

摘要

随着大规模电子商务行业中数据传输量的增加，管理如此大量的请求成为一个关键问题。通过在分布式服务器系统上应用任务调度来处理请求是一种有效的方法。但是，当服务器节点在短时间内处理极大量的请求时，过高负载量是不可避免的。没有有效的高负载监视方法，很难高效的对后端服务器集群进行监控和管理。据我们所知，监控高负载异常节点的现有方法既不灵活也不直观，并且不能检测服务器节点异常，例如定位客户端发送的不合理请求。在本文中，我们提出了一种基于真实数据集的可视化分析的作业调度监控方法，该方法允许监控人员了解该区域中运行节点的状态，并通过各种视图组件观察导致高负载服务器的可疑请求。这种思路为服务器端任务调度集群监测提供了一种全新的方法，并且经过测试显示是可行且有效的。

关键词：数据可视化 任务调度 异常检测

Abstract

With the increasing amount of data transmission in large-scale e-commerce industry, managing such an enormous amount of requests simultaneously becomes a key problem. It is an effective method to handle the requests by applying task scheduling on the distributed server system. However, high load on the servers is inevitable in scheduling when requests of a server node process extreme large amount of tasks in a short period of time. It is difficult to maintain load balance without an effective high-load surveillance approach. To best of our knowledge, existing methods of monitoring high-load abnormal nodes are neither flexible or intuitive and are not capable of detecting server node anomalies such as positioning unreasonable requests sent by the clients. In this paper, we propose a job scheduling monitoring method based on visual analysis from a real dataset, which allows the monitoring personnel to know the status of running nodes in the area and observe the suspected requests causing a high load of servers via various view components we inject into the system.

Keywords: Task scheduling, visual analysis, abnormal detection

第一章 绪论

1.1 研究工作的背景与意义

现如今网络电子商务项目正在蓬勃发展，人们在享受着足不出户购物和信息浏览，但是往往当使用者的数目变得足够多时，安全隐患往往随之到来。尽管各大互联网企业采取了一些应对措施，比如将用户节点置于类比虚拟机的容器中，或者使用算法对用户请求进行调度处理，使指令被分散在不同的服务器上进行处理，但如果控制这台用户节点的服务器因为某些原因出现超负荷运转，甚至宕机的话，目前整个相关领域是没有很好的监视系统来全程监视负责处理用户请求的服务器的运行状态的。比如全球先进的超级计算机研究所——美国德克萨斯州高级计算中心 (TACC) 现在存在检测正在运行服务器的监视器，但是功能和用户操作流程十分不人性化：他们必须通过具有丰富经验、工作年限很长的后台管理者，通过报错信息得到运行异常的服务器编号，再去监测系统手动输入，通过查看后端返回的一系列运行参数，通过经验判断这台服务器的出错原因从而采取补救措施。类似这样的做法，缺点十分明显，比如在整个操作的过程中，操作流程十分僵硬，充斥着很多人为感知和人为判断，使得操作效率的低下；或者在某一时刻，如果因为在某一地区的用户普遍都出现了系统故障导致在短时间内出现了大面积的服务器运行异常，在如此庞大的工作量之下，工作人员的补救效率在不先进的系统下是很迟钝的，这种中间环节的纰漏就会导致解决过程的滞后，从而引发连带问题。

1.2 本文的主要贡献与创新

本论文的贡献主要是，通过数据可视化手段，实现了对服务器工作状态的全过程实时监测，克服了以前人为操作的种种不利，在功能上也进行了扩充，可以实现的具体功能和创新点如下：

- 及时而直观地了解到时间节点下，正常和异常服务器的运行状态参数 (CPU 利用率、memory 利用率、disk 利用率等) 和服务器本身的各项硬件指标 (CPU 数目、memory 大小等)，通过时间和服务器节点实时切换；

- 通过服务器所带的任务 (task)、作业 (job) 和实例 (instance) 信息，分析出导致服务器运行异常的原因是由什么任务，或者哪一台用户的异常操作而引起的，在可视化视图中分析异常原因；
- 直观地观察到服务器任务 (task)、作业 (job) 和实例 (instance) 之间的包含关系和数目大小，及时得到主要占用服务器资源的作业信息，做出相应调整；
- 通过热力图刷新迅速掌握某服务器节点在过去一段时间的工作状态，帮助工作人员寻找规律，辅助判断异常节点的出错动机；
- 了解到在整个有记录的时间线上所有服务器节点的运行状态和某一时刻下的全平台的服务器异常数目，定位异常高峰，联动其他组件有针对性地确定需要观察的时刻。

总而言之，该方法的创新点就是在传统的监测系统的基础上，通过增加可视化元素，使操作过程变得简洁流畅；在此思路的基础上，增加了各种功能，来帮助监测人员合理地定位异常，通过可视手段分析异常原因，定位故障源，从而有效地实时控制服务器的运行状态，为庞大的商业系统的正常运行保驾护航。

1.3 本论文的结构安排

本文的章节结构安排如下：

第一章：绪论部分，简述研究背景和现阶段研究进展，分析需求从而得出需求；

第二章：详细阐述本文的相关研究、研究意义、研究手段、简述数据可视化方法和研究数据集等研究基础；

第三章：分析系统设计思路，按模块化阐述系统的各部分功能，通过操作视图来具体演示系统的工作流程从而实例化分析，并且加入了测试用例来检验我们想法的正确性和系统的可行性；

第四章：全文的总结和归纳，分析系统仍然存在的缺陷，以及对未来工作的展望；

第五章：参考文献

第二章 项目意义及背景介绍

2.1 项目意义

对电子商务任务调度模式下的异常监测的可视化实现，对此类控制平台提出了一种新的思路和方法，也是数据可视化在交叉领域的全新进展。

2.1.1 电商集群的规模化

2018 年 7 月 10 日，2018 中国互联网大会发布了新一版的《中国互联网发展报告 (2018)》。报告中指出，2017 年，中国电子商务交易服务营收规模为 5027 亿元，首次突破 5000 亿大关。2017 年第三方互联网支付也达到 143.26 万亿，网络购物市场交易规模达 5.33 万亿元，而网络零售的市场交易规模为 7.18 万亿。中国网上支付用户规模达 5.31 亿人，其中手机支付用户就达 5.27 亿人，较 2016 年底增加 5783 万人，年增长率为 12.3%，规模增长迅速。正如网民在现实生活中体会到的，电子商务交易已经成为我们购买日常用品的首选手段，在中国庞大的网民基数和中国网络技术飞速发展的双重影响下，这一数字在以后还会以大增幅增长。根据商务部统计，2020 年预计中国网络零售市场规模为 9.6 万亿，是 2012 年的 10 倍之多。电子商务在深度影响着国民生活的同时，也掌握着货架经济的命脉，倘若电商集群由于某种原因崩溃的话，给整个国家带来的影响即将是灾难性的。

中国电子商务的龙头企业——阿里巴巴网络技术有限公司 (以下简称阿里巴巴)，在国内的电子交易平台里扮演着举足轻重的地位。根据 2018 年阿里巴巴公布的财年财报显示：2018 全年，阿里巴巴营收 2502.66 亿元人民币（约 398.98 亿美元），同比增长 58%，核心电商业务收入 2140.20 亿元人民币，同比增长 60%，均创下 IPO (Initial Public Offerings, 指股份公司首次向社会公众公开招股的发行方式，简称 IPO) 以来年度最高增幅，2018 财年净利润为 832.14 亿元人民币（约 132.66 亿美元）。如此庞大的交易额和成交额使专家和研究人员的目光放在其身上。2018 年阿里巴巴在 Github 上公布了一组数据，该数据刻画了阿里巴巴在 8 天地域范围内 4000 台服务器的运行状态的数据，而这组数据也将成为本项目的研究数据集，在本章第三部分，将会着重对该数据进行详细介绍。

2.1.2 故障问题及手段

电子商务平台的实现其实并不是一个不能达到的要求，但是任何系统架构在达到一定规模之后，大大小小的问题往往会接踵而至，而这也是一个企业生存的关键。2018年8月1日，阿里巴巴旗下的淘宝网交易平台，淘宝服务器出现大范围的故障，全国多地网友在微博反馈称自己的淘宝崩溃无法查看订单，淘宝 App、PC 版网页均出现“网络竟然崩溃了”的提示，即使切换网络和重启手机也无效，在长达数小时的等待之后，该漏洞得到了修复。具体的原因，阿里巴巴并没有给出明确的回复，之后这件事也就不了了之。然而这已经不是阿里巴巴遇到的第一次服务器崩溃事件，每隔数月就会时不时的发生类似的服务器故障。由此可见，即使是如此宏大的电子商务企业也会因为后端服务器的宕机事件，造成企业形象的不良影响的同时也造成了利润的亏损。换句话说，如果能快速发现故障，找出导致该进程异常的原因，再利用分支等手段同时进行维护，结果将会截然不同。而本系统的设计初衷就是以阻止此类问题的发生而提出的。

