_id) {  synchronized (\_map) {  NextAgentKeeper ak = \_map.get(\_id);  if (ak == null) {  M.trace("启动 NextHostKeeper => ", \_id);  ak = new NextAgentKeeper(\_id);  ak.start();  \_map.put(\_id, ak);  }  return ak;  }  } |

表4-23: NextAgentKeeper线程实例启动

get(String \_id)方法相关引用图如下：



图4.4: get函数的引用关系

中心控制节点服务在LeaderServer.init()函数中进行启动，首先启动LeaderTaskSender线程实例，接着获取agent数据表中所有计算节点的集合agentList。遍历该agentList, 中心控制节点启动相应数量的NextAgentKeeper线程对下级计算节点进行状态通信以及管理，关键性代码如下：

|  |
| --- |
| public static void init() {  // 启动任务发射  LeaderTasksSender.getInstance();  // 启动相应数量的NextAgentKeeper线程  List<Document> list = DB.agent.\_\_list(null, null, Docat.\_\_select(DEF.\_ID));  for (Document d : list) {  DB.agent.\_\_update\_one(d,  new Document(DEF.\_STATUS, DEF.\_AGENT\_SYN\_STATUS\_UNKNOW),  null, -1,  Docat.\_\_select(DEF.\_ID));  NextAgentKeeper.get(Docat.getString(d, DEF.\_ID));  }  } |

表4-23: LeaderServer.init()函数原型

LeaderServiceResource, LeaderService， LeaderTaskSender, NextAgentKeeper四个类的交互时序图如下



图4.5: 任务分发时序图

LeaderFeedbackResource是Leader对Agent的内部反馈接口。当请求路径为/leader可以直接跳转到agentApi()函数，用于处理对agent的api请求。agentApi()会调用LeaderFeedbackService.router(), 处对agent的具体请求，如“solverSyn”求解器同步，“taskSyn”任务同步。然后由router()路由函数跳转到具体处理该请求的函数。当请求路径为/ leader/solver/file时，会跳转到agentSolverFileDownload函数用于处理对agent的文件同步。

## 4.2 本章小结

本章是关于作业调度平台中心控制节点对集群进行任务分发，中心控制节点对下级计算节点进行管理通信，以及调度策略的设计与实现。集群任务分发和对下级节点进行管理的核心是对线程，数据库表task, dist, solver, agent数据进行查询更新与删除操作，以及多种边界异常的处理。调度策略的设计与实现主要依据求解器映射表的填充，每个solver类型的求解任务会映射一个或多个按照leftLoads进行降序排列的agent，当发送具体任务时，可以根据求解器映射表将该求解器相关任务优先分配到剩余负载最大的计算节点上，并更新相应task, agent, 求解器映射表等有关状态字段，为下一次求解任务分发做准备。

# 第5章 实验结果与分析

## 5.1 总体设计

（1）求解器按需部署

为了验证图谱布局加速算法在图谱布局时的有效性，本实验使用两个数据集来验证，社会人员关系图谱数据集和不同量级的图谱数据集。社会人员关系图谱数据集是具有实际意义的数据集，不同量级的图谱数据集是不具实际意义的，通过程序自动生成的，节点数量在[0,500]、[501,1000]、[1001,1500]、[1501,2000]、[2001,2500]…[9500,10000]等不同区间的数据集，使用这两种数据集分别对基于KK图谱算法的人员社会关系图谱布局加速算法做了测试。根据第一个数据集的实验结果从节点重叠程度、边交叉数目的多少、图谱的整体布局效果等方面分析可视化加速算法的布局效果；根据第二个数据集的实验结果从节点数目和布局时间分析图谱布局加速算法的时间复杂度，最后分析该算法的有效性。

