2018 年英特尔杯大学生电子设计竞赛嵌入式系统专题邀请赛

2018 Intel Cup Undergraduate Electronic Design Contest

- Embedded System Design Invitational Contest

作品设计报告

Final Report

Intel Cup Embedded System Design Contest

报告题目：基于容器的 mini 云平台

学生姓名：康益菲、廖东、屈彬

指导教师：伍卫国

参赛学校：西安交通大学

2018 年英特尔杯大学生电子设计竞赛嵌入式系统专题邀请赛

参赛作品原创性声明

本人郑重声明：所呈交的参赛作品报告，是本人和队友独立进行研究工作所取得的成果。除文中已经注明引用的内容外，本论文不包含任何其他个人或集体已经发表或撰写过的作品成果，不侵犯任何第三方的知识产权或其他权利。本人完全意识到本声明的法律结果由本人承担。

参赛队员签名：

日期：　　年　　月　　日

基于容器的 mini 云平台

摘要

嵌入式平台有功耗低、单价低、体积小等优点，而容器技术虚拟损耗低，启停快。因此，我们设计实现了一个基于嵌入式平台和容器技术的 mini 云计算平台，平台具有功耗低，构建容易，鲁棒性强等优点。

针对大量嵌入式应用有较长时间是非高负载的，存在资源空闲的现状，我们基于 mini 云平台实现了一个共享算力平台。我们设计实现了一种基于 GBRT 的可以预测节点资源占用率的调度器，实现了计算任务对节点用户的透明。

关键词：云计算平台，容器技术，共享算力

THE MINI CLOUD COMPUTING PLATFORM BASED ON CONTAINER

ABSTRACT

The embedded platform has the advantages of low power consumption, low unit price, and small size, and the container technology has low performance loss in virtualization and quick start and stop. Therefore, we designed and implemented a mini cloud computing platform based on embedded platform and container technology. The platform has the advantages of low power consumption, easy construction, and strong robustness.

For a large number of embedded applications that have a long period of time is not a high load, and there are many idle resources, we have implemented a shared computing platform based on the mini cloud platform. We have designed and implemented a scheduler based on GBRT that can predict the occupancy rate of node resources. So it is transparent to node users when realizes the shared computing task running.

Key words: cloud computing platform, container technology, shared computing

目　录

第一章 绪论

1.1 背景介绍

随着科技的发展，自然科学越来越复杂，越来越多的科学研究需要大量的计算资源，例如：核聚变反应模拟，人类基因组计划，气动模型计算等等。单机的计算能力已经不能满足研究需求，云计算应运而生。云计算（Cloud Computing）是分布式计算（Distributed Computing）、并行计算（Parallel Computing）、效用计算（Utility Computing）、网络存储（Network Storage Technologies）、虚拟化（Virtualization）、负载均衡（Load Balance）、热备份冗余（High Available）等传统计算机和网络技术发展融合的产物。云计算将大量的服务器通过虚拟机技术虚拟为一个个计算资源节点，用户无需关心硬件的实现和维护，只需要在云端购买计算资源，即可快速的获取自己所需的资源。同时云计算还有快速伸缩等优点

现有云计算集群往往是由高性能服务器构建的。高性能服务器虽然性能很高，然而大部分高性能服务器耗电量高，热功率很大，需要专门设计的机房。电力费用、散热费用这些运维费用已经成为云计算集群成本的主要部分，谷歌、微软等企业都在冰岛兴建了自己的数据中心，以利用冰岛寒冷的气候降低机房因散热产生的昂贵的电力费用。

嵌入式平台广泛应用于我们的日常生活中。大到汽车，小到空调遥控器，都是基于嵌入式平台的应用。嵌入式平台有着低功耗，体积小等的优点，但是往往嵌入式平台性能都比较低下。但是近几年嵌入式处理器的性能有着很大的提高，平台扩展性也不断增强。Intel 公司在 2017 年推出了 UP2 Board 嵌入式平台，采用了 Intel 最新的奔腾 N4200 处理器+8GB RAM+64GB eMMC 配置，有 HDMI 和 DisplayPort 视频输出，双以太网端口，4 个 USB 3.0 接口，1 个 SATA 3 接口，1 个 mini PCI-e 接口，1 个 M.2 接口，具有非常强的可扩展性。其处理器性能比上一代处理器性能提高 33%，而 TDP 只有 6W。因此高性能低功耗的嵌入式平台可以应用于很多新的方面，比如说构建云计算集群。

而高性能低功耗的嵌入式平台恰恰可以解决这些问题，嵌入式平台的功耗相对较低，发热量也比较小，而且嵌入式平台体积比传统服务器小很多，在同样的机房空间下会有更多剩余的空间，从而在相同的散热设备下，达到比传统服务器更好的散热效果。

同时，由于嵌入式平台单价低，可以轻易部署大量的计算节点，系统有更好的鲁棒性，提高了系统的容灾能力。因此，用嵌入式平台构建云计算平台相对于传统方式来说有更高的可靠性。

传统的云计算平台往往是基于虚拟机技术的，但是虚拟机镜像非常大，常常有数十 GB，而且虚拟机相比于宿主机性能损耗很大。嵌入式平台往往没有很大的硬盘空间，并且性能虽然较高，但是虚拟化之后，计算性能将严重降低，不再具有可观的实用价值。

容器技术是近几年新兴的一种虚拟化技术。Docker 是其中最具有代表性，也是应用最为广泛的开源容器技术。Docker 是基于 Go 语言开发的容器。Docker 依赖于 groups、namespaces 和 SELinux 等 Linux 核心技术。Docker 容器类似于一个轻量级的沙箱，为每个容器提供一个单独的用户态环境，以便在同一台机器上运行不同的应用程序。Docker 容器是资源分割和调度的基本单位，是一个跨平台、移植性强且易于安装使用的容器解决方案。它相较于虚拟机，其性能损耗很低，只是用户态的虚拟化。而且 Docker 镜像采用多层架构，大小只有数十 MB。Docker 的启停时间也很短，只有几秒，相比于虚拟机几分钟的启停时间，使用 Docker 可以使系统在有节点崩溃时更快的恢复工作，使系统有更好的容灾能力。

因此，相对于像 Vmware 这种传统虚拟机技术而言，容器技术更为适合嵌入式平台的虚

拟化，从而为构建云计算平台提供了基础。

综上，嵌入式平台相比于传统服务器有着单价低，低功耗，热功率低，体积小的优点。容器技术相比于传统虚拟机技术占用硬盘空间少，性能损耗低，更适用于嵌入式平台。因此，我们基于嵌入式平台和容器技术构建了一个新型 mini 云计算平台。

BOINC（Berkeley Open Infrastructure for Network Computing，伯克利开放式网络计算平台）是目前主流的分布式计算平台之一，由加州大学伯克利分校（University of California - Berkeley）计算机系于 2003 年发展出来的分布式计算系统。它本身设计成用于 SETI@home 项目，但逐渐在其他领域包括数学、医学、天文学、气象学等。BOINC 旨在为各研究者提供汇集全球各地大量个人电脑的强大运算能力。2018 年 6 月 BOINC 平均提供 19.901 千万亿次浮点运算/秒的计算能力。BOINC 迄今已经完成了十几个大规模科学计算项目，由此可以看出，共享计算平台有很好的应用前景。

高性能嵌入式平台已经深入到生活中，然而现有的应用，比如：智能摄像头、家庭网关、无线门禁系统，不能充分利用嵌入式平台的计算资源和网络资源。而且，它们并不是一直处在高负载状态，在用户不使用时设备会处于空闲的低负载状态。如此，大量的计算资源被闲置。针对这一现状，我们基于 mini 云计算平台开发了一个共享算力平台，以充分利用闲置的计算资源。这样，一方面闲置的设备得以利用，另一方面也为需要高性能计算的客户提供了廉价的计算资源。

1.2 主要工作内容

本文基于 Intel UP2 Board 平台和 Docker 容器技术设计并实现了一个 mini 云计算平台，并针对目前共享剩余算力的需求，开发了一个基于 mini 云计算平台的共享算力平台。

mini 云平台用多块 Intel UP2 Board 构成了计算集群，使用 Docker 容器技术对集群进行了虚拟化，并且实现了一个云平台管理系统。用户可以在管理系统中自定义 Docker 镜像，设计应用，管理应用。

共享算力平台在 mini 云平台的基础上，针对嵌入式设备具有空闲算力的特点设计了一个可以预测设备可用性的调度器，通过调度器来为计算任务实时分配资源。并且实现了一个共享算力平台管理系统，用户可以在管理系统中设计和管理自己的计算任务。