2.1.3 设计优势

本系统设计将基于可视分析，在对服务器异常进行图形化展示的同时，还拥有异常定位、时序性监测等特点，对出现异常信号的服务器节点进行实时监测，并直观的展示造成该异常的任务，从而达到快速、准确的定位故障的目的。

2.2 项目背景及相关理论

2.2.1 研究现状

面对异常检测问题 [1]，Xiaowei Qin 等人提出了一种面向对象的检测框架 [2]，它具有两步聚类，称为沙漏聚类。两个参数，关键质量指标和因果参数，通过结合自组织映射（SOM）和 k-medoids 的混合算法，将它们聚类成不同的类型。Pei Yang 等人 [3]。建议使用生成的拮抗网络 (GAN) 来检测异常。Daojing He 等人 [4]，介绍使用软件定义网络 (SDN) 检测流量异常的优势。A.R.Jakhale[8] 使用数据挖掘 [5][6][7] 技术，利用滑动窗口模型和聚类技术检查网络流的异常数据包。Alireza Tajary[9] 等，提出了一种吞吐量感知的瞬态故障检测方法，它利用了多核服务器处理器的特性。

为了识别大型，动态和异构数据中的异常 [10]，Nan Cao 等，介绍一种视觉互动 [11][12][13] 系统和框架，称为 Voila。该系统主要实现在线监测和与用户的互动。Y.B. Luo 等人 [14]，提出了一种基于流量限制可穿透能见度图（FL-LPVG）的异常

检测方法。该方法基于网络流序构建复杂网络,挖掘相关图的结构行为模式,提取网络流特征序列,利用 LPG 将统计特征序列转换为关联图,通过数据挖掘和信息检测异常流量基于熵的理论技术。其优点是该方法大大简化了异常检测过程,有效降低了高维数据的维数。但是为了提高这个系统的效率,我们必须从大量数据中完全挖掘行为特征。因此,它肯定会带来如何处理大数据以及如何提取有效信息的挑战。

Josef Kittler 等人 [18], 在解决异常检测问题时引入域异常的概念 [15][16]。异常有许多方面,每个域都是异常的许多方面之一。在此基础上,他们参考贝叶斯概率推理设备,并提出统一的异常检测框架 [17],以识别和区分每个领域的异常。该设备通过定义各种异常属性来提供域异常事件的分类。该框架的创新特点是它暴露了异常的多方面性质,并且可以识别可能导致异常事件的各种原因,以及相应的检测机制。

2.2.2 数据可视化

本项目的特征是基于数据可视化的电商平台集群管理。数据可视化是由于视觉给人类带来的感知是最直观最有效率的一种获取信息的手段。目前国内的数据可视化行业也在蓬勃发展,涌现除了不少有能力的人才和惊艳的科技成果,下面将用一小部分篇幅来介绍一下这门领域的相关特点及其主要用途。

从生物学的角度来考虑,人类所有器官能接收到信息的 80% 都来自于视觉,在大数据时代下,对信息的表达则就显得尤为关键,而数据可视化也就承担起了这一重要任务,充当着数据和人类之间的关键载体。顾名思义,数据可视化是关于数据视觉表现形式的科学技术研究。其中,这种数据的视觉表现形式被定义为,一种以某种概要形式抽提出来的信息,包括相应信息单位的各种属性和变量。它是一个处于不断演变之中的概念,其边界在不断地扩大。主要指的是技术上较为高级的技术方法,而这些技术方法允许利用图形、图像处理、计算机视觉以及用户界面,通过表达、建模以及对立体、表面、属性以及动画的显示,对数据加以可视化解释。

2.2.3 电子商务分布式任务调度

作业调度是用于控制作业的无人值守后台程序执行的计算机功能,也称为批量调度。作业调度程序需要协调实时业务活动与跨不同操作系统平台和业务应用程序环境的传统后台 IT 处理的集成。作业调度程序需要通过几个参数管理客户端计算机集群的作业队列,例如作业优先级,计算机源可用性 or 用户允许的同时作业数。这种作业调度方法已成为大型电子商务业务平台处理用户并发请求的常用方法。

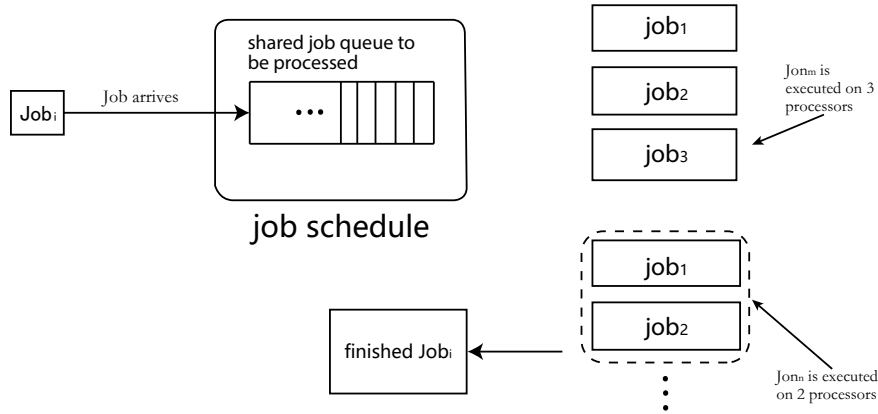


图 2.1: 作业调度模型

图 2.1 说明了本文假设的作业调度程序模型。在该图中，用户提交的作业到达共享作业队列，并且作业调度器获得作业请求的处理器数量（`job` 大小）。没有给作业调度程序提供其他信息。作业调度程序收集处理器的状态，空闲或忙碌，并将共享作业队列中的作业分派给空闲处理器。这里有两个执行作业的策略。在第一个策略中，作业调度程序将作业调度到 S 处理器，并保证在 S 处理器上执行作业直到完成。这里， S 表示工作规模。在第二个策略中，作业调度程序不保证在 S 处理器上执行作业，也就是说，执行作业的处理器数量取决于并行计算机上作业的拥塞程度。在第一个策略中执行的作业称为刚性作业。

同时，一旦操作员处理了工作流程被实例化，它就被称为“任务”。实例化在调用抽象运算符时定义特定值，参数化任务成为 DAG 中的节点。基于该层处理，任务被分成几个部分，可以分派到服务器进行处理，这通常称为“任务实例”。任务实例表示任务的特定运行，其特征在于标记，任务和时间点的组合。由于流程的抽象和复杂性，图 2.2 向我们展示了如何作业实例将逐步拆分为任务实例。

2.2.4 数据集分析

本研究中使用的数据集 `cluster-trace-v2018`¹ 将从我们的一个生产集群中进行跟踪和采样，`cluster-trace-v2018` 是 Alibaba Open Cluster Tracking Program² 的一部分。该计划由阿里巴巴集团发布，并于 2017 年提出并开始跟踪。通过实际生产提供集群跟踪，该计划帮助研究人员，学生和对该领域感兴趣的人更好地了解现代互联网

¹ https://github.com/guanxyz/clusterdata/blob/master/cluster-trace-v2018/trace_2018.md

² 阿里巴巴集群跟踪计划由阿里巴巴集团出版。通过提供来自实际生产的集群跟踪，该计划帮助研究人员，学生和对该领域感兴趣的人更好地了解现代互联网数据中心（IDC）和工作负载的特征。
<https://github.com/guanxyz/clusterdata>