（2）图谱展示加速算法实验

为了验证图谱展示加速算法在图谱展示时的有效性，该实验使用节点数量在[0,500]、[501,1000]、[1001,1500]、[1501,2000]、[2001,2500]…[9500,10000]等不同区间的数据集。从节点数目和展示耗时分析图谱展示加速算法的时间复杂度，并且绘制出相应的可视化时间曲线，分析该算法的有效性。

## 5.2 基于KK图谱算法的人员社会关系图谱布局加速算法实验

### 5.2.1 实验设计

基于KK图谱算法人员社会关系图谱布局加速算法实验验证该算法的有效性以及加速效果，首先验证布局加速算法是否能够合理布局，是否对原有的算法进行了优化；其次验证展示加速算法是否可以有效展示图谱，是否相对于单线程进行了优化。实验的设计主要包括实验数据的构造、获取以及实验结果和实验分析三个方面。

第一，实验测试的数据集是验证算法的有效性以及优劣的重要因素。本实验的数据集主要来自两个途径，第一个是社会人员关系图谱数据集[24]。这个数据集构造了一个包括62个节点和159条边的社交图谱，节点表示人，节点之间的边表示人之间具有紧密的交流与对话。如果两个人之间具有频繁的交流现象，那么在这个构造社会图谱中人对应的节点之间就会有一条边。第二个数据集是不具实际意义的，属于程序随机构造的数据集，本课题所研究的图谱布局加速算法是千量级的节点加速布局，所以数据集必须达到千量级，这个数据集通过程序自动生成，并且具有一定节点和边的。每500个节点的间隔，从500个节点一直到10000个节点，分别对基于KK图谱算法的人员社会关系图谱布局加速算法做了测试。第一个社会人员关系图谱数据集主要用于验证图谱布局加速算法的有效性，第二个数据集主要验证图谱布局加速算法的加速效果。主要从时间复杂度上进行分析。第二，使用基于KK图谱算法的人员社会关系图谱布局加速算法实完成实验，获取实验结果。第三，在布局效果和时间复杂度方面对算法的有效性和优劣进行分析，基于不同节点数目的图谱，所做的布局加速实验结果，从中分析出规律或者总结出实验结论。

### 5.2.2 实验结果

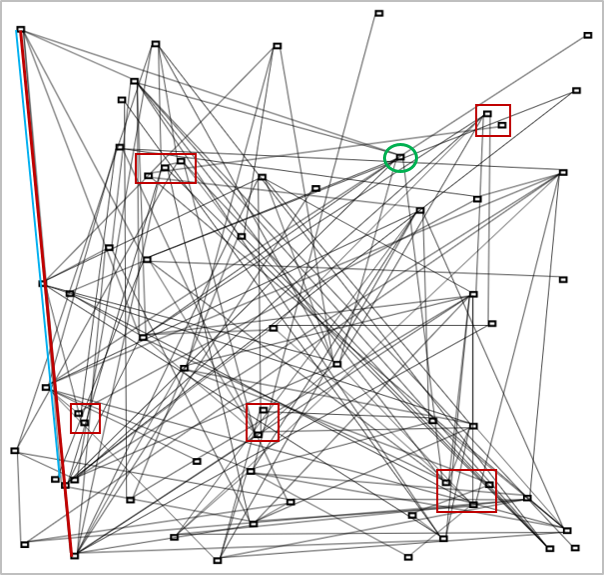
社会人员关系图谱数据集是一个包括62个节点和159条边的社会图谱。通过这个社交图谱可以验证图谱布局加速算法的有效性。在未对其使用KK图谱算法布局前，社会人员关系图谱数据集未布局可视化图如图 5.1所示。

图 5.1 社会人员关系图谱数据集未布局可视化图

从图 5.1可以看出蓝色边和红色边完全重合，红色矩形框中的节点距离太近，导致该图整体十分杂乱的，很难辨清哪些节点之间有关系，哪些节点之间没有关系；椭圆形绿色中的节点本来连接了许多节点，该节点显得非常重要，但是在改图中完全显示不出来，完全无法看出哪些节点是重要节点，具有连通其他节点的作用。下面给出使用KK图谱算法布局之后的图谱，社会人员关系图谱数据集KK图谱算法布局可视化图如图 5.2所示。