本文第一章简介了云计算平台、嵌入式平台和容器技术的相关背景和特点，并且介绍了主要工作内容和文章编排。

本文第二章详细地介绍了云计算平台和 Docker 容器技术的相关知识和特点。

本文第三章介绍了系统的设计和实现，首先介绍了系统的总体构成，然后介绍了系统的功能与指标。最后分模块介绍了模块的功能，架构和具体实现。

本文第四章介绍了对系统进行的测试。首先介绍了测试方案，然后分模块介绍了功能测试的过程与结果。

本文第五章对工作加以总结和分析，介绍了系统的优点和特色，并对未来工作加以展望。

第二章 相关技术介绍

2.1 容器技术介绍

2.1.1 Docker 简介

Docker 是近年来推出的较为成熟的开源容器引擎[1]，基于 Go 语言开发。Docker 最初是由 DotCloud 公司发起[2]，其源代码现在托管在 GitHub 上。Docker 依赖于 Linux 的一些核心技术，如 cgroups、namespaces 和 SELinux 等[2-4]。Docker 引擎是一个基于内核级的 Linux 容器技术，Docker 容器类似于一个轻量级的沙箱，Docker 为每个容器提供一个单独的环境，以便在同一台机器上运行不同的应用程序[4]。 Docker 容器是资源分割和调度的基本单位，是一个跨平台、移植性强且易于安装使用的容器解决方案[5]。

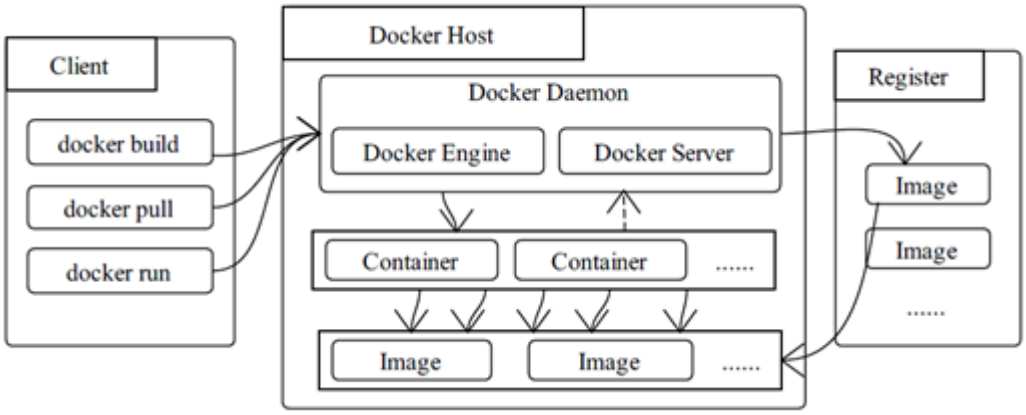


图 2-1 Docker 体系结构

Docker 体系结构如图 2-1 所示，用户通过 build、pull、run 命令管理镜像，在宿主机上运行镜像即可得到一个 Container 容器。

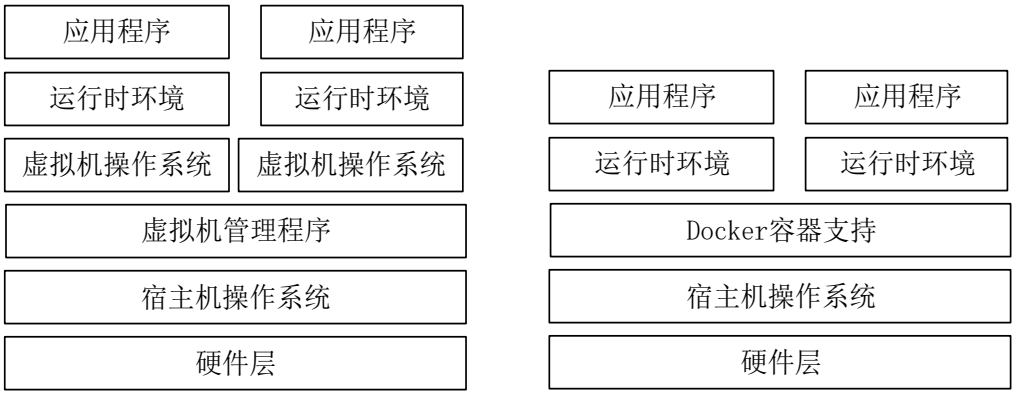
Docker 容器技术的发展带来了如下好处[5, 6]：

1) 可持续部署。Docker 为开发人员提供统一的镜像，并构建标准的开发环境，开发人员和测试人员可直接部署该软件镜像，简化了软件的部署和测试过程。

2) 良好的跨平台特性以及可移植特性。由于 Docker 镜像可以在任意安装 Docker 的平台运行，如今越来越多的云平台都支持 Docker。

4) 较高资源利用率。Docker 容器共享宿主机操作系统，从而降低了系统负载，在同等条件下，相对于传统虚拟机来说，Docker 可运行更多的应用程序，保证了软硬件资源利用率的最大化。

5) 应用镜像仓库。Docker 官方构建了镜像仓库，类似于 GitHub 的组织和管理形式。用户可方便的获取、发布、更新镜像。



a)传统的虚拟化方式　　b)Docker虚拟化方式

图 2-2 Docker 与传统的虚拟化方式比较

如图 2-2 所示比较了使用传统虚拟机和 Docker 来部署应用程序[7]，左图中在同一个虚拟机管理程序上部署不同的应用程序需要不同的操作系统或者不同的版本，例如 RHEL Linux、Debian Linux 等；而右图中运行应用程序的虚拟机操作系统和虚拟机管理程序被 Docker 容器所代替[8]，所有应用程序共享同一个操作系统，因此基于 Docker 部署应用程序要比传统的虚拟化方式部署应用程序更快捷更节省资源，可见采用 Docker 虚拟化的方式实现多个系统或应用同时运行在同一个主机上。由于 Docker 容器使用宿主机操作系统，因而重启容器并不意味着重启操作系统。本文提出的基于 Docker 创建和管理渲染集群可有效提高系统资源利用率，降低部署成本、便于维护和管理以及具有良好的容错容灾性能。Docker 镜像类似于虚拟机镜像，开发人员可以在文件中轻松地定制和打包应用程序的运行时环境。一旦创建了 Docker 镜像，用户就可以使用 Docker 引擎在任何主机上运行镜像。“构建一次即可在任何地方运行”的特性极大地简化了应用程序的开发、交付和部署。现在，越来越多的公司使用 Docker 构建了自己的平台[9]。Dockerfile 可以让用户管理一个单独的应用容器，本文基于 Dockerfile 的优势使用 Dockerfile 创建渲染镜像。

Docker API 最主要的用户是 Docker 自己，默认情况下 Docker 只允许通过 unix socket 通信操作 Docker daemon，也可通过 HTTP 调用其 Rest API，此时需要单独配置启动参数。采用 Docker API 以及相关结构化开发工具来联合编排 Docker 容器。通过安装 docker-py 实现对 docker 相关资源的操作，docker-py 主要提供了 Client 类，用来封装用户执行的各种 docker 命令，例如 build、run、commit、create_container、info 等接口，对 REST 接口的调用使用了 request 库。而对于这些 API，用户也可以通过 curl 进行调用测试，本文将使用 Docker API 实现渲染集群管理系统的开发。

2.1.2 Docker 编排技术介绍

编排（Orchestration）主要是来描述自动配置、协作、管理服务的一个过程。根据被部署的对象之间的耦合关系，以及被部署对象对环境的依赖，制定部署流程中各个动作的执行顺序。在 Docker 的世界里，编排用来描述一组实践过程，这个过程会管理运行在多个 Docker 容器里的应用，而这些 Docker 可能运行在多个宿主机上，因此在 Docker 应用场景中，编排意味着用户可以灵活地对各种容器资源实现定义和管理，编排的功能是复杂系统实现灵活可操作性的关键。

Compose 是 Docker 官方的编排工具，可以使用 YAML 文件来配置应用程序的服务，其负责对 Docker 容器集群的快速编排。如果说 Dockerfile 重现一个容器，那么 Compose 重现容器的配置和集群。Compose 允许用户通过一个单独的 docker-compose.yml 模板文件来定义一组相关联的应用容器作为一个项目。

Docker Swarm 是 Docker 公司发布的 Docker 集群管理工具，Docker 集群是指在多台已

经安装和部署了 Docker 的物理机上通过使用 Docker 容器来使用物理机的存储和计算资源。基于 Docker Swam 的分布式软件开发是云计算提出的一种新方法，它有巨大的提供多云开发环境的潜力，且无需担心其复杂性。Swarm 工具实现统一管理多个宿主机节点上的 Docker 镜像和容器，作为一个虚拟整体暴露给用户，方便用户管理并使用多台主机。

