分类号：TP311.5　　单位代码：10335

密　级：无　　学　　号：21351033

浙江大学

硕士学位论文

中文论文题目：基于 Docker 的企业 PaaS 系统设计与实现

英文论文题目：the enterprise PaaS system design and implementation based on Docker

申请人姓名：陈东东

指导教师：施青松 副教授

合作导师：

专业学位类别：工程硕士

专业学位领域：软件工程

所在学院：软件学院

论文提交日期　　年　　月　　日

基于 Docker 的企业 PaaS 系统设计与实现

论文作者签名:_____

指导教师签名:_____

论文评阅人 1：_____

评阅人 2：_____

评阅人 3：_____

评阅人 4：_____

评阅人 5：_____

答辩委员会主席：_____

委员 1：_____

委员 2：_____

委员 3：_____

委员 4：_____

委员 5：_____

答辩日期：_____

the enterprise PaaS dystem design and implementation based on Docker

Author's signature:_____

Supervisor's signature:_____

Thesis reviewer 1：_____

Thesis reviewer 2：_____

Thesis reviewer 3：_____

Thesis reviewer 4：_____

Thesis reviewer 5：_____

Chair：_____

Committeeman 1：_____

Committeeman 2：_____

Committeeman 3：_____

Committeeman 4：_____

Committeeman 5：_____

Date of oral defence：_____

浙江大学研究生学位论文独创性声明

本人声明所呈交的学位论文是本人在导师指导下进行的研究工作及取得的研究成果。除了文中特别加以标注和致谢的地方外，论文中不包含其他人已经发表或撰写过的研究成果，也不包含为获得 **浙江大学** 或其他教育机构的学位或证书而使用过的材料。与我一同工作的同志对本研究所做的任何贡献均已在论文中作了明确的说明并表示谢意。

学位论文作者签名：　　签字日期：　　年　　月　　日

学位论文版权使用授权书

本学位论文作者完全了解 **浙江大学** 有权保留并向国家有关部门或机构送交本论文的复印件和磁盘，允许论文被查阅和借阅。本人授权 **浙江大学** 可以将学位论文的全部或部分内容编入有关数据库进行检索和传播，可以采用影印、缩印或扫描等复制手段保存、汇编学位论文。

（保密的学位论文在解密后适用本授权书）

学位论文作者签名：　　导师签名：

签字日期：　　年　　月　　日　　签字日期：　　年　　月　　日

摘要

近几年，云计算已经成为一个流行词语，而 PaaS 系统就是云计算的一类重要类型。目前市场上已经拥有大量的公有 PaaS 平台，例如 Google 公司的 GAE、微软公司的 Windows Azure 以及开源的 Cloud Foundry。PaaS 集合了现有的各种服务和工具，并对它上面的各类应用提供所需的技术支持。对于企业而言，将重复的工作和技术服务变成一种环境服务将大大提高企业的开发效率，减少在服务配置、环境配置等与业务逻辑无关的事务上的时间。因此，PaaS 系统对于企业也有非常深远的应用价值和意义。

本文的主要研究内容是利用 Docker 开发出一套小型 PaaS 系统，针对中小企业服务器数量少，简单易部署的需求。本文首先举例了比较流行的 PaaS 平台，分析了他们的各自特点和缺陷。然后为解决中小企业的需求设计并实现了一套简单易部署，灵活易用的 PaaS 系统。

PaaS 系统使用了开源的应用容器引擎 Docker，开发者可以打包他们的应用和依赖包到一个可移植的容器中，然后发布到任何流行的 Linux 机器上。本系统也使用了 rethinkDB 作为系统数据库存储，它可以方便高效的将 json 对象存储于文件。使用了 etcd 作为 Docker 主机的注册中心，并利用同步机制来发现 Docker 主机的注册和宕机。

本 PaaS 系统将使用 Go 作为唯一编程语言，整个系统可以复制到另外一台计算机运行，实现了简单易部署的需求。使用 Docker 作为应用装载工具，从而满足了应用部署灵活的需求。

关键词: 云平台，PaaS，Docker，服务自发现，容器

Abstract

With the development of internet technology, cloud computing is becoming more and more popular. PaaS(Platform as a Service)is one kind of cloud computing. Now, there are many public PaaS platforms exist on the market. Such as Google's GAE, Microsoft's Windows Azure as well as the open source PaaS: Cloud Foundry. PaaS is one collection which consist of development service, tools and much necessary technical support for applications. For company, let's the repeatly work and config operation become one platform will improve their development efficiency and reduce the time spent on service configuration, environment configuration and other work which has nothing to do with business logic. Thereforce, PaaS also have much important significance for SMEs.