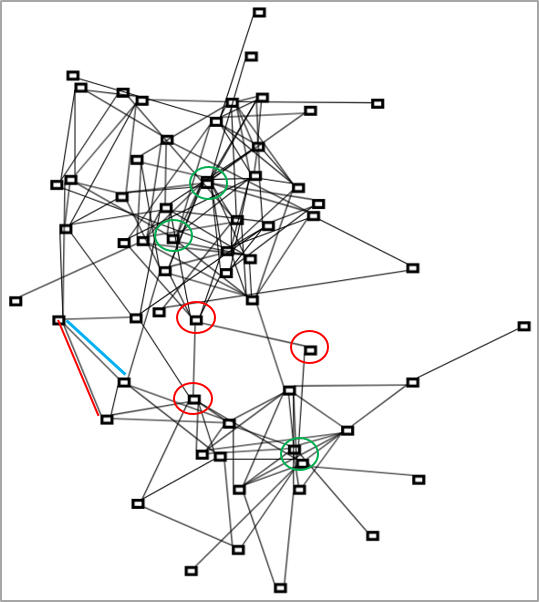


图 5.2 社会人员关系图谱数据集KK图谱算法布局可视化图

观察图5.2，可以发现图中虽然边仍然存在交叉，但是节点之间的联系可以清楚的表达出来，红色线条和蓝色线条显示的十分清楚；绿色椭圆圈起来的节点是与大部分节点有关系的节点，在次图中也可以清楚的辨认；并且红色椭圆圈起来的三节点起到连通两个社区的作用，在整个图谱中具有举足轻重的作用，在本图中也显示的特别清楚。该图整体上可以清晰的辨清图中节点之间的关系，并且能够轻而易举的辨别出哪些节点与多个节点连通，哪些节点与很少的节点连接。

以上是社会人员关系图谱数据集验证KK图谱算法的布局效果；接下来通过测试范围从500个节点到10000个节点的图谱，间隔为500个节点，记录运行时间如图 5.3所示。

图5.3 图谱可视化布局加速算法时间

从图5.3可以清楚地看出当500个节点时消耗时间0.8分钟，1000个节点时消耗时间7.1分钟，2000个节点消耗时间25.2分钟，3000个节点消耗194.6分钟，4000个节点消耗456.3分钟，5000个节点消耗837.8分钟，6000个节点消耗1400.8分钟，7000个节点消耗1900.6分钟，8000个节点消耗2567.7分钟，9000个节点消耗3408.6分钟，10000个节点消耗4429.5分钟。从图5.3结合数据可以看出，随着节点数目的增加，布局耗时也在慢慢增加，并且增加的幅度也在慢慢变大，从500个节点到1500节点时间消耗增长了22.1分钟，从1500个节点到2500个节点时间消耗增长了35分钟，从2500个节点到3500个节点时间消耗增长了234.6分钟，从3500个节点到4500个节点时间消耗增长了333.7分钟，从4500个节点到5500个节点时间消耗增长了489.1分钟，从5500个节点到6500个节点时间消耗增长了479.5分钟，从6500个节点到7500个节点时间消耗增长了654.8分钟，从7500个节点到8500个节点时间消耗增长了783.5分钟，从8500个节点到9500个节点时间消耗增长了870.5分钟。

总之，图谱的节点数目越多消耗时间越长，时间消耗的增加也在随着节点数目的增加而增加。

### 5.2.3 实验分析

对于基于KK图谱算法的人员社会关系图谱布局加速算法分别用了社会人员关系图谱数据集验证了KK图谱算法布局的有效性，使用随意生成的从500个节点到10000个节点的图谱测试了KK图谱算法的时间复杂度，及其随着节点数目变化时间的变化曲线。