Docker Compose 便于多容器应用的部署，而 Docker Swarm 允许创建 Docker 容器集群。Swarm 通过扩展 Docker API 力图让用户像使用单机 Docker API 一样驱动整个集群，Swarm 对外提供完全标准的 Docker API，只需要理解 Docker 命令，用户就可以开始使用 Swarm 集群[34]。本文选择 Docker Swarm 有两个原因，首先它是开源的，并没有被锁定在任何平台上；其次，它是在同一个 Docker 守护进程中进行的。

第三章 系统设计

3.1 系统方案

本系统由 3 个模块构成，分别为平台管理模块，容器模块和调度器模块。其中容器模块是所有功能的基础组件，和管理平台模块一起构成了 mini 云平台系统。在两个模块的基础上我们根据共享算力平台的特点研发了调度器模块，三个模块一起构成了共享算力平台。

系统的功能有：用户可以进行自定义 Docker 镜像，即上传自定义的 Dockerfile 文件。用户可以创建 Docker 集群，即上传 YAML 文件来自定义运行自己镜像的 Docker 容器数量，网络配置，储存配置，启停依赖关系等。用户可以上传计算代码和数据文件，然后创建计算任务。系统会根据任务大小及数量自动将计算单元分发给各个节点，待计算完成后，用户即可下载计算结果。用户还可以管理 Docker 集群和计算任务。系统功能用例如图 3-1 所示

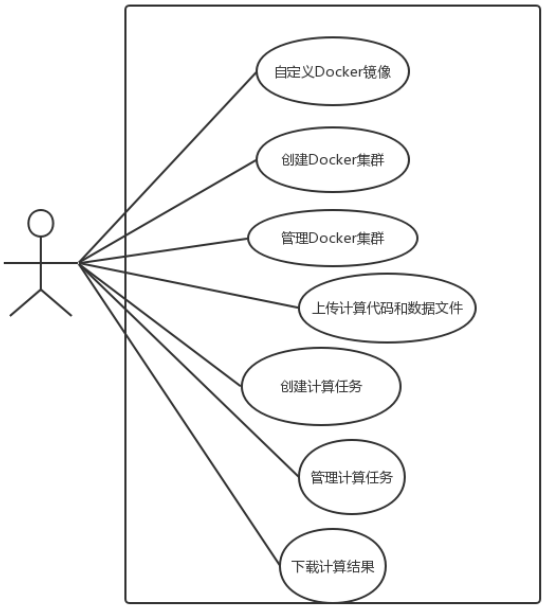


图 3-1 系统功能用例图

3.2 平台管理模块

3.2.1 平台管理模块设计要点

平台管理模块是与用户交互的核心，承担着数据展现以及与其他模块通信的重要任务，所以需要简洁清晰的 UI 设计，良好的交互设计；另外，管理模块也承担着用户权限控制、身份鉴定功能。于是，我们对平台管理模块的功能进行了细致的拆分，划分了以下几个子模块。

1) 登录管理。提供登录界面，用户通过分配的用户名、密码进行登录
2) 权限管理。区分不同用户的权限，根据用户权限的不同给予用户不同的操作权限，比如超级管理员有所有最高级的权限，普通用户只能控制属于自己的节点。
3) 系统概览。主要呈现系统的运行概况，展示活跃节点数目、流量、节点分布等等情况。
4) Docker 上传/分发。提供 Docker 镜像、Dockerfile 上传的接口，用户可以将具有一定功能的镜像上传到服务器，然后选择节点进行分发，被选中的节点将获取到该镜像，并在调度器的调度下运行。

5) 共享算力。该功能允许用户上传代码和文件，所有的节点共享同一份代码，并获取不同的数据文件，在调度器的调度下，执行程序，并将结果回传。

6) 节点管理。该功能提供节点管理的交互界面。用户能够看到节点的工作情况，分布地区、硬件设备等等信息，并能够控制节点的启\停，还能对节点进行批量的控制。

3.2.2 平台管理模块框架设计

该模块一方面有自己的独立的数据，比如用户管理，又需要与其他模块进行交互：调度器模块、容器模块，所以不光有 UI 的功能，还要有独立的后台管理这些数据，我们对比分析了各种架构方案，最终选择了前后端分离框架：Web 前端进行数据展示，与用户进行交互；后端与数据库和其他模块对接；两者通过预先定义的 REST API 进行数据交互，以 json 格式传递数据。图 3-2 是平台管理模块架构。

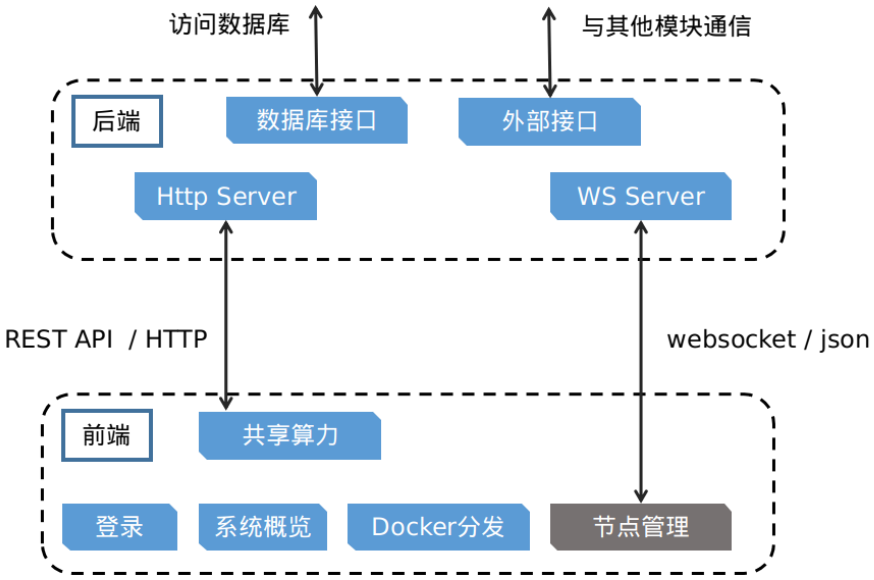


图 3-2 平台管理模块架构

最终，前端采用了 vue + css + html + javascript+ websocket 开发，后端采用 python + flask + redis + flask-restful + mysql + websocket 的方案。

前端：

1) Vue

Vue 是一套用于构建用户界面的渐进式框架。与其它大型框架不同的是，Vue 被设计为可以自底向上逐层应用。Vue 的核心库只关注视图层，不仅易于上手，还便于与第三方库或既有项目整合。另一方面，当与现代化的工具链以及各种支持类库结合使用时，Vue 也完全能够为复杂的单页应用提供驱动。

2) Websocket

HTTP 协议是一种无状态的、无连接的、单向的应用层协议。它采用了请求/响应模型。通信请求只能由客户端发起，服务端对请求做出应答处理。这种通信模型有一个弊端：HTTP 协议无法实现服务器主动向客户端发起消息。这种单向请求的特点，注定了如果服务器有连续的状态变化，客户端要获知就非常麻烦。大多数 Web 应用程序将通过频繁的异步 JavaScript 和 XML（AJAX）请求实现长轮询。轮询的效率低，非常浪费资源（因为必须不停连接，或者 HTTP 连接始终打开）。

因此，WebSocket 出现了。WebSocket 连接允许客户端和服务器之间进行全双工通信，以便任一方都可以通过建立的连接将数据推送到另一端。WebSocket 只需要建立一次连接，就可以一直保持连接状态。这相比于轮询方式的不停建立连接显然效率要大大提高。

后端：

1）python 语言以其优雅高效的特性，被全世界开发者青睐，正因如此，使其拥有一个巨大的开发者社区，第三方库在所有编程语言中也是最全面的之一。不论是 web 服务还是机器学习，或者是 ui 客户端都有用武之地，是一门名副其实的胶水语言。

2）python-flask 是基于 python 开发的一种轻量级 web 服务器，相比于 django 等框架，flask 更加灵活小巧。flask 基于插件扩展，需要的时候下载相应的插件即可，整个系统能够保持在“小而精”的状态，通常是小型项目的不二之选

3）flask-restful 是 python-flask 的一个插件，主要实现了 URI 的路由，方便实现 RESTful（表现层状态转化）风格的 API，用这种风格设计接口，会另 API 更加明晰，是业界当前的主流设计方法。

4）python-websocket 是采用 python 实现的 websocket 服务端，主要实现 3 个接口，分别是：client_connected()、client_left()、messageArrived()，对应客户端连接、客户端离开、消息到达这三个事件。基于这三个 API，我们可以很方便地设计自己的通信、控制逻辑