The main contents of this paper is to develop a small PaaS system for SMEs base on Docker technology. This PaaS will satisfy the requirement of easy to deploy and easy to use.Firstly, it analyzed serveral popular PaaS platform and provide their feature and disadvantage. Therefore, the PaaS platform is designed and implemented which is easy to deploy and easy to use.

The PaaS platform base on the open source application container engine: Docker. Developer can package their applications and dependencies in a container image, and then publish it to any popular linux machines. The rethinkDB is used as the database to store the json data. Etcd is used as a register server. The PaaS platform also use synchronizatioin mechanisms to discover registration and downtime of Docker server.

This PaaS system use Golanguage as the only programming language, so that the PaaS could run without any other configuration or library. This feature satisfy the need of easy to deploy. The system use Docker as application deployment tools to satisfy the need of easy to install application.

Key Words:cloud platform, PaaS, Docker, Automatic service discovery, container

目录

图目录

表目录

第1章 绪论

1.1 课题背景及研究的目的和意义

1.1.1 课题背景

云计算在当今社会已经变成一个热点，它的主要概念是将计算机集中，组成一个拥有海量计算机的资源池。在其之上所有的系统和应用都是通过虚拟化技术来搭载的，通过虚拟硬件，将系统运行于虚拟硬件之上，然后再基于系统运行更加上层的系统。这使得一个系统可以跨越数据中心，跨越网络，跨越地域限制，在世界的任何地点都能得到所需的虚拟资源。目前主流的云计算系统可按照基础设施、平台、软件分为三类，每一类可作为独立的服务:

基础设施类的云计算系统（IaaS）的云计算系统[1]主要是向客户，包括企业或者个人，提供虚拟化得到的硬件资源，这些资源可以是计算资源，存储资源，网络设备资源比如交换机和路由器等。该类系统使用网络向用户提供特定的硬件资源服务。

平台类的云计算系统（PaaS）[2][3]是向用户提供包含应用以及服务开发，运行，升级，维护或者存储数据等服务的云计算系统。平台类的云计算系统核心是提供中间件服务的云计算系统，用户使用该类型云计算系统可以调用中间件提供的各类服务，实现自己应用的开发，配置和运行。应用所需的中间件软件，虚拟化服务器与网络资源，应用的负载平衡等维护方面由平台类云计算系统提供服务予以解决。

应用类云系统（SaaS）[4]是直接为各用户提供所需的软件服务，同样这些服务是通过 web 应用方式提供的，用户可以通过浏览器使用网络来远程登入到这些软件服务的界面，使用服务提供的各类软件功能。与现在的 B/S（浏览器/服务器）系统类似，但本质不同，应用类云计算系统向用户收费是租凭式的，按使用的计算资源，时间等标准收费，传统 B/S 系统是卖整套系统给用户。

PaaS 平台可以从应用的角度来管理 IaaS，而传统的工具是通过资源调度的角度来管理 IaaS，这样的做法可以直接根据上层应用的需要来进行自适应的资源调度服务。这些调度服务可以通过 IaaS 对硬件资源虚拟化后所提供的 API 来操作，从而实现实时监控平台上的各类资源，资源以 API 的形式开放给 SaaS 用户。随着大数据的兴起，云计算也越来越成为热点，目前已经有很多成功的 PaaS

平台出现在国内外，如微软的云计算平台 Windows Azure，Google 的云计算平台 Google App Engine，VMware 的一项开源 PaaS Cloud Foundry。本文需要设计和实现的 PaaS 系统也将拥有 PaaS 的核心功能，主要包括应用部署，应用监控等，但同时又突破了两大限制，从而实现了不限制应用开发技术和 PaaS 系统快速部署。

1.1.2 课题的来源及研究意义

本课题是对 cloud foundry 和 Docker 研究后提出的，希望针对中小企业编写一个能自由使用各种资源的，简单易安装的微 PaaS 系统平台。该平台的主要功能包括管理员账户管理、Docker 服务器集群管理、应用部署管理以及服务自发现。本文负责该微 PaaS 系统的全套设计与实现，功能实现基于 Docker 容器技术，实现了包含账户管理，负载均衡，Docker 服务器集群管理，批量应用部署及管理，Docker 主机自动发现添加及其宕机检测等多项功能。