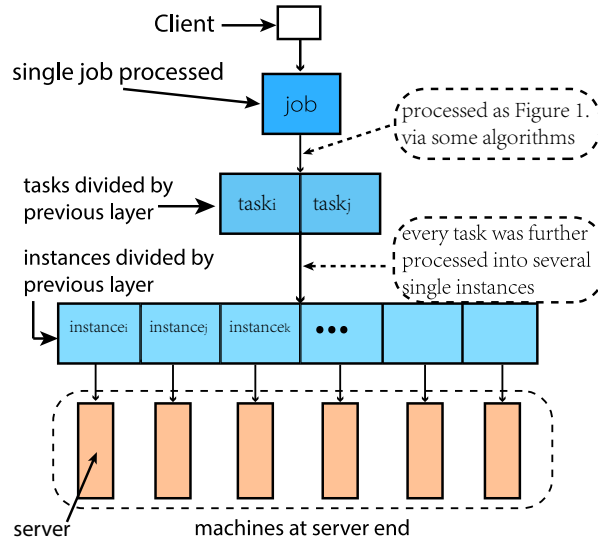


图 2.2: 来自作业层的工作流程型

数据中心的功能和工作量 (ID)。该项目目前已在 Version 2017³和 2018 版本中提供。2017 版本是该程序的原型，它记录了 3000 个服务器节点的 12 小时跟踪。为了研究的全面性和完整性，我们打算在 2018 年版的基础上进行研究。cluster-trace-v2018 包括大约 4000 台机器，跨时 8 天，由 6 个单独的数据文件组成，数据文件总大小达到了 206GB，以下是该数据集的简要介绍：

- machine_meta: 机器的元信息和事件信息。
- machine_usage: 每台机器的资源使用情况。
- containe_meta: 容器的元信息和事件信息。
- container_usage: 每个容器的资源使用情况。
- batch_instance: 有关批处理工作负载中实例的信息。
- batch_task: 有关批处理工作负载中实例的信息。

以上六个数据集在空间性上和时序性上向我们充分展示了每台服务器的信息、运行情况以及服务器所接受的任务调度，横向和纵向的展示了在为其 8 天的事件范围内所有任务的调度情况以及接受这些调度的机器的运行状态。

³ <https://github.com/guanxyz/clusterdata/tree/master/cluster-trace-v2017>

2.3 关键问题

2.3.1 数据处理

由于研究所采用的数据集十分庞大，故成为了本项目进行的一大难题。用以往的方法去分析和处理该数据集并不是最理想的手段，比如将整个数据读入内存然后进行处理，或者将数据按时序拆分成多个数据文件在单独进行处理。前者不可行的原因是实验地点的硬件条件还不能满足将如此庞大的数据这个整个读入内存，这必将造成计算机内存过载导致系统崩溃，而后者也不是最理想化的解决办法，因为比如将数据文件按天数切割成 8 份，每一份的文件大小也将达到 15GB，即使可以读入内存但是也造成了内存占用率过高从而导致其他进程无法正常运行。

在另一方面来说，假设我们用某种方法将数据成功的导入了我们指定的数据库，但是在后台与数据库交互的过程中，由于查询的数据量过于庞大，会导致超出后台数据接口所能容纳的数据量最大限额，导致查询时间过慢或者因为超出接口限额请求时间过长而进程中断导致程序崩溃。

综上所述，以上两个难点都是因为数据量的规模所导致的问题，所以如何处理大数据量的数据集是本次项目亟待解决的一个难点。

2.3.2 集群表示及定位故障源

如何将任务 (job)、作业 (task) 和作业实例 (instance)(以下分别简称 job、task 和 instance) 与运行节点通过可视化手段巧妙地结合起来将成为一个有价值的话题。按照传统方法，先将所有节点与其运行状态在一个视图中表示出来，再通过交互使之与另一数据表中的 job 等参数链接起来，虽然可行但是再操作的过程中略显繁琐。而且根据数据文件的特征，决定了不能从节点使用情况的视图出发去查询作业实例等信息，两者的数据文件中的时间戳存在被包含的关系，所以不能用传统机械的方法去处理本项目的数据结构。

该关系可以大致归纳为一个树形结构，job、task 与 instance 从前往后依次存在包含关系，而对于每一个 instance 都有一台独立的服务器节点用来处理该 instance 所发出的请求。该请求的分布可能是分布式的，这就导致了通一个 job 下所包含的 instances 被调度到了不同的节点来进行处理，所谓分布式任务调度。

值得注意的是，用户节点与 job 之间只存在一对多的关系，不存在不同用户所调度的 job 属于同一个 job，这一点十分关键，在第三章第一节将具体分析此数据结构的特点。

2.3.3 故障域算法

用来判断硬件性能的算法不少，但是用于检测服务器负载异常的算法却不多见，而且使用范围有限。大规模服务器集群的异常确定算法类似于云计算基础设施中异常识别的处理模式。该方法，最相关的主成分（MRPC）⁴，由于缺乏动态规范化单位和规模的关键参数，因此无法应用于我们的研究。

2.3.4 时序性及空间性

在研究目标为实时性的要求下，对时序性的反映往往是至关重要的。采用数据库的目的是可以实时对数据库内的记录进行可视化输出，然而这还远远不够，我们需要通过节点的过往记录和对未来节点运行状态的预测来具有时效性的描述节点，这样管理者不但可以从视图的渲染中找到异常规律从而在以后有针对性的操作节点，还可以对未来节点的运行状态进行预测，适当关闭或者限流问题节点从而对整个服务器集群和用户端的操作进行优化和改进。

在布局的空间性问题上，往往要在有限个数的可控维度上尽可能多的表示信息，所以节点的空间性排布成为一个值得商榷的问题。按照团队的初步设想，将节点表示为具有时序性的三维排布，每个维度各自表示判定异常的相关参数（CPU 利用率、memory 利用率、disk 利用率），但这样做的成果其实并不理想：使用者并不关心每个节点的异常参数，所以对维度的利用是一种浪费，不符合可视分析的规范。

⁴ Most relevant procedure component

你好

第三章 系统设计与实现

3.1 设计思路

3.1.1 主要任务

为了解决现有任务调度控制方法的不人性化和不直观以及拓展更多功能的需求，故提出本项目，旨在为解决以上问题而更近一步。具体任务列举如下：

- 通过可视化分析的方法及时地了解到某时间节点下，正常和异常服务器的运行状态参数 (CPU 利用率、memory 利用率、disk 利用率等) 和服务器本身的各项硬件指标 (CPU 数目、memory 大小等)，并且可以进行实时切换的功能；
- 通过调度信息 (task、job 和 instance)，由图形展示出导致异常的原因归咎于哪个具体 task 或 job，由此得到用户操作对后端服务器的影响；
- 直观看到服务器任务 (task)、作业 (job) 和实例 (instance) 之间的包含关系以及数目大小，得到主要占用服务器资源的作业信息，做出相应调整；
- 通过热力图刷新迅速掌握某服务器节点在过去一段时间的工作状态，帮助工作人员寻找规律，辅助其他模块判断异常节点的出错动机；
- 了解到在某一时间点下拥有记录的所有服务器节点的运行状态和该时间下运行某任务的服务器异常数目，定位异常高峰，并由此有针对性地确定需要观察的时刻。

本文拟根据以上需求设计并实现相应的系统模块，设计和实现过程在本章的余下章节将展开详细说明。

3.1.2 关键问题的解决方法

3.1.2.1 数据处理

上文中提到，处理、读取并查找庞大数量级的数据文件成为了一大难题。出于对本项目模型性和实时性的考虑，经过团队的讨论最终决定选择八天中第二天的数据作为数据集模拟数据流进行研究。然而问题是经过分割后的数据文件的体积仍然超过了研究地点所有独立设备的内存极限，解决方法在于 python 对数据处理方法的改变：如算法 1 所示，改变以往的处理中小型数据的思路和方法，将庞大的数据集每次按需读入设备内存进行处理，故得出两种可行的方法，python csv 中的按行读取方法和 pandas 的块级读取的方法，即每次根据电脑内存大小只读取原数据文件的一小部分，等这一部分被处理和计算结束后再去重复操作处理其他部分的数据。这样便实现了任意大小的文件都可以被任意设备进行处理和计算的方法。

Algorithm 1 Huge data file reade

Input: data_path
 1: **initialize:** reader=csvReader(data_path)
 2: **for** row **in** reader **do**
 3: filter by Conditions
 4: write(row)
 5: **end for**

3.1.2.2 集群表示及定位故障源

为了准确而简洁地表示每一个任务所持有的 job、task 与 instance 参数，以及如何将三者与具体被调度地服务器节点联系起来，经过讨论拟采用以下方案：采用可是分析展示方法中的 tree 结构，可以清晰明了地展示三者之间地层次关系。与此同时，在叶子节点的节点处与相应处理服务器绑定，反映正在运行此 instance 的服务器所处的运行状态。通过以上方法，就完美解决了任务调度信息的层次关系和不同数据文件之间的关联。