实验证明，KK图谱算法的布局效果完全达到了预期，节点和边显示合理，节点和节点之间的关系清晰明了，边交叉也比较少。在不超过1000个节点时算法的时间复杂度保持在一个较小的范围里边，当800个节点时，时间消耗不超过一分钟。

## 5.3 基于多线程的人员社会关系图谱展示加速算法实验

### 5.3.1 实验设计

本实验是对基于多线程的人员社会关系图谱展示加速算法的验证，在图谱能够展示的基础上进行加速。

本实验通过比较多线程展示策略和单线程展示策略花费的时间来体现加速。本次实验测试展示加速算法使用的数据集通过程序自动生成具有一定节点和边的千量级图谱。节点区间为[0,500]、[501,1000]、[1001,1500]、[1501,2000]、[2001,2500]…[9500,10000]的图谱使用单线程方法和多线程方法分别展示，然后比较各自所消耗的时间。

最后，基于不同节点数目的图谱，所做的展示加速实验结果，从中分析出规律或者总结出实验结论。

### 5.3.2 实验结果

实验测试的社会图谱数据包括500、1000、1500…10000个节点的图谱，分别使用单线程展示策略和多线程展示策略，如图 5.4所示。

图 5.4 图谱展示加速算法运行时间

图 5.4是不同数量节点的图谱使用单线程策略展示和多线程策略展示时，所消耗的不同时间。当图谱具有500个节点时单线程耗时127毫秒多线程耗时112毫秒，1500个节点时单线程耗时210毫秒多线程耗时210毫秒，2500个节点时单线程耗时253毫秒多线程耗时235毫秒，3500个节点时单线程耗时290毫秒多线程耗时270毫秒，4500个节点时单线程耗时336毫秒多线程耗时321毫秒；5500个节点时单线程耗时370毫秒多线程耗时356毫秒；6500个节点时单线程耗时430毫秒多线程耗时410毫秒；7500个节点时单线程耗时510毫秒多线程耗时479毫秒；8500个节点时单线程耗时546毫秒多线程耗时518毫秒；9500个节点时单线程耗时590毫秒多线程耗时518毫秒；10000个节点时单线程耗时610毫秒多线程耗时566毫秒；

总体上，多线程策略展示要优与单线程策略展示。在500个节点时多线程展示策略消耗112ms的时间，单线程展示策略需要127ms的时间，它们之间没有太大的区别。虽然多线程展示策略和单线程展示策略之间的差别不是特别明显，但是还是可以看出随着节点数目的增多，多线程的优势越来越明显，它们之间消耗的时间差别越来越大。

### 5.3.3 实验分析

随着节点数目的增多，多线程展示策略与单线程展示策略之间的时间差越来越大，也就是多线程展示策略越来越有优势，这是因为随着节点数目的增多，多线程额外消耗的上下文切换以及线程资源管理所消耗的时间变得更加微不足道，所以多线程的优势更加明显。随着节点的增多，这种优势会更加凸显。