5）gevent 是基于 epoll 的协程库，当 python 程序在进行 I/O 等待时，能够自动切换运行其他指令，直到 I/O 完成，所以 gevent 能极大提高 python 网络程序的效率。另外因为 python 的线程有 PIL（全局解释锁）、进程切换代价太大，所以高并发的时 gevent 会起到很明显的加速作用。

6）redis 是一种键值对数据库，数据存储于内存之中，可以实现大并发数据查询。并且支持 python、java、c++等各种常用的数据库，通常应用与会话保持、缓存等等场合。

3.2.3 平台管理模块详细实现

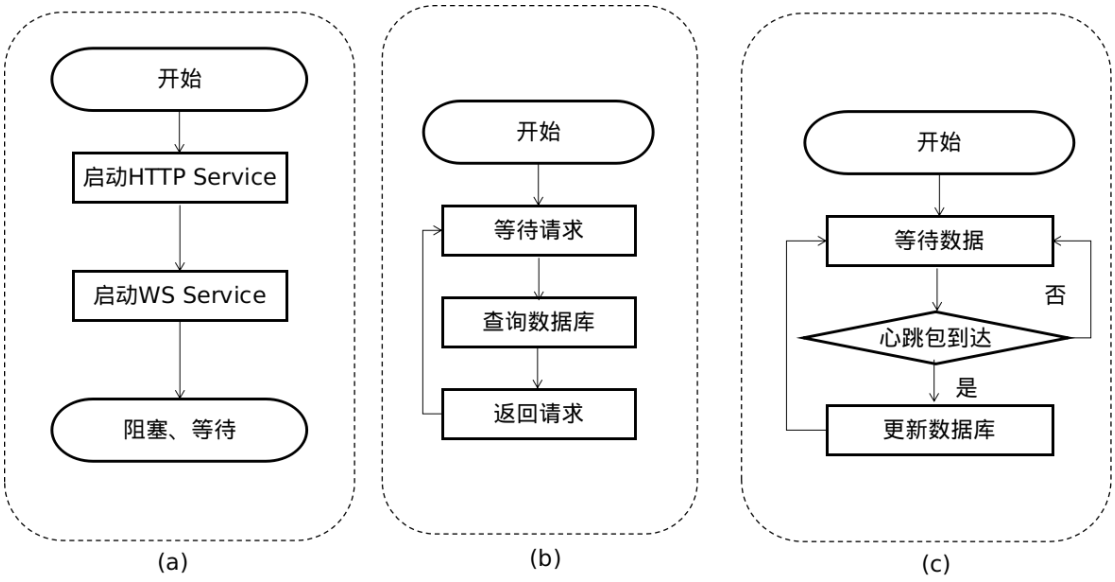


图 3-3 平台管理模块运行流程

运行流程如图 3-4，使用 gevent 编写多任务逻辑，主线程只负责启动服务，具体的工作逻辑在相应的子线程完成。

1) 数据库设计

数据库主要存放用户 ID、密码等等信息，于是在数据库中建立了数据表，如 3-1。

表 3-1 所有数据表

数据表名称	存放内容	主键

续表 3-1

users	用户信息，如用户名、密码等	id
access	权限表	id+权限
login_history	登录历史	id+time
sessions	会话信息	id

一共建立了四个表，分别是存放用户账户的 users 表、存放用户权限的 access 表、存放登录信息的 login_history 表以及存放会话信息的 sessions 表。

所有的 users 都有统一格式的编号，并且是不重复的。下面我将详细介绍每个表存储的内容：

users 表：存放用户的信息，包括 ID、用户名、密码等等。其中的密码以 hash 方式存储，是为了避免系统数据库泄露后，被黑客窃取用户的密码，如表 3-2。

表 3-2 user 数据表

列名	类型	含义
id	String(20)	用户编号
username	String(20)	用户名
hasn_password	String(40)	用户密码 hash
location	String(30)	注册地址

access 表: 存放用户权限等级的表，包含用户 ID 和权限等级设置这个表的原因是这样能够更方便地管理用户的访问权限，比如划分等级等等。

表 3-3 access 数据表

列名	类型	含义
id	String(20)	用户编号
access	Int	用户权限等级

其中，access 是权限等级，有一个 Int 型变量记录，数值越低等级越高，每个账户只能有高一级的账户建立。

login_history 表：历史记录表的存在是为了方便数据统计以及数据分析，如表 3-4。后续可以根据登录的 IP、location 等信息，经过聚类分析、挖掘得到热点工作区域，从而进行更合理的资源分配、资源优化。比如访问北京天安门的客户非常多，那就可以在分析后，在附近布置更多的设备。

表 3-4 login_history 表

列名	类型	含义
id	String(20)	用户编号
time	Int	登录时间
location	Int	登录地址
ip	String(12)	登录 IP

续表 3-4

dev_type	Int	设备类型

以上三个数据表都存放在 mysql 中，将信息进行持久化存储，而 sessions 是存储动态的登录信息，所以不适合在传统数据库中存储，因此以非结构化的方式存放在 redis 内存数据库中，以键值对方式存储，id 作为键

sessions: 存储会话信息，也就是当前连接的客户端的信息。为了查询以及统计的性能，将 id 作为了键，保证 o(1)的时间复杂度下能查找到某 ID 的信息。存储如下信息：

```
"id" : {
        "location" : "beijing",
      "dev_type" : "1/2",
      "ip" : "115.115.115.115",
 }
```

location 是实时的位置信息，dev_type 是设备类型。

对于计算节点，存储的就是如下信息：

```
"id" : {
      "location" : "xian",
      "ip" : "115.115.154.115",
      "temprature" : "34",
      "load" : "32",
      "memory_free" : "2048",
      "hdd_free" : "204800",
      "task_number" : "4"
 }
```

location 依然是实时位置信息；temprature 是节点的温度，单位是摄氏度；load 是实时负载，满载是 100；memory_free 和 hdd_free 是剩余的内存空间和磁盘空间，单位均为 MB；task_number 是当前正在运行的任务数。

2) Websocket Server 实现

Websocket Server 主要重写了三个函数，分别是客户端加入、客户端离开、消息到达。Websocket Server 只需要接受 Edge Worker 的连接。Websocket Server 的工作流程如图 3-4 所示。

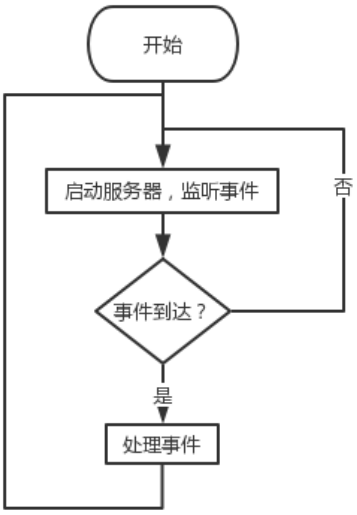


图 3-4 平台管理模块 Websocket Server 运行流程

服务器采用了事件-响应编程模式，使用了异步 I/O 模型，服务器首先注册监听器，然后等待事件发生，一旦触发，则开始处理过程，如图 3-4。

与同步编程模型相比，采用异步编程的好处显而易见。在以往使用同步方法编写的程序中，首先会执行 IO 函数，然后操作系统会让该线程休眠，然后调度其他线程，此时，当前线程就进入了阻塞状态，如果当前任务是单线程的话，整个任务就阻塞了起来，没法执行其他命令。一种解决方法是使用多线程，然后在线程结束后增加回调函数，线程结束后调用这个函数。这样的话，可以在启动 I/O 函数后继续做别的事，但是在无形中增加了编程的复杂程度。

python 从框架的层面解决了这一问题，从库函数的角度封装了回调，使得编程人员在编写程序的时候无需自己手动开启新的线程。

平台管理模块 Websocket Server 运行流程流程如图 3-5 所示。当浏览器连接之后，会触发 onNewClient()事件，系统会执行如下过程：不断将节点的信息发送给浏览器端，浏览器收到数据后不断更新界面，避免了轮询过程中不必要的请求包的浪费。当节点断开连接时，会触发 onClientLeft()事件，断开推送流。