3.1.2.3 故障域算法

数据集本身有许多参数来说明后端服务器节点的状态。控制阈值是确定异常域的有效方法，取决于三个有价值的领域：CPU 利用率，内存利用率和磁盘利用率。通过为这三个参数分配权重，系统可以确定节点在处理任务实例时是否处于高负荷。算法二描述了在我们的研究中识别异常的方法：

$$Value_{hl} = x \cdot \frac{S_{\beta+\gamma}}{S} + y \cdot \frac{S_{\alpha+\gamma}}{S} + z \cdot \frac{S_{\alpha+\beta}}{S}$$

在上面的算法中， α ， β 和 γ 分别代表 CPU，内存，磁盘的阈值。我们使用概率统计学的思想来设置适当的阈值，以保持异常高负荷节点的比例接近 10%。可以像这样计算高负荷的负荷值。

3.1.2.4 时序性及空间性 为了将数据可视化与节点状态的时序性结合起来，我们拟通过日历图的热力排布来展示一个服务器节点在过去一段时间内的运行状态。通过像素块颜色的深浅，直观反映某节点的健康程度。

由于缺少节点的地理位置信息，所以不能将集群的地理分布在地图上展示，前期拟定的三维数值坐标展示将只用树状结构叶子节点的热力色块来展示。因为使用者并不用清楚的知道此时此刻服务器的各种利用率参数，只需要知道该节点健康与否即可。

3.1.3 技术栈

Vue.js + express + python + R + webpack + mongoDB + mongoose + vuex + Promiss + D3.js + echarts.js + axios + L^AT_EX

3.1.3.1 编程工具及设备选择

Visual Studio Code(March 2019 version 1.33)

System:Windows 10 Intel(R) Core(TM) i5-8400 CPU 16.0GB 2.81GHz

3.1.3.2 前端框架

根据项目的特点，我们选用 `vue.js` 作为用户前端设计和实现的框架，具体的优势如下：

- 基于 MVVM 的双向绑定机制，使开发者在编写前后台参数传递的时候简洁易行，避免了过多的 `ajax` 请求来进行组件间与前后台的通讯，加快了用户在操作可视化视图时系统的处理速度；
- 拥有很多已整合的辅助模块，比如 `vue-resource`、`vuex`、`vue-d3` 等，对开发过程尽可能地简化，更有效率的进行数据请求、状态管理和可视化组件的开发；
- 还有其他如组件化、渐进式、异步批处理等优势，为节省文章篇幅在此不再赘述。

3.1.3.3 后端框架

根据项目的特点，采用了 `node.js` 的 `express` 作为后端框架，主要还是因为其具有如下优势：

- `Node.js` 比传统的后台处理语言更加快速，代码更加简洁
- 与后台数据库 `mongoDB` 的整合更加流畅，请求更加方便
- 更活跃的社区支持，使后期调试变得更有效率

3.1.3.4 数据库

因为系统实现的目标是实时监控服务器集群，所以计划采用后端连接数据库，用来储存数据并返回给后台所请求的数据。由于项目所采用的模拟数据集为 `csv` 文件格式，`mongoDB` 将会是我们的首选数据库：其非关系型数据存储格式完美适配

原数据文件特点，而且具有弱一致性，对于本项目的可视图形渲染一次请求大量数据的特点，一定程度上保证了处理和刷新视图的速度和用户访问速度。

3.1.3.5 可视化框架

在可视分析领域，有非常多种数据可视化的实现方式，类似 python、R 等在数据处理方面颇有建树的语言都有各自的第三方库来支持数据可视化。但是如果实现更自由的功能和表达性更强的视图，那最好的方式就是用专业的数据驱动文档语言 D3.js 来实现，不但拥有更全面灵活的组件、更活跃的社区，还能完美结合现有的 BS 架构，嵌入到已有的系统中。

3.2 模块设计

基于前端框架 vue.js 的模块化布局渲染的原则，加上出于未来对功能的拓展和维护有利的考虑，本项目自身将采用模块化开发。每一种不同的功能模块将被设计为不同的组件。各种组件各司其职，互不干扰，通过组件间的通信来实现系统与用户的交互。

与此同时，基于上文对整个系统的构思和对关键问题的分析，将对系统做如下的组件设计：模块一：在整个系统中，必须含有可以映射整个时间轴的数据总览模块，该模块作为整个用户 UI 界面的入口模块，是在整个时间线上对所有服务器节点状态的表示。由于拟向使用者展示所有异常服务器的变化趋势，在趋势的表达上，会使用一种折线图的方式来模拟每个时间戳上所含有的所有异常服务器节点的个数。在此组件里，用户可以根据图像所表达的趋势针对特殊时刻或者自己想观察的时刻，来过滤节点，同时根据传递的参数，刷新别的视图，从而达到整个页面的布局响应，即所谓的 Level-0 层次组件。

模块二：在 Level-0 级组件的基础上，我们将设计一种渐进式的响应模块 Level-1。该类模块是上一层模块的递进，通过使用者对上一层模块的操作，进一步向其展示更加深入的可视化视图。本层级拟设计两个视图，其一是分布式力向 (force-directed) 布局的树形视图，其二是拥有数量表示的旭日 (sunburst) 视图。以上是通过分布式任务调度的具体表达，从而来解释 job、task、instance 之间的从属关系，从而帮助判断造成异常的用户原因 (详细设计将在 3.2.1.2 中进行阐述)。

模块三：通过对 Level-1 级视图的操作，进而展示最顶端层级 Level-2。Level-2 是 Level-1 之上更细致更深入的视图层级，可以精确到具体的某个节点，从时序性和空间性两个角度展示其最一段时间的运行状态和相关的运行参数。拟用基于热力像素的日历图 (calendar) 图来表示某一节点在从现在到过去的状态描述，用直方图 (histogram) 来展示服务器节点本身的属性和运行参数，从而辅助 Level-1 层判断该节点的异常类型 (详细设计将在 3.2.1.3 及 3.2.1.4 中进行阐述)。

3.2.1 总览曲线 (Level-0)

3.2.2.1 基本设计

基于从时序性的角度展示服务器节点异常个数的总览曲线，将对该视图作如下设计：横轴代表时间 (原数据文件 `time_stamp` 字段)，纵轴代表该时间点下异常节点的个数 (<4000)，用曲线的拐点来表示异常个数，然后再将所有的拐点连结起来形成一条无断点的 DAG¹ 图形。由原数据集的特征可以得出，横轴的控制范围是 (100000, 172000) 之间，单位为秒，为期一整天的记录；纵轴是 (0, 4000) 之间的异常节点数，随着时间的递进做着规律性变化。

3.2.2.2 时序性导致的自变量密集度过高的问题

由于本次项目所取的数据是原数据第二天的记录，时间戳记录从 100000 到 172000，每十秒一个记录，所有对自变量来说有 7000 余个记录，映射在时间坐标轴上就是 7000 余个坐标点，远远超过了常规视图中的基数。这就导致了一个问题，当操作者想在一台长 1980px 的显示设备上看到整个曲线，由于时间点的密集度过高，所呈现出来的视图远远脱离了折线的特征，整条曲线的记录过分重叠，使得人眼无法辨认和映射每个时刻对应的异常节点的个数，更不要说去观察曲线的变化趋势了 (如图 3.1 所示)

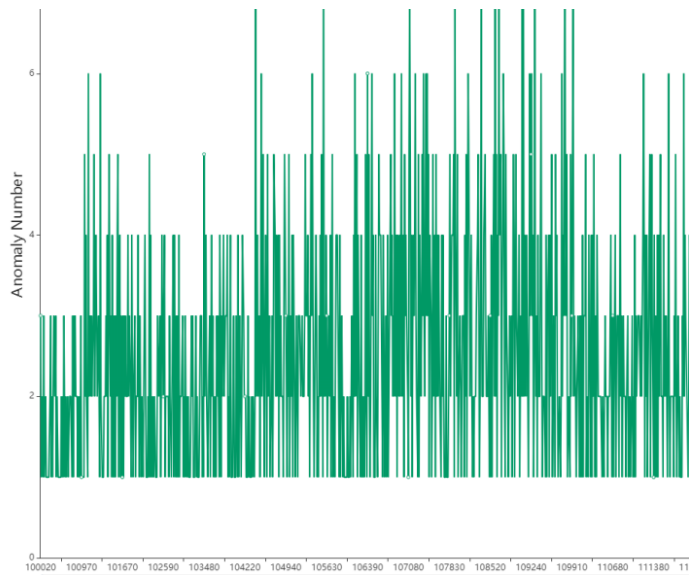


图 3.1: 记录过于密集的曲线

为了解决以上问题，经过资料的查阅，拟定出来以下方法来协助我们观察密集度过高的曲线：在时间轴的下方插入一个范围控制条，来控制我们向观察的区域大小。由于我们往往不需要观察所有节点的轨迹，当我们想看到某一小段时间内的折现变化情况，就可以通过改变控制条的长短，来决定视图所要显示的范围大小。