## 5.4 人员社会关系图谱可视化加速算法应用

可视化加速算法只能在实际应用中体现其实际价值，该部分主要从广告精准推荐和社交娱乐三个方面说明其实际应用。

### 5.4.1 用户广告精准推荐

广告推荐需要将用户的兴趣爱好以及社交行为等数据收集起来，从中挖掘出隐藏的其他用户，将广告推荐给这类用户，某人的社会关系图谱如图5.5所示。

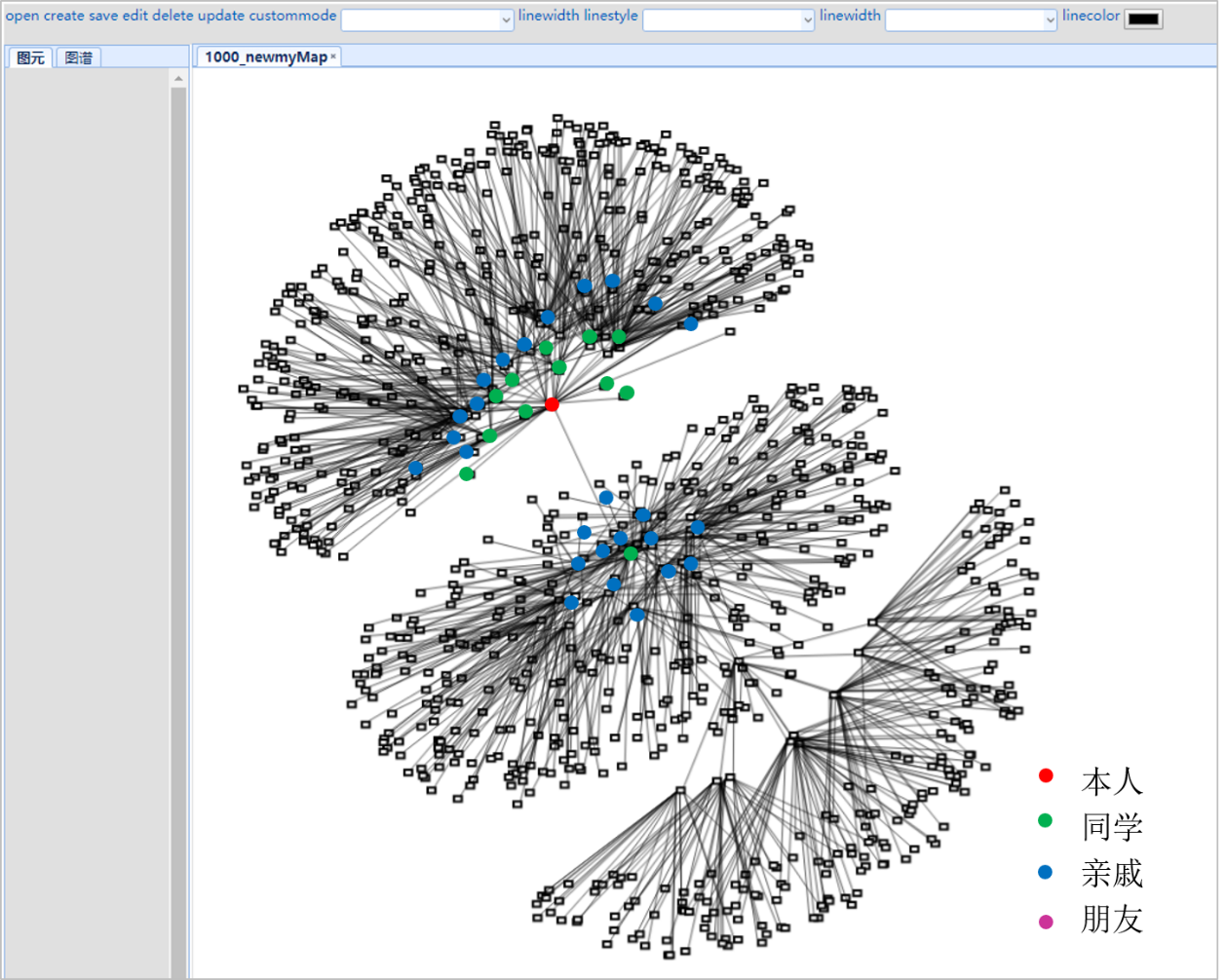
图 5.5 红色人的社会关系图谱

图5.5中红色人喜欢电子产品，则与其紧密联系的同学、同事、QQ好友和亲人（即图中的绿色人）也可能喜欢电子产品，与这些人紧密联系的人群（即图中的蓝色人）也就有很大可能喜欢电子产品，可以再次扩展人员社会关系图谱，当图谱中节点数目过多时图谱可视化就会消耗过多的时间，通过此算法优化了图谱可视化的时间，图5.5使用可视化加速算法时耗时为37ms。