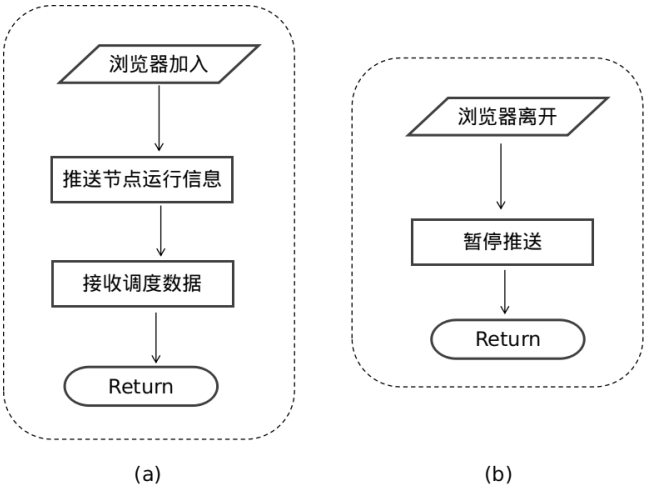


图 3-5 平台管理模块 Websocket Server 运行流程

3) HTTP Server 实现

Http Server 主要接受前端的请求，并返回所需信息。我们采用了 python-flask 框架编写了该段程序，同样是异步编程模型。

首先引入相应框架中的包：

```
from flask import Flask # flask 对象
from flask_cors import CORS # CORS 跨域
from flask_restful import Resource,abort,reqparse # 资源
import flask_restful as restful # restful
```

这里使用了 flask 框架，因为其与 django 相比起来更加轻量，与其他框架和库整合起来更加方便。注意到我们也引入了 CORS（跨域资源共享）对象，这里对 CORS 做一个解释：

浏览器为了安全，使得 html 页面中的 url 不能访问不同协议、端口、域名不同的地址，这是浏览器的同源限制，其中，同协议、端口、域名即为同源

但是，随着现代 web 架构的升级替换，前后端已经分离开发，很多情况下后端都是提供 api 共前端访问，此时浏览器就有可能访问非同源的 uri，于是理所应当受到同源策略的限制。

这种情况下，便可以通过 W3C 的 CORS(Cross-Origin Resource Sharing)方案来解决这个问题，这个方案简单来说，就是向响应头 header 中注入请求头，这样浏览器检测到 header 中的 Access-Control-Allow-Origin 时，即可进行跨域操作。

使用 CORS 需要浏览器和服务器的同时支持。当前几乎所有浏览器都支持了，我们只需要在服务器总添加相应的支持便可以了。

新建 app 对象之后，使用 CORS 动态修改 app 响应头文件，使其支持跨域。

```
app = Flask(__name__)
CORS(app) #跨域
api = restful.Api(app)
```

接下来编写相应的响应代码，继承 Flask 中的 Resourse 类，然后重写四个方法：post、get、put、delete 四个方法

```
class RESTapi(Resource):
    def post(self): # 响应 post 请求
        pass
    def get(self):  # 响应 get 请求
        pass
    def put(self):  # 响应 put 请求
        pass
    def delete(self): # 响应 delete 请求
        pass
```

然后将实体类与 url 建立联系，也即是通常所说的路由。这里的路由方式支持很多，这里不详细介绍。最后在 main 函数中启动 app，端口为 5000

```
api.add_resource(RESTapi,'/api')
if __name__ == '__main__':
    app.run(host="0.0.0.0",port=5000)
```

4) HTTP API

平台管理模块对外会提供 api 接口，当前有如表 3-5 的 api：

表 3-5 API 接口

url	作用	参数

续表 3-5

GET /nodes?limit=0	访问可用节点	limit，返回内容个数
POST /sessios	前端登录	username,password
POST /users	用户注册接口	username,password,location
POST /dockers	上传 docker 文件	Docker 文件
POST /project	上传代码工程	代码文件夹
POST /data	上传数据	数据
POST /tasks	启动任务	任务描述
GET /node/002	获取节点信息	返回节点信息
GET /systeminfo	获取系统运行概况	无

5) 调度算法实现

访问 POST /tasks 接口时，会执行节点的调度算法，如表 3-6。

表 3-6 调度算法列表

算法名称	基本策略
Random	随机
Spread	尽量均匀分布
Binpack	尽量占满一个结点

6) 平台管理模块 UI 设计

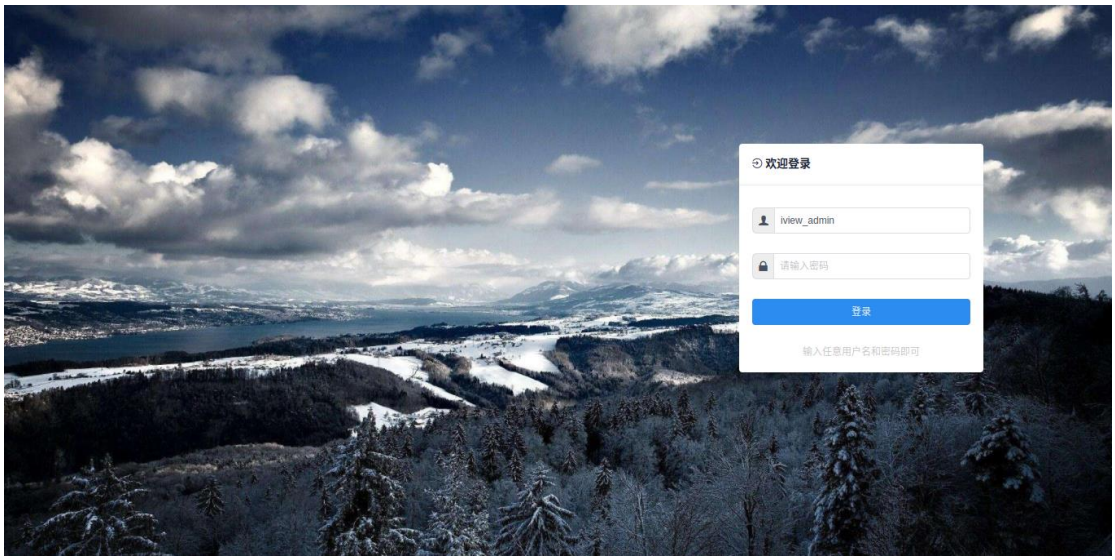


图 3-6 登录界面

登录界面要求用户输入用户名和密码，输入正确后进入系统概览界面。

图 3-7 系统运行概览

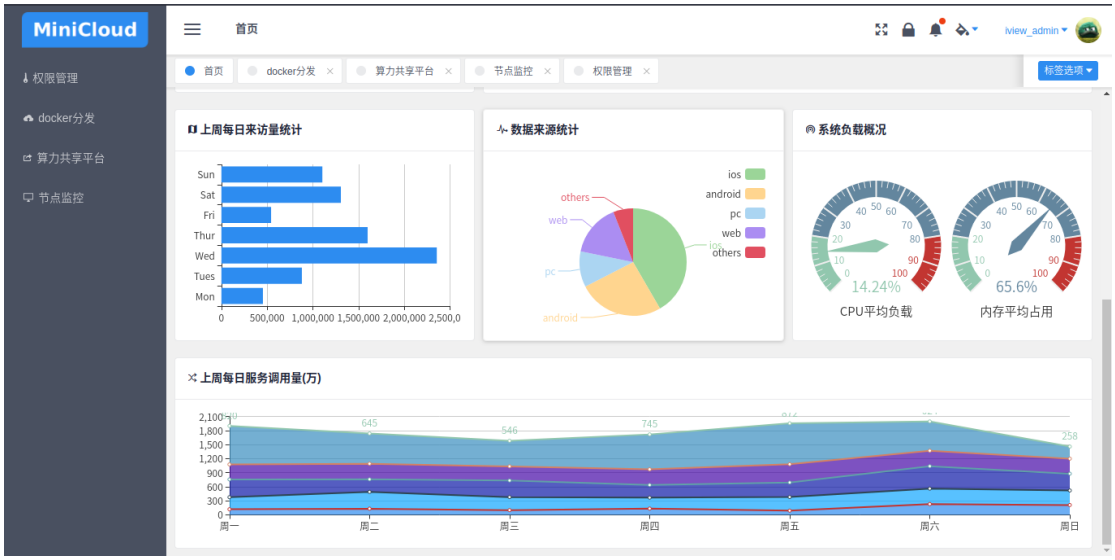


图 3-8 系统运行概览

系统概览界面展示了系统的大致运行状况，方便用户直观地看出系统的状态。如果系统运行异常，用户可以从 cpu 占用、内存占用、服务调用次数等等指标中分析得出。这些数据都是使用 websocket 连接，由服务器定时推送的，避免了刷新带来的延迟。