¹Directed Acyclic Graph，指的是一个无回路的有向图。

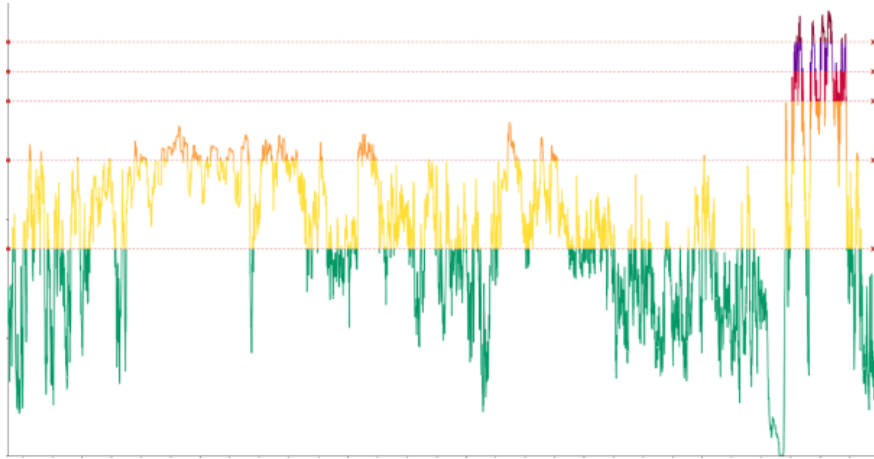


图 3.2: 优化后曲线视图

如图 3.2所示，通过控制下方控制条的长短，可以有效减小原视图折线的密集度，从而可以很清楚的展示异常节点的数量变化规律，从而帮助操作者更加有效的观察全程的变化趋势从而挑选有研究意义的时刻。

3.2.2 节点异常检测 (Level-1)

3.2.3.1 基本设计 想要通过视图体现出任务调度中各个层级 (job,instance,task) 之间的关系，基础元素拟采用树形结构的节点集，一个节点集表示中心父节点 job 下的 task 与 instance 的分布情况，结构示意图如图 3.3所示：

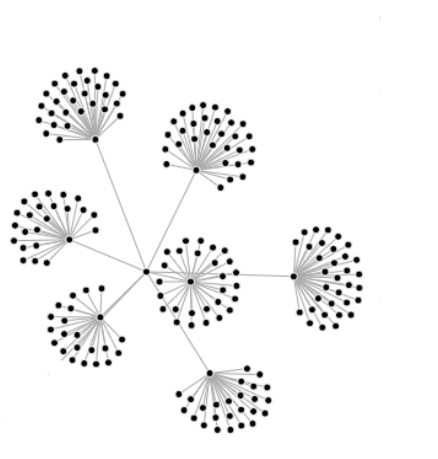


图 3.3: 树形节点结构图

我们将基于 Level-0 组件设计渐进响应组件 Level-1。这种类型的模块是前一个模块的渐进模块，并通过用户在前一个模块中的操作进一步显示更详细的图形视图。此级别旨在设计两个视图，其中一个是强制定向布局中的树，另一个是具有强响应函数的旭日形图。所有这些都是通过分布式任务调度的具体表达来解释作

业，任务和实例之间的联系，从而有助于确定用户导致负载异常的原因。异常定位的主要处理模块如图 3.4所示。这种节点集分为三个级别。有必要通过视图反映任务调度中每个级别（作业，实例，任务）之间的关系。基本元素旨在采用树结构的节点集，每个集合逻辑地表示中心父节点作业下的任务和实例的分布。为了视图的直观性和后续交互的可操作性，我们将不同层级的节点分别设置参数，以便使用者更好的区分：

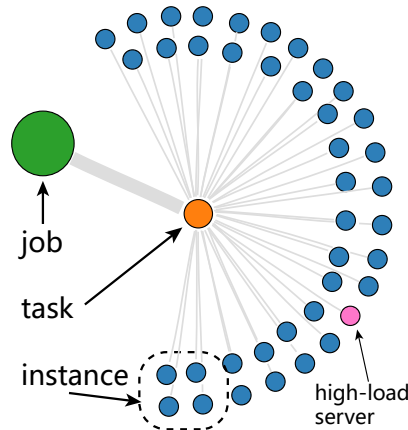


图 3.4: 设计展示

job 节点的半径为 8px, 颜色用十六进制表示为: #2ca02c;

task 节点的半径为 5px, 颜色用十六进制表示为: #ff7f0e;

instance 节点的半径为 3px, 颜色用十六进制表示为: #1f77b4。

在此基础上，如果我们想要给视图元素添加与运行 instance 的服务器节点的健康状况，拟定将环绕 task 的 instance 节点群增加颜色维度，渲染的色块表示该 instance 所在服务器的异常与否。这样我们就可以得到任务调度与服务器节点的关系，通过观察异常节点的分布和 job 簇之间的关系，得到出现异常的原因。

重点是反映视图组件中的异常服务器节点，除了作业调度的显示。如上面的图 2 所示，每个任务实例由一个独立的服务器机器独立处理，我们显示处理任务实例的服务器节点的负载级别。这种方法似乎是随机的，但通过图形方法，操作员可以获得异常服务器节点布局的规律，并进一步了解一些异常作业节点。

3.2.3.2 数量级的精确表示

经过以上分析，基本确立了实现系统功能的主视图的设计，虽然在表示上清晰明了，但是其中的元素还缺少一种对任务调度的精确刻画。所以出于功能完整性的考虑，在保留原视图的基础上，增加 Level-1 层级额外的可视化组件，如图 3.5 所示：

以上组件在可视分析领域被称作旭日 (sunburst) 图，在保证了层级关系的基础上，还增加了对节点元信息的描述，比如可以输出原组件不能输出的调度参数的

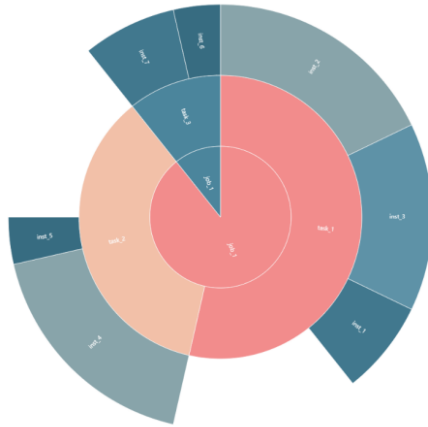


图 3.5: 旭日图模型

Id 信息，每个 task 所包含 instance 的数量级以及各层级调度元素占有所有元素的占比的多少，并且交互性强，更加全面客观的反映 job 簇的精确信息。

3.2.3 时序性展示 (Level-2)

为了增强时序性的展示，拟加入可以体现过往状态的可视化组件来实现对服务器节点的全面描述。我们挑选日历图 (calendar) 的原因是其功能和特点完全可以还原原本的设计思路：用横坐标和纵坐标两个维度来表示过去时间的延续性，用中央色块的颜色变化来展示在某一时刻下该节点运行的健康程度。

颜色表示的原理是基于生物学的人体视觉通道，红色的饱和度越大表示该节点在此时的异常程度更低。在时间的表示上，我们以 10 秒为一个递进单位，来展示服务器节点在某时刻之前 1 小时之内的所有运行状态，试图从中寻找规律，判断该节点的异常周期或者常规的健康状况。

上面显示的图形方法适用于整个时间线中的所有后端服务器节点。然而，当操作员想要观察单个节点的状态时，它们似乎并不那么令人信服，并且所谓的 1 级和 2 级组件仅在同一时刻说明某个时刻的状态。我们添加新组件以增强时序描述，并丰富我们从上一层视图中选择的指定异常节点的描述。我们打算通过加热图描绘过去一段时间内某个服务器的状态，并通过一种直方图显示机器的各种操作参数。通过热图和直方图的两个组成部分，我们使特殊节点的顺序描述更加全面和详细。