### 5.4.2 社交娱乐

在社交娱乐方面社会关系图谱一直起着举足轻重的作用。谈到社交娱乐，在21世纪不得不说网络游戏，或者称之为社交游戏。网络游戏利用社会关系图谱将密切相关的人聚集起来，在游戏中促进了人与人之间的互动、交流，促进了文化的传播。比如，2016年大火的游戏“王者荣耀”，该款游戏，通过腾讯的其他软件的巨大流量，迅速的在青少年以及中年人群中快速传播。该游戏中根据用户的游戏等级，可以自动匹配与之相应的玩家，将各行各业的各个年龄段的人都聚集到一起，提供了一个互动交流平台；并且该游戏可以于用户的QQ好友、微信好友、同事、朋友和亲戚匹配，这也是基于人员社会关系图谱，如图5.6所示。

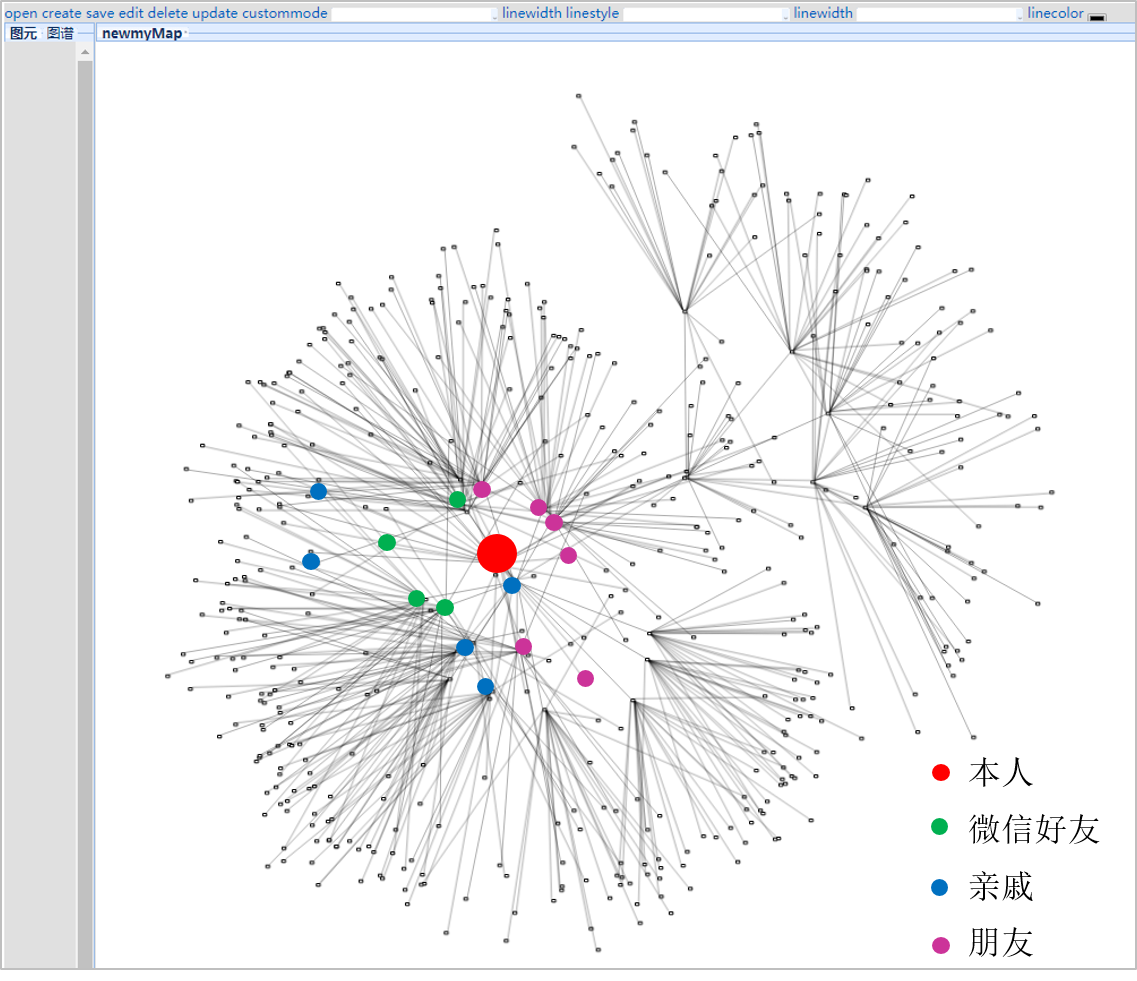


图5.6 游戏人员关系图

图5.6中红色人与绿色表示微信好友蓝色表示的亲戚以及紫色表示的朋友匹配玩游戏，随着这个关系图谱的扩展，图谱可视化消耗更多的时间，使用人员社会关系图谱可视化加速算法可以优化图谱的可视化。

## 5.5 本章小结

本章是关于基于KK图谱算法的人员社会关系图谱布局加速算法和基于多线程的人员社会关系图谱展示加速算法的验证实验,以及可视化加速算法的应用。

图谱布局加速算法的验证实验主要检验该算法的有效性和优化效果。社会人员关系图谱数据集完美的测试了图谱布局加速算法的有效性以及显示效果，500到10000个节点不同规模的图谱测试了算法的时间消耗，总体而言，图谱布局加速算法已经有较低的时间复杂度，测试证明也消耗了相对较少的时间，特别是在布局方面，具有比较好的布局效果。

图谱展示加速算法的验证实验主要是检验算法的优化效果，这个优化是与单线程展示策略相比较体现出来的。测试发现在节点数目比较少时（500个节点），单线程展示策略与多线程展示策略所消耗的时间差不多，多线程展示并没有太多优势，但是随着节点数目（7000个节点）增多时，多线程展示策略的优势开始显现，所以多线程展示策略更加适合于千量级节点图谱甚至更大规模的节点数目的图谱。

人员社会关系图谱可视化加速算法的应用主要依据是人员社会关系图谱加速算法的特征、价值以及一些重要优势，其典型应用主要分为四大部分。本章主要对社会化推荐和社会化娱乐的应用方面做了详细介绍。广告精准推荐属于社会化推荐，社交娱乐属于社会化娱乐范畴。随着互联网的发展，人员社会关系图谱可视化加速算法还会在更多领域应用。

# 参考文献

###### Time, clocks, and the ordering of events in a distributed system.

###### 吴渝,林茂,雷大江.一种3D空间中的两级力导引可视化算法[J].重庆邮电大学学报,2015,27(5):668-671.

###### 吴鹏,李思昆.适于社会网络结构分析与可视化的布局算法[J].软件学报,2011,22(10):2467-2475.

###### Kozo Sugiyama,Kazuo Misue.Graph Drawing by the Magnetic Spring Model.[J].Journal of Visual Languages and Computing,1995,Vol.6(3):217-231.

###### Andreas Noack.An energy model for visual graph clustering.[J].Liotta, Giuseppe(ed.), Graph drawing.11th international symposium, GD 2003, Perugia, Italy, September 21-24.

###### Yehuda Koren,Ali Çivril.The Binary Stress Model for Graph Drawing[B]. Lecture Notes in Computer Science2009 Springer.

###### Xiaoxin Yin,awei Han,Philip S. Yu.CrossClus: user-guided multi-relational clustering[J]. Data Mining and Knowledge Discovery,2007,15:321-348.

###### Jianguo Jin.PPOS SYSTEM: A System of Partitioning Polygonal Objects[C].Information Science and Engineering,2007,42:43-65.