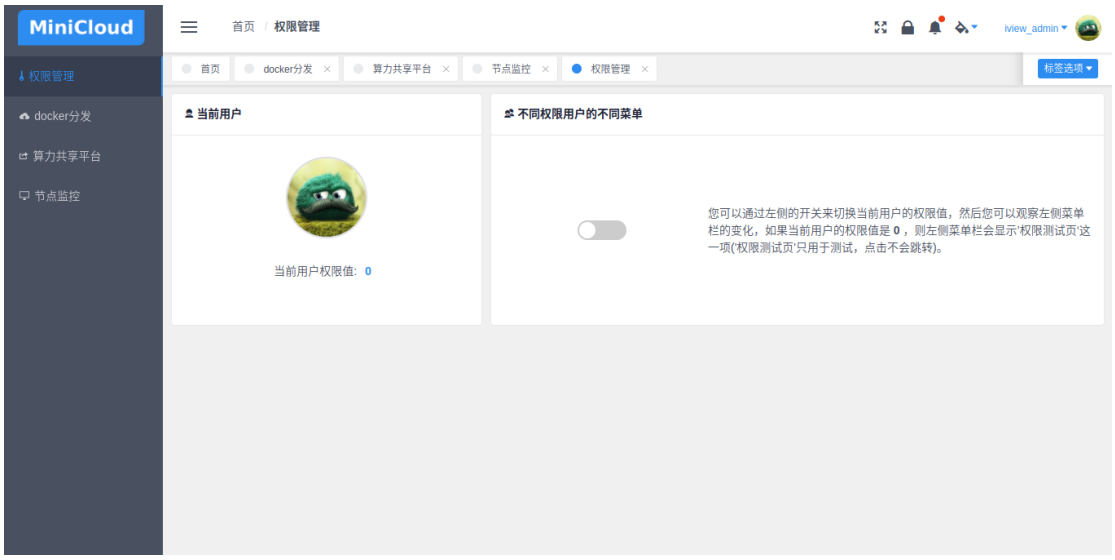


图 3-9 权限管理测试界面

权限测试界面体现了系统的权限管理功能，不同用户登录到系统可以看到不同的界面，控制了用户可操作的界面范围。

图 3-10 Docker 分发/上传界面

Docker 分发/上传界面分为左右两个部分，左边是镜像列表以及上传组件，用户可以选择其中的一个镜像进行分发，或者是删除这个镜像，如果列表中没有，则用户可以点击下面的上传组件，将自定义的 Docker 文件上传到服务器；右边是节点状态以及选择列表，实时呈现各个节点的负载情况、忙/闲状况，用户可以通过手动点击左边的复选框来选择要分发 Docker 的节点，另外最下边还有智能选择的功能，点击这个按钮后，系统会根据各节点的负载情况，评估合适的节点，然后将推荐的节点进行勾选。点击中间的分发按钮，镜像将会分发到选中的节点，并运行起来.

图 3-11 算力共享平台

共享算力界面与 Docker 分发界面有很大的相似之处，界面右边也是节点选择平台。左边分为上下两个部分：上边为代码选择组件，用户可以选择电脑上的代码文件夹进行上传；下边为数据选择组件，用户可以选择数据文件进行上传。点击分发按钮后，每个节点将会获得同一份代码文件，获得不同的数据文件。运行完成之后结果会返回平台。

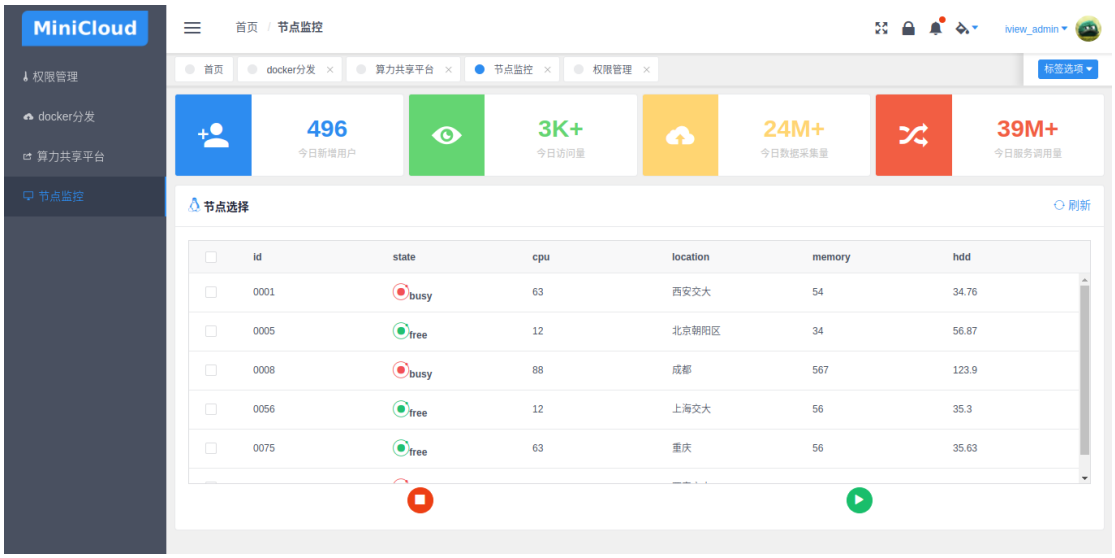


图 3-12 节点监控界面

节点监控界面显示了所有节点的详细信息，包括 cpu、内存、硬盘、地址等多个信息，用户可以实时观察到各个节点的运行状况。该界面还提供了启动/停止功能。在表格复选框中选中节点后，点击界面下方的启动按钮，便可以启动相应节点中的功能，点击停止按钮，则被选中节点运行的进程停止运行。

3.3 容器模块

容器模块是系统的基础，它管理维护着一个 Master 节点和多个 Worker 节点。容器模块由部署在每个节点上的 Docker 引擎、节点通信模块和资源占用检测模块构成。

Docker 引擎是 Docker 容器技术的基础，它实现了 Docker 虚拟化，支持解析 Dockerfile

来创建 Docker 镜像,并且可以根据节点列表分发 Docker 镜像。节点通信模块实现了 Master 节点与 Worker 节点间的可靠通信,使得 Master 节点可以分发镜像到 Worker 节点上，Worker 节点的资源使用情况也可以上传至 Master 节点，以用于分析。资源占用检测模块会根据 Master 节点下发的配置，实时监测系统各方面资源的占用情况，例如：CPU 使用率，内存使用率等。

我们使用社区版的 Docker 软件作为容器引擎，将其部署到每一个节点上，并创建一个 Overlay 网络，使得所有节点处于同一网络下。

由于 Master 节点和 Worker 节点存在频繁的数据交换，但是数据包相对较少，因此我们使用 TCP 长连接的方式来进行 Master 节点和 Worker 节点的通信，数据包采用 JSON 格式，字段含义如表 3-7 所示。其中 TYPE 表示的是 JSON 包的类型,TYPE 为 LOAD 的表示 Worker 节点向 Master 节点上传的数据包，其中包含：CPULOAD,MEMORYLOAD,TIM-E,STATE 字段。CPULOAD、MEMORYLOAD 和 HDDLOAD 字段分别代表了 CPU 使用率、内存使用率和存储剩余空间。TIME 是采集使用率时的 UNIX 时间。STATUS 是节点状态，0 表示没有任务运行，1 表示正在运行，2 表示节点发生错误。TYPE 为 COM 的是 Master 节点向 Worker 节点下发的命令，它只有 COMMAND 一个字段，其值即为命令。

表 3-7 JSON 字段含义

字段	含义
TYPE	类型(可选：LOAD，COM)
CPULOAD	CPU 使用率
MEMORYLOAD	内存使用率
HDDLOAD	存储剩余空间
TIME	UNIX 时间
STATUS	节点状态
COMMAND*	Master 下发的命令

资源检测模块负责检测节点的资源占用情况，我们使用 psutil 库来获取特定进程的 CPU 使用率和内存使用率。Docker 每创建一个容器，就会创建一个 docker-containe 进程，容器运行在这个进程中，因此我们可以监视每一个 Docker 容器的资源占用情况，及时处理各种错误。

3.4 调度器模块

共享算力平台下发的计算任务要在不影响节点原有用户任务的情况下运行,保证使用的是剩余算力，不会对节点用户产生干扰。因此，Docker Swarm 自带的编排策略不能满足平台的需求，必须开发自己的调度器模块。

虽然 Docker 的启停速度很快只有秒量级，但是节点原本运行的应用可能突发高负载的时间也是秒量级的，因此只依赖 Dokcer 的快速停止来避免影响原应用的卡顿是很可能失败的。因此，我们使用机器学习模型来预测节点未来的资源占用率，节点空闲资源大于下发计算任务所需资源的时间称为节点空闲时间。只有节点空闲时间大于计算任务运行时间的节点才会被下发计算任务。这样，就可以避免下发的计算任务影响节点原有的用户任务的运行。