图 3.6 描述了单独节点状态的变化，颜色块的差异说明了过去 60 分钟内服务器性能的状态。每个块表示过去一段时间内的 10 秒的跨度，并且由所有块组成的区域表示一小时的时间线。视图组件通过可视通道的映射在时间上向操作员示出单个服务器节点的性能。具有低饱和度的那些块表示服务器的硬件的良好性能，并且那些深红色的块表示服务器的性能在目前具有高负载值的硬件时并不是那么优

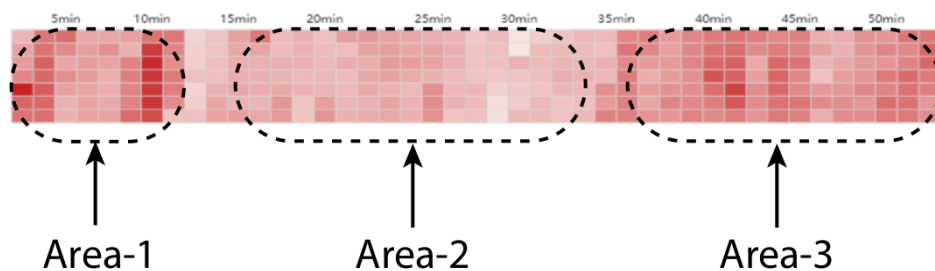


图 3.6: 日历图模型

越。

例如，图 3.6 中的 Area-1 说明了服务器 A，例如，在过去的 15 分钟内处于高负载状态，尤其是在您选择观察之前的大约 14 分钟内。然后在 17 分钟之后，服务器运行状态的性能保持在正常水平，如 Area-2 所示，之后，在 Area-3 中，服务器以某种方式具有另一个高负载阶段。

此级别的视图组件通过直观的性能抽象增强了特定服务器节点的时间可伸缩性，并使用另一个视图组件直方图增强了节点的详细描述，该直方图将与下一章中的其他模块一起进行分析。

3.3 模块实现

3.3.1 客户端与服务端的搭建

需要先设置整个项目的目录结构，只有完整简洁的项目结构，才能支撑起整个系统错综复杂的请求。整个项目的文件结构如下：

```

├── build
├── config
├── node_modules
├── server
│   ├── db# 数据库连接与配置
│   ├── routes# 后端路由文件
│   ├── views# 视图文件
│   └── app.js# 服务器入口文件
├── src
│   ├── App.vue# 视图根节点
│   └── main.js#webpack 打包入口文件

```

```
| |— components# 项目组件
| |— router# 前段路由
| |— store# 全局状态管理
|— static# 静态资源
|— index.html# 项目搭载 html 文件
|— package.json# 配置列表
|— README.md# 项目解释文档
|— 项目日志
```

- “build” 和 “config” 文件夹里存放项目所需要的配置项
- “node_modules” 文件夹存放项目所需要的第三方依赖
- “server” 文件夹存放后台服务端的代码文件
- “src” 文件夹存放前段客户端的代码文件
- “static” 存放静态文件和测试数据

服务端入口文件为 `app.js`，声明 `express` 后端框架，设置本地服务器端口，引入数据库并且发送请求测试后台是否连接成功。配置代码与后端接口页面显示如下：

Listing 3.1: 配置前后端服务器

```
const metaModel = require( '../db/model/meta' );
const usageModel = require( '../db/model/usage' );
const resultModel = require( '../db/model/result' );
const result = require( '../db/model/result' )

module.exports = function(app){
    app.get( '/', function(req, res){
        res.send( 'Hello , here is ' )
    });
};
```

图 3.7 证明后台已经搭建成功，成功向后台返回信息。

在后端搭建成功的基础上，我们将利用 `vue` 语法实现第一个前端展示页面：在 `components` 中创建模块 `test.vue` (以 `vue` 为后缀名的是 `vue` 中声明的组件文件)，想要在项目结构中调用该文件，必须首先将组件挂载到根节点 `App.vue` 中，在再服务端入口文件中引入根节点，在 `webpack` 打包后完成对项目结构的封装，即可在前端页面展示

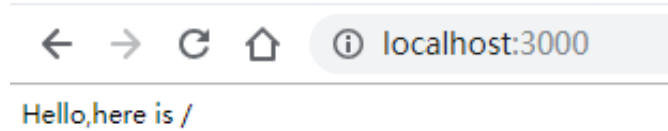


图 3.7: 连接成功

3.3.2 总览曲线 (Level-0)

为了达到简洁健壮的动态 web 设计的要求，实现曲线组件之前需要统筹组件本身与其他模块之间的数据流向工作。设计结果的规划如下：BS 架构的前端技术框架为 vue.js，在框架中用 vue 组件的形式去实现可视化曲线的模块，通过异步请求数据向本地服务器端发送数据请求，经过 mongoose 的查找和筛选之后返回给前台所要求的数据集，再通过 d3.js 或者 echarts.js 将数据渲染到前端组件中，即完成了一个可视化模块的实现。

Listing 3.2: 挂载服务器 app

```
const express = require('express');

const routers = require('./routes/index.js');
const metaModel = require('./db/model/meta');
const usageModel = require('./db/model/usage');
const resultModel = require('./db/model/result')
require('./db/db');
const app = new express();
routers(app);
app.listen(3000);
```

Listing 3.3: 测试 vue 渲染 DOM 元素

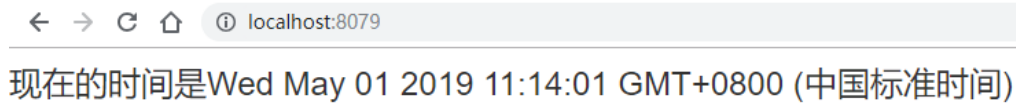
```
<template>
<div>
<h1>{{message}}</h1>
</div>
</template>

<script>
```



```
export default {
  data () {
    return {
      date: '',
      message: ''
    },
  },

  created () {
    let now = new Date();
    this.message = `现在的的时间是${now}`
  }
}
</script>
```

A screenshot of a web browser window. The address bar shows 'localhost:8079'. The main content area displays the text '现在的的时间是Wed May 01 2019 11:14:01 GMT+0800 (中国标准时间)'.

← → ↺ ⬆ ⓘ localhost:8079

现在的的时间是Wed May 01 2019 11:14:01 GMT+0800 (中国标准时间)

图 3.8: 测试结果

如图 3.8所示，vue 组件中的元素已经通过全局引用成功挂载到了服务端。至此，前后端的搭建工作已经顺利完成，接下来的章节将介绍每个可视分析组件的具体实现过程。

3.3.3 分时任务调度展示 (Level-1)

在实现可视化功能之前，需要对数据结构和视图代码做一些准备。设定将渲染曲线的数据集 Array 设计成二维数组的结构。但由于原数据文件的个体过大，为了优化查找速度，我们将原数据先做一层处理，以得出任务调度的服务器节点的异常情况 (用 boolean 值表示)。

对异常域的判定，我们将采用加权模拟的算法，将整个故障占比稳定在 10% 左右，根据对三个关键参数 (util_CPU、util_memory、util_disk) 的计算，得出判定 machine 是否异常的布尔值，再对 mongoDB 数据库表结构加以重构，得出我们将进行进一步统计的数据集。

我们的数据层任务是得到每一个时刻异常服务器节点的总个数，所以还要对数据进行加工。原生 JavaScript 算法可以实现对目标数据集的聚合，但更有效率的实现方法是利用 mongoose 的聚合管道，语言优美并且功能性更强大，如关键代码段 3.4 所示：

Listing 3.4: 管道实现聚合查询

```
.aggregate([
  {$match:{warning:{ $eq:1}}},
  {$group:{_id:'$time_stamp',warning:{ $sum:1}}},
  {$sort:{'_id':1}}
])
.then(d=>{
  res.json(d)
})
```

得到所需要的信息之后，还需要对数据进行重构来匹配可视化组件。在保证处理速度的前提下，这里我们用最经典的 js 代码来操作并重构数组，得到符合要求的数据集之后，需要我们来编写真正的可视化组件了。在上一章提到过，我们需要加入控制条来对要展示的有效数据进行筛选，从而是整个视图变得更加合理和美观；其次，我们需要加入更多的效果控件来对用户体验进行优化：