###### Haojun Sun,Shengrui Wang,Qingshan Jiang.FCM-Based Model Selection Algorithms for Determining the Number of Clusters[J].Pattern Recognition,2004(37):2027-2037.

###### 杨立文.基于改进的社团发现算法的社区发现技术[D].吉林.吉林大学,2012:23-25.

###### 金超.基于图聚类的社会网络数据挖掘算法研究[D].山东,山东理工大学,2017:28-31.

###### 井靖,蒋烈辉,刘铁铭,司彬彬,曾韵,朱晓清.软件逆向分析过程中的多维图谱抽取方法.[J].计算机应用与软件,2016,33(4):1-4.

###### 水超,陈涛,李慧,陈国升.基于力导向模型的网络图自动布局算法综述[J]. 计算机工程与科学期刊,2015,37(3):457-460.

###### 赵俊岚. XML编程中的DOM与SAX技术[J].计算机工程,2004,30(24):70-72.

###### 李海威,韦天瀚.基于Q函数优化的加权有向复杂网络模糊聚类算法设计研究[J].广东科技,2016,10(4):54-55.

###### 曹梦琦,李德敏,张光林,郭畅.一种时间最短的交通网络路径求解方法[J]. 计算机工程与应用,2018:2-6.

###### 王荣,江东,韩惠.基于Floyd方法的最短路径算法优化算法[J].甘肃科学学报,2012,24(4):1110-114.

###### 王顶,徐军,段存玉,吴玥瑶,孙静.基于PageRank的用户影响力评价改进算法[J].哈尔滨工业大学学报,2018,50(5):61-63.

###### 郭华东.基于线性回归与马尔科夫链相结合的云资源监控预测算法研究与实现[D]. 浙江,浙江大学,2017:17-19.

###### Tomihisa Kamada,Satoru Kawai.An algorithm for drawing general undirected graphs.[J].Information Processing Letters,1989,Vol.31(1):7-15.

###### 吴海林.基于社区划分和改进PageRank的影响力最大化算法[J].移动通信,2017,10:82-84.

###### MOORE B CJ.An introduction to the psychology of hearing[M].London:Academic Press,2003.

###### 崔俊明,李勇,李跃新.基于非加权图的大型社会网络检测算法研究[J].电子技术应用,2018,44(2):80-83.

###### Gail-Joon Ahn,Ravi Sandhu.Role-based authorization constraints specification.Acm Transactions on Information and System Security,Vol.3,No.4,2000.

# 致谢

四年前，我怀着对象牙塔的憧憬和神圣殿堂的向往，带着纷繁的心情和梦想，第一次来到了哈尔滨这座城市，一切都是新奇与陌生，从最初高考后懵懵懂懂的选择了软件工程专业，到如今四年的时光已经悄然逝去，我很清楚的明白自己当初的选择是正确的，人生中坚持并一直着手去做自己喜欢并感兴趣的东西是很难得的。大三那一年，除了学好本专业课程外，把剩余的课余时间全都花在了导师指导的科研项目和实习工作中，在导师的指导中以及实习工作的磨砺中，我感觉到了自己的不足，也感受到了自己一天天成熟带来的变化。

感谢您，我的导师，我的大学，正是在给予我的每一种挑战与机遇中，让我懂得了学习与生活。懂得时间与成长，懂得奋斗和拼搏。是你为我架起了走向成熟的桥梁，在慢慢人生的长路上，你将是我多彩世界永远美丽与难忘的记忆。品味人生，才能珍惜人生。走过岁月，走过季节，也走过我的大学时代。大学生活留给我更多的时间去思考。想想走过的路，想想现在的路，想想未来的路，不能不说是对自己的重新认识。

六月了，我已经闻到了离别的气息，四年的大学生活也将结束，我也终于明白“转瞬即逝”的含义。大四，在这即将远离大学时代的时刻，才真正懂得回眸的意义，再见青春，永恒的迷惘。