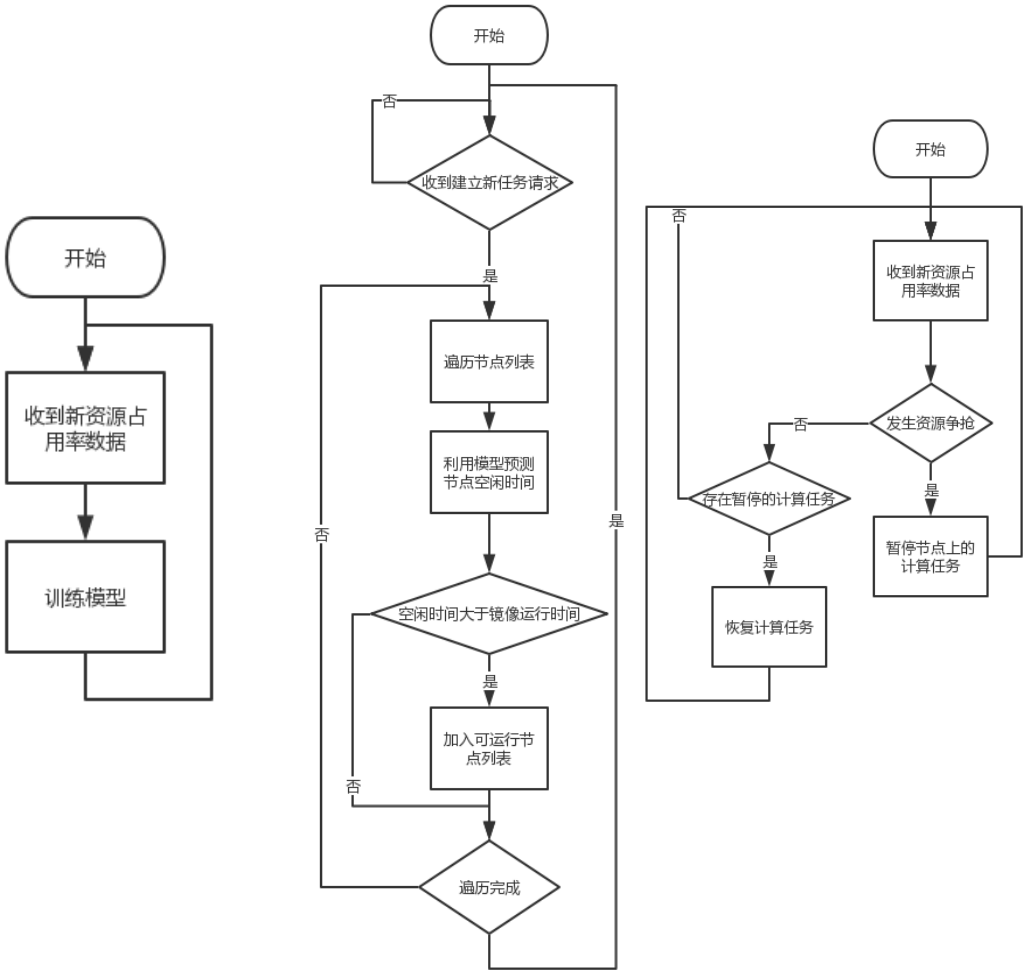
我们设计了几种典型的应用的资源占用率曲线，比如网络摄像头，智能网关等。然后将数据导入到 Matlab 中进行训练。我们使用了 24 小时的数据，每 2 秒采集一次资源占用率。结果如表 3-1 所示，其中 RMSE 和训练时间都是在 10 叠交叉验证下得到的结果。

表 3-8 机器学习模型训练结果

模型	RMSE	训练时间
Linear Regression	22.55	12.540s
Robust Regression	24.172	6.415s
Fine Tree	3.3492	215.940s
Medium Tree	3.11	39.204s
Coarse Tree	2.9901	43.940s
Boosted Tree	3.1577	19.960s
Bagged Tree	3.0597	74.799s

从表 3-1 中可以看出 Boosted Tree 的 RMSE 比较小，而训练时间非常短，在精度和时间上有比较好的平衡，因此我们使用 Boost Tree 模型。

调度器模块由 3 部分构成，分别是训练模块，预测模块和突发处理模块。训练模块主要用来接受 Worker 节点上传的数据然后对节点的模型进行训练。预测模块主要用来在平台管理模块下发计算任务时使用模型预测 Worker 节点的空闲时间，然后返回空闲时间大于计算任务所需时间的 Worker 节点。突发处理模块是用来在 Worker 节点上发生原有的用户任务和计算任务争抢资源时，也就是说在运行计算任务时用户任务突发高负载时，将计算任务暂停，尽量避免对用户任务的影响。模块运行流程图如图 3-14 所示。



(a)训练模块　　(b) 预测模块　　(c)突发处理模块

图 3-14 调度器运行流程图

调度器模块使用 Python 编写，Gradient Boost Regression Tree 模型使用 Scikit-learn 机器学习库进行训练和预测，容器模块通过调用调度器模块的函数来使用其功能。

第四章 系统测试

4.1 Docker 分发测试

进入 docker 分发测试页面，可以看到所有的 docker 镜像已经呈现在列表中，每个镜像都呈现出相关的信息，包括镜像大小、镜像备注；右边以表格形式呈现各个节点，有每个节点的 CPU 信息、内存、HDD 等。

选择左边已经存在的镜像 docker0002，然后勾选右边的前三个节点，其中两个节点处于空闲状态，另一个处于忙碌状态。

点击分发按钮，触发分发操作，系统会执行分发的算法，将镜像分发到合适的节点。本测试中，有两个节点符合要求，另一个节点不符合，所以界面的右上角弹出分发成功的消息，有两个成功，一个失败。

图 4-1 Docker 分发成功界面

4.2 算力共享平台测试

打开算力共享平台，可以看到页面分为左右两个部分，左边是程序与数据包上传控件，右边是节点选择界面。

首先我们点击右下角的智能选择按钮，可以看见节点列表中的 2、4 个节点被选中。

图 4-2 智能选择节点成功界面

分别点击选择代码和选择数据，进行资料上传。然后点击分发按钮，界面右上角弹出分发情况，本例中成功 2，失败 0，因为前面我们已经通过智能选择自动选择了节点，所以没有失败案例。

图 4-3 程序与数据分发成功界面

4.3 节点监控功能测试

打开节点监控功能页面，可以看到页面分为上下两个部分，上部是系统运行概况，包括了新增用户、访问量等等。

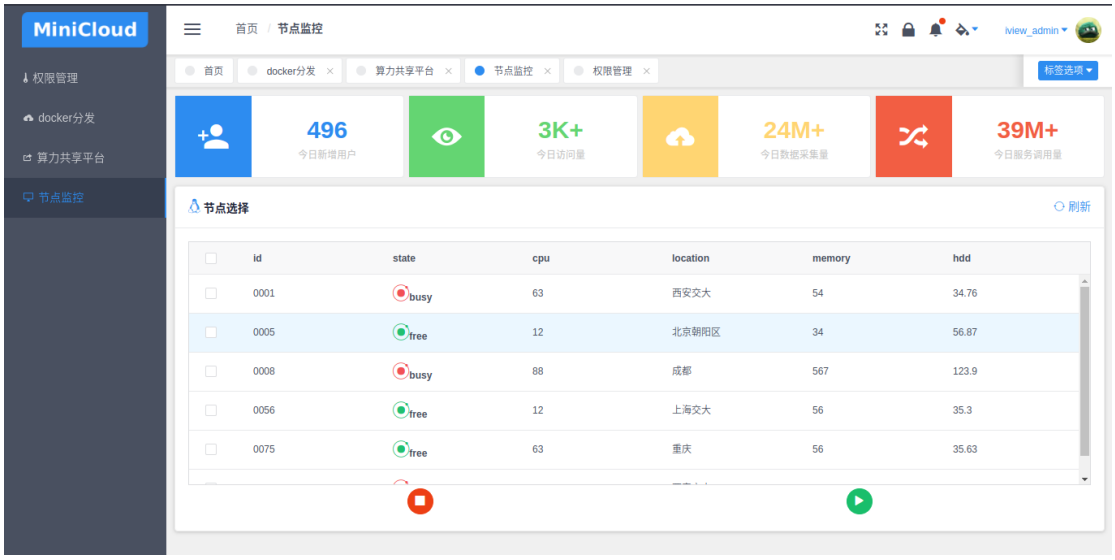


图 4-4 节点监控界面

选中 0005 和 0008 号节点，点击界面下方停止按钮，系统将通过调度模块停止选中的节点。因为本例中，0005 号节点为 free 状态，所以会造成停止失败。