- 实现鼠标滚轮来带动页面的缩放；
- 实现健康状态对应视觉通道的色块图例，以及在视图中根据对数据的绑定来显示颜色分类，增强判别能力；
- 实现鼠标 hover 状态来展示对应的时刻数据以及异常节点个数；
-

时序曲线的实现模块 linec，被项目根节点调用并声明，通过 webpack 打包输出在 web 浏览器上，效果如图所示：

如图 3.9 展示的是在 116250 时刻到 120150 时刻的异常节点个数的时序变化，可以看出在图中峰值是 119110 秒时异常个数最多的 2722 个，最低的是 118380 时刻的 1084 个异常节点数。可见我们的设计对全过程趋势的表达效果良好，每个时刻的状态颜色映射较清晰。



图 3.9: 入口视图效果图

3.3.4 具体节点参数展示 (Level-2)

3.2.3.1 力导向树形图 在输入数据集准备可视化渲染之前，还需要对输入的数据集进行研究。储存在数据库中记录任务调度的原生数据是按条目划分的 JSON 数组，每一条记录含有 `inst_name`, `task_name`, `job_name` 等字段名，我们需要将其经过处理最终输出我们可以渲染的形式，实现的算法如下：

Listing 3.5: 格式化 row data

```
...
let exist = new Array();
let index = new Array();
let group = new Array();
for(let j in Json){
  if(exist.indexOf(Json[j].inst_name) == -1){
    index.push(Json[j].inst_name);
    group.push(1);
    exist.push(Json[j].inst_name);
  }
}
...
```

如代码段 3.5 所示，我们将数据处理为单位 job 簇是具有一个根节点 (job)、多个二级子节点 (task) 和若干个叶子结点 (instance) 的格式，某一时刻下包含多个 job 簇，组成在这一时间点下任务调度的可视化视图。展示实现之后的效果图，下图是在时间戳为 120010 时刻的任务调度可视化分析图：

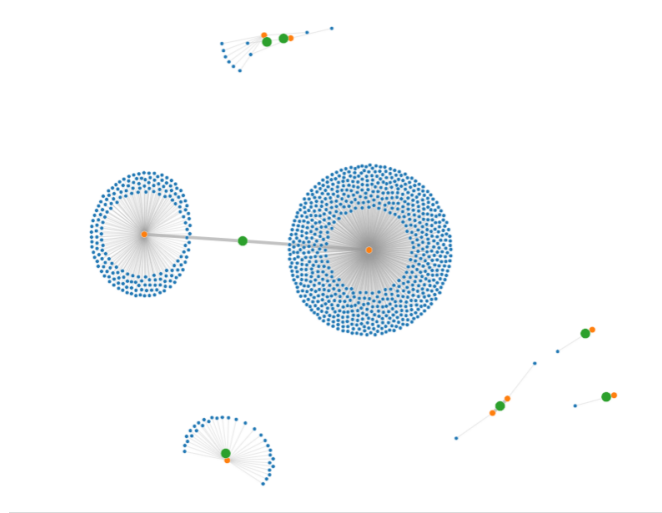


图 3.10: 力导向图效果图

如图 3.10所示，根据 job 级节点的渲染格式为 #2ca02c，节点半径最大来观察，得出该时刻共有 7 个 job 簇：左下角是 job 和 task 数目为 1、instance 数目为 32 的节点集合，出于视图中间、体积最大的是有 1 个 job 根节点，2 个次级 task 节点和共 1851 个的 instance 节点。

整个布局采用力向布局，拥有节点数越多的集合对其他单位的斥力也就越大，这样很好的保证了图形的可观察性和布局的美观，D3.js 画布元素选择与力向布局的参数设置的部分代码如图所示：

Listing 3.6: 基于 D3 关于 svg 图形的实现

```
d3() {
  let svg = d3.select("svg"),
  width = +svg.attr("width"),
  height = +svg.attr("height");

  let color = d3.scaleOrdinal(d3.schemeCategory10);

  let simulation = d3.forceSimulation()
    .force("link", d3.forceLink().id(function(d) {
      return d.id;
    })).distance((d)=>{
```

```

return d.value == 0.25 ? 70:10;}
))
.force("charge", d3.forceManyBody().strength(-5))
.force("x", d3.forceX())
.force("y", d3.forceY())
.force("center", d3.forceCenter(width / 2, height / 2));

```

3.2.3.2 节点健康状态的关联查询展示

根据 usage 数据文件中的 warning 字段进行关联查询，但是由于 D3 视图文件和 usage 文件不再一个表内，而 mongoose 聚合管道关联查询所得的数据集由于太过庞大超过了后阿提数据接口所能容纳的最大限度，采用经过讨论拟采用以下算法：

Listing 3.7: 基于 vuex 对数据的集中式管理

```

usageModel
.where('time_stamp').equals(req_time)
.then(response => {

let warningArray = new Array();
for (let i in warningId) {
for (let j in response) {
if (warningId[i][0] == response[j].machine_id) {
let obj = new Object();
...
obj.inst_id = warningId[i][1];
obj.warning = response[j].warning;
warningArray.push(obj)
break
}}}
o.warningArray = warningArray;
res.json(o);
})
...
context.commit('getdata', res.data.result);
context.commit('getarray', res.data.warningArray)
...

```

构造数据映射列表使数据在渲染的过程中在内置 `vuex` 中检查状态，根据结果对节点颜色进行修改，即可实现节点健康状态的关联展示。

3.2.3.3 旭日图

为了增强视图的可表达性和交互的操作性，在保留 Level-1 级视图中第一组件功能的完备性的前提下，增加另一组件旭日图，来表达关系调度的层级关系和各单位数量关系。

首先，先设计算法以格式化数据：以 `d3.js` 所带的 `d3.stratify` 函数对上一节的处理结果进行层次化处理，但由于该函数的机制原则是只能包含一个根节点，所以在数据中先构造一个辅助根节点“`origin`”，在调用 `root.descendants()` 取父节点的所有后代包含关系，再设计算法使得结构型树形结构变为扁平型数据结构。文字描述算法的复杂性略有抽象，下图是该思想的部分实现算法辅助说明：

Listing 3.8: 构造父子节点结构数据

```
let links = linksProcess(a);

let root = d3.stratify()
  .id(d => {
    return d.target
  })
  .parentId(d => {
    return d.source
  })
  (links)
let d = root.ancestors()[0].children;;
let data = structure(d)
```

所以将每次得到的关系数据集送入该函数群输出结果，再将结果送到已经设计好的 `d3` 可视化代码中，即可实现对旭日图的渲染，如图 3.11 所示：

从图中即可更有效的了解到 `instance` 节点的数目信息，以及显示各个层级元素的具体名称，以便通过交互在 Level-2 视图中查看。

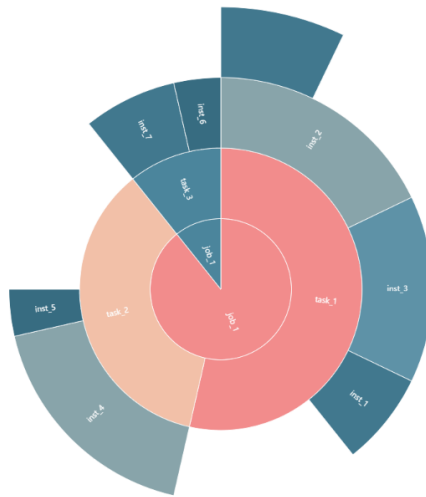


图 3.11: 旭日图效果图

3.3.5 时序性展示 (Level-2)

在描述整个调度元素的层级 Level-1 上, 我们增加针对独立服务器节点的表达模块, 称作 Level-2 层级。在 Level-2 层级上, 我们可以更全面系统的看到某个节点的细节信息。

仅仅满足了任务调度的空间性描述还远远不够, 如果能在时序性的刻画上帮助使用者了解到服务器节点的过往情况, 往往能更加全面的判断节点的故障类型, 甚至找出异常性的周期规律。可见, 增加视图的时序性是有必要的。

对二维模式 (时间-状态) 的刻画上, 为了符合可视分析原则之一的直观性, 我们选择像素排布的模式, 对服务器节点所组成的空间阵列, 用视觉通道的映射来实现对健康状况的表达。

由于监视器测得的数据的时间间隔是以 10 秒为一个单位, 在 10 秒级的间隔来说, 节点状态的变化往往是趋于线性的, 所以我们拟以 10 秒为刷新单位, 将过去 1h 之内的状态信息渲染于已经设定好的像素中。

根据热力图显示的原理, 每一个色块使用基于红色的渲染, 用色相的饱和度与色彩的明暗来描述服务器节点的状态, 实现视图如下:

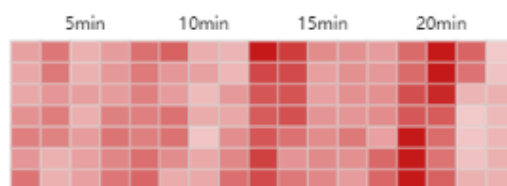


图 3.12: 日历图效果图

该模块表示了 0-20min 的区间内，服务器节点状态的描述，由图可见，红色越深代表此时该节点的负载越高，红色越浅则表示节点的负载越低，性能越好。色块的布局基于对时间的排布，可见该表示方法对向使用着输出节点状态的时序性的效果还是非常好的。

3.3.6 交互实现

为了增强系统的交互性，优化使用者的用户体验和加强整个系统的功能性，我们拟将更多的交互手段加入系统，让使用者可以通过对其中一个组件的操作，来带动全部组件的刷新效果，使所有组件之间更具有整体性。这时便突出了之前我们使用 Level 来定义组件层级的逻辑价值。

如我们在上文计划的那样，最基础的逻辑层 Level-1，是整个系统的入口元素，也是我们进行用户操作的最初始的元素，通过该组件的特点，使得我们对全局的把控更加具有针对性，所以 Level-0 层级组件也适合作为入口元素关联其他组件来实现视图的联动效果。

众所周知实现组件之间的传值是一件非常繁琐的工作。在 vue 中需要不断地调用 props 来对你要传递的参数进行操作，而且针对于本次项目组件之间的关联度非常大的情况下，这种方法在后期调试和维护时难度将会非常的大。在这种需求的驱动下，我们计划将采用 vuex 来实现对全局状态的管理和维护。

由于篇幅原因不再过多解释 vuex 的概念及其工作原理，将直接说明其调用的过程。首先在原曲线组件代码中增添点击事件，事件中全局调用 vuex，将 vuex.\$store.state 的 time_stamp 值更新为点击元素的 params。在状态管理层的 state 层级实现的代码结构如下：

Listing 3.9: state 层级结构

```
let state = {
  time_stamp: '',
  data: [],
  counter: 0,
  warningArray: []
};
```

在 mutations 实现的代码结构如下：

Listing 3.10: mutations 层级结构

```
let mutations = {

submitTime(state, name) {
state.time_stamp = name
},
getdata(state, d) {
state.data = d;
},
getarray(state, d) {
state.warningArray = d;
}
};
```

在 actions 实现的代码结构如下：

Listing 3.11: actions 层级结构

```
let actions = {
getData(context, d) {
context.commit('submitTime', d);
Vue.http.get('/result?name=${d}')
.then(res=>{

context.commit('getdata', res.data.result);
context.commit('getarray', res.data.warningArray)
});
}
};
```

以上代码的区别在于，`state` 是全局状态变量声明，而 `mutations` 是对 `state` 进行操作的控件，而 `actions` 中含有各种异步操作。所以在点击事件触发后，是利用 `axios` 向后端数据接口发起数据请求，将时间参数通过 `axios` 传递到后台，在后台进行 `mongoose` 的数据处理过程，然后将处理得到的结果输出到接口，再通过 `vuex` 调用 `mutations` 控件改变全局 `data` 数组的值，即可实现对 `state` 的更新，以便其他所有 `vue` 组件在需要时通过 `this.$store.state.data` 来调用全局状态 `state`。

实现了这种功能之后，在组件中对全局状态的渲染也就水到聚成了。通过监听函数 `watch()` 来对 `state` 的刷新保持监听模式，一旦数据发生更新，便触发对可视化

模块的重新渲染以达到刷新的效果。

通过全局状态管理模式的另外一个好处就是，所需要被渲染的数据经过点击只需要请求和发送一次，在其他组件请求的过程中，不需要再发送 axios 请求，而是直接使用 vuex 的全局状态来保持对视图的更新。在一定程度上优化了视图的处理速度，对这种一次请求 1M 以上数据对数据请求要求十分高效的项目来说，这种改变是有必要的。

3.4 性能测试与评估

你好

参考文献

- [1] J. Al Dallal, P. Sorenson, “System testing for object-oriented frameworks using hook technology” , Proceedings 17th IEEE International Conference on Automated Software Engineering, 2002.
- [2] Xiaowei Qin, Shuang Tang, Xiaohui Chen, Dandan Miao, Guo Wei, “SQoE KQIs anomaly detection in cellular networks: Fast online detection framework with Hour-glass clustering” , China Communications, pp.25-37, 2018.
- [3] Pei Yang, Weidong Jin, Peng Tang, “Anomaly Detection of Railway Catenary Based on Deep Convolutional Generative Adversarial Networks” , 2018 IEEE 3rd Advanced Information Technology, Electronic and Automation Control Conference (IAEAC), 2018.
- [4] Daojing He, Sammy Chan, Xiejun Ni, Mohsen Guizani, “Software-Defined-Networking-Enabled Traffic Anomaly Detection and Mitigation” , IEEE Internet of Things Journal, 2017.
- [5] Vania Bogorny, Shashi Shekhar, “Spatial and Spatio-temporal Data Mining” , 2010 IEEE International Conference on Data Mining, 2010.
- [6] Hetal Thakkar, Barzan Mozafari, Carlo Zaniolo, “A Data Stream Mining System” , 2008 IEEE International Conference on Data Mining Workshops, 2008.
- [7] G. Williams, R. Baxter, Hongxing He, S. Hawkins, Lifang Gu, “A comparative study of RNN for outlier detection in data mining” , 2002 IEEE International Conference on Data Mining, 2002. Proceedings, 2002.
- [8] A. R. Jakhale, “Design of anomaly packet detection framework by data mining algorithm for network flow” , 2017 International Conference on Computational Intelligence in Data Science (ICCIDS), 2017.

- [9] Alireza Tajary, Hamid R. Zarandi, “An Efficient Soft Error Detection in Multi-core Processors Running Server Applications” , 2016 24th Euromicro International Conference on Parallel, Distributed, and Network-Based Processing (PDP), 2016.
- [10] Chihiro Sakazume, Hiroyuki Kitagawa, Toshiyuki Amagasa, “DIO: Efficient interactive outlier analysis over dynamic datasets” , 2017 Twelfth International Conference on Digital Information Management (ICDIM), 2017.
- [11] Nan Cao, Chaoguang Lin, Qiuhan Zhu, Yu-Ru Lin, Xian Teng, Xidao Wen, “Voila: Visual Anomaly Detection and Monitoring with Streaming Spatiotemporal Data” , IEEE Transactions on Visualization and Computer Graphics, pp. 23 –33, 2018.
- [12] Cheong Hee Park, “Anomaly Pattern Detection on Data Streams” , 2018 IEEE International Conference on Big Data and Smart Computing (BigComp), 2018.
- [13] Li Xinran, “Research on massive data 3D-visualization” , 2012 IEEE International Conference on Computer Science and Automation Engineering, 2012.
- [14] Y.B. Luo, B.S. Wang, Y.P. Sun, B.F. Zhang, X.M. Chen, “FL-LPVG: An approach for anomaly detection based on flow-level limited penetrable visibility graph” , 2013 International Conference on Information and Network Security (ICINS 2013), 2013.
- [15] Hongbin Xia, Wenbo Xu, “Research on Method of Network Abnormal Detection Based on Hurst Parameter Estimation” , 2008 International Conference on Computer Science and Software Engineering, 2008.
- [16] Zhixin Sun, Jin Gong, “Anomaly Traffic Detection Model Based on Dynamic Aggregation” , 2010 Third International Symposium on Electronic Commerce and Security, 2010.
- [17] Yong Li, Wei-Yi Liu, “Backward probabilistic logic reasoning algorithm for decision problem with Conditional Event ALgebra on Bayesian networks” , 2008 International Conference on Machine Learning and Cybernetics, 2008.
- [18] Josef Kittler, William Christmas, Teófilo de Campos, David Windridge, Fei Yan, John Illingworth, Magda Osman, “Domain Anomaly Detection in Machine Perception: A System Architecture and Taxonomy” , IEEE Transactions on Pattern Analysis and Machine Intelligence, pp.845-859, 2014.