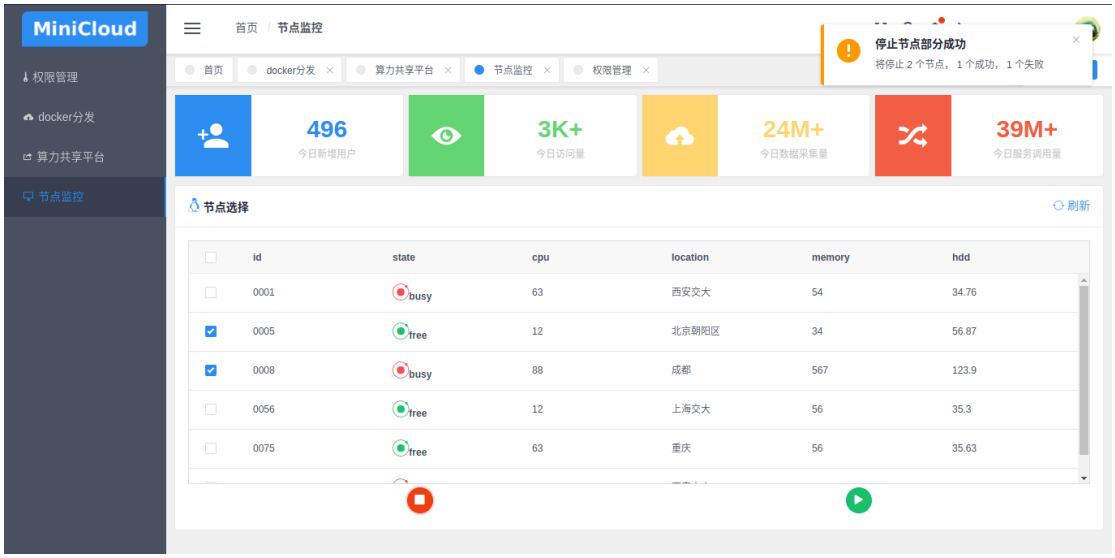


图 4-5 节点停止功能测试界面

操作完成之后，0008 号节点负载释放，变为 free 状态。

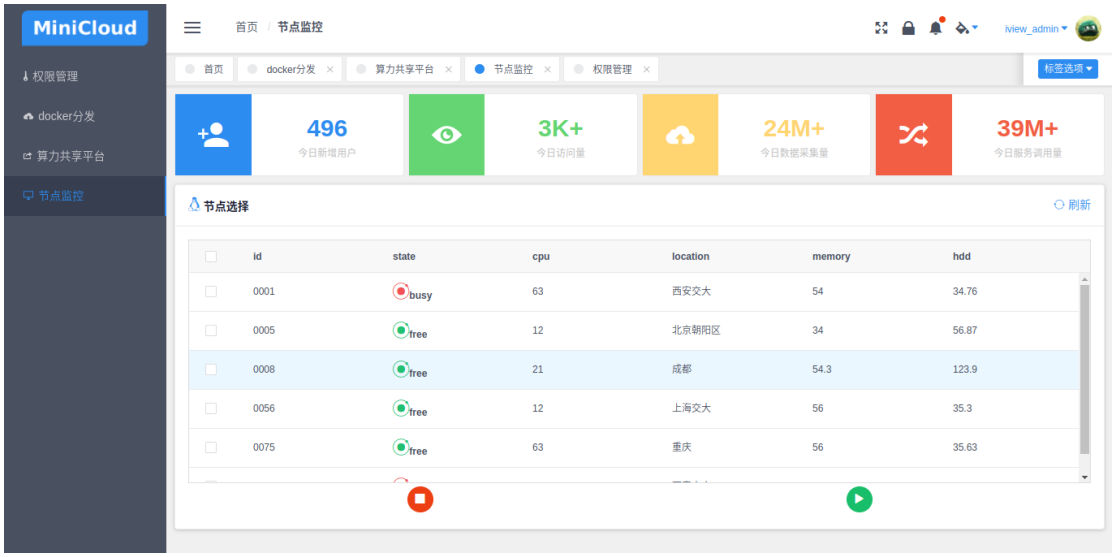


图 4-6 节点释放成功后界面

第五章 结论

5.1 系统特色

随着嵌入式平台性能的提高，用嵌入式平台构建云计算集群成为可能。传统高性能服务器耗电多，发热量大，需要专用的机房来散热。而嵌入式平台功耗低，单价低，体积小，热功率小，非常适合大规模的部署。而且由于嵌入式平台单价低，因此相同的资金可以部署更多的节点，故障节点对系统的影响更小，系统有更强的鲁棒性。

但是传统的虚拟机技术虚拟损耗非常大，而且镜像体积大，不适合嵌入式平台性能相对于服务器较低，存储空间小的特点。同时，虚拟机启停很慢，通常是分钟级，因此在节点故障时，系统容量恢复需要几分钟时间，也不适用于嵌入式集群节点数量非常多的特点。而容器技术恰恰适合上述所有特点。相比于传统虚拟机技术，容器技术性能损耗低，通常在百分之几，镜像体积小，只有几十 MB，启停速度非常快，只有秒量级。因此非常适合于作为嵌入式平台上的虚拟化技术。

因此，我们基于容器技术设计实现了一个 mini 云计算平台。平台由多块 Intel UP2 嵌入式平台构成，采用 Docker 开源容器引擎作为虚拟化技术，管理系统采用 Web 界面。系统用户可以直接通过浏览器访问平台管理系统。用户可以自定义 Docker 镜像，或者从 Docker 镜像仓库下载。用户可以自定义使用节点数量、节点位置以及容器数量。用户可以监控并管理自己的任务，系统提供自动的故障恢复功能。

同时，我们注意到现存的大量嵌入式应用，比如网络摄像头，家庭网关等，它们都不是一直处于高负载状态。它们有大量的时间是不被用户使用，或者进行低负载任务的。因此它们有很多计算资源被闲置。因此，我们基于 mini 云平台实现了一个共享算力平台。因为这个平台要利用的是剩余算力，也就是说不能对节点原本的用户产生影响，因此我们设计实现了一种基于 Gradient Boosted Tree 的调度器，它可以根据节点的历史资源占用率来预测节点未来一段时间内的资源占用率，从而把计算任务部署到可以不影响节点原有任务的空闲节点上，在实现了对闲置资源的高效利用的同时也使得计算任务对节点原有用户是透明的。

综上，本系统特色有：1.构建了嵌入式平台的 mini 云计算平台，具有功耗低，体积小，热功率小，构建容易，鲁棒性强的优点。2.基于 mini 云平台构建了一个共享算力平台，实现了一个可以预测节点资源占用率的调度器，在实现了对闲置资源的高效利用的同时，实现了计算任务对节点原用户的透明。

5.2 系统展望

由于 Docker 是利用 cgroups、namespaces 等技术实现容器间资源隔离的，并没有对硬件和内存空间进行虚拟化，因此 Docker 容器是不能在运行中在节点之间进行迁移，当然这也是容器轻量级的设计理念所限。但是，在传统服务器集群系统中，保活迁移是一项非常有用的功能，因此未来可以从这方面入手，在不非常影响 Docker 性能的前提下，实现保活迁移功能。

参考文献

[1] 王鹃, 胡威, 张雨菡, et al. 基于 Docker 的可信容器 [J]. 武汉大学学报(理学版), 2017, 63(2): 102-108.
[2] Yu HE, Huang W. Building a Virtual HPC Cluster with Auto Scaling by the Docker [J]. Computer Science, 2015,
[3] Stubbs J, Moreira W, Dooley R. Distributed Systems of Microservices Using Docker and Serfnode; proceedings of the International Workshop on Science Gateways, F, 2015 [C].
[4] Merkel D. Docker: lightweight Linux containers for consistent development and deployment [J]. 2014, 2014(239):
[5] 杨保华, 戴王剑, 曹亚仑. Docker 技术入门与实战 [M]. : 机械工业出版社, 2015.
[6] Boettiger C. An introduction to Docker for reproducible research [J]. Acm Sigops Operating Systems Review, 2015, 49(1): 71-79.
[7] 黄凯, 孟庆永, 谢雨来, et al. 基于 Docker swarm 集群的动态加权调度策略 [J]. 计算机应用, 2017,
[8] Bernstein D. Containers and Cloud: From LXC to Docker to Kubernetes [J]. IEEE Cloud Computing, 2015, 1(3): 81-84.
[9] Wang K, Yang Y, Li Y, et al. FID: A Faster Image Distribution System for Docker Platform; proceedings of the IEEE International Workshops on Foundations and Applications of Self* Systems, F, 2017 [C].