该 PaaS 系统与传统的 PaaS 系统具有较大差别，这些差别将给该 PaaS 系统带来意义深远的特性。

1.使用 Docker 容器来装载应用，装载的应用对象为应用容器镜像，该镜像已经包含了运行环境部署，依赖包，网络配置，存储配置，应用代码部署等操作。开发者可以在开发测试完成后将自己的整个环境和应用一起打包成 Docker 容器镜像发送到 PaaS 中进行原样部署。这就保证了该容器内的应用环境及配置与开发测试环境一致，使该 PaaS 系统有能力将几乎所有应用移植到它上面运行。

2.使用 Go 语言做开发语言。首先，将 Go 语言代码生成的程序的性能接近 C/C++程序。其次，Go 编译得到的程序是一整个完整的程序，可以直接将程序拷贝到服务器运行起来，而无需下载安装各种依赖。该 PaaS 系统可以一键安装运行，无需繁琐的安装部署。

3.目标群体是中小企业。该 PaaS 系统没有大型 PaaS 云的复杂服务，所有服务可以由企业自己制作 Docker 镜像安装。PaaS 系统占用计算机资源小，满足中小企业只有几台服务器却想系统管理自有服务器的意愿。代码结构清晰，很适合进行二次开发或者根据需求修改代码功能。

1.2 相关的国内外研究现状

1.2.1 对 PaaS 云平台的国内外现状

PaaS 系统已经在国内外普遍流行，目前主流的 PaaS 产品有 Microsoft

Windows Azure，Google App Engine，VMware Cloud Foundry，Force.com，Heroku，Amazon Elastic Beanstalk，CumuLogic，Sina App Engine，Baidu App Enginee，Aliyun Cloud Enginee 等[5][6][7]。

GAE，Azure，Elastic Beanstalk 是目前 PaaS 平台的三大供应商，占据着 PaaS 服务市场的大部分份额。GAE 给客户提供了垂直一体化的开发环境，使客户可以在这环境中创建 web 应用[8]。微软在 2008 年也发布了共有云 Windows Azure[9]，开发者在其 PaaS 之上可以使用已有的消息总线和队列系统等，而且还提供数据存储服务和网络服务。但是这些做法会将开发者绑定到特定的服务上[10]。Elastic Beanstalk[11]为在 Amazon Web Services 云中部署和管理应用提供了一种方法。该平台直接在它自身上面构建 tomcat 软件栈，从而使 java 程序能直接在其之上运行。但它还是对语言存在较多限制，无法提供对全语种的支持，比如非常流行的 js。

Cloud foundry 是目前开源 PaaS 系统中最热门的公有云平台，主要由商业公司做支持[12]，是 VMware 公司推出的一款开源 PaaS 平台，它是一个 Ruby 语言开发的分布式系统，为开发者提供了应用和服务平台，并实现了与 IaaS 平台交互的工作。Cloud Foundry 开发时考虑到了多方面的因素，它支持多种框架、语言、云平台及应用服务[13]。Cloud Foundry 开发人员在开发过程中已经考虑到了该系统的可自愈性，在各个层级都可以水平扩展，不仅可以大规模运行在大型的数据中心中，还可以运行在个人电脑上，二者的代码库一致[14]。Cloud Foundry 的开源特性使其方便进行二次开发，而且它的部署也相对比较容易，给开发者提供了一个很好的平台。但是代码结构相当庞大，定制需要理解整个框架及内部实现，对与中小企的门槛稍高。

Force.com 是 Salesforce 的开发者平台，允许开发者扩展 salesforece 的功能从而创建应用[15]。Force.com 提供了 web 服务的应用编程接口以及工具集，一个用于创建应用的用户接口，一个用于存储数据的数据库以及基础的网站托管功能。近日，平台还添加了移动功能，使得基于 force.com 的 PaaS 平台开发移动应用也一样便捷[16]。 使用 force.com 的缺点是：应用开发人员不能随意使用编程语言创建通用的应用程序。应用逻辑层建立于 apex 之上，它是运行在 force.com 上的一种强类型，面向对象又类似于 java 的语言。

Heroku 是支持可移植平台及服务的公司之一[17]，Heroku 公司起步的时候，只支持 Ruby 的平台独立性，即 Ruby 编码能以任何形式部署[18]。后来被

salesforce.com 公司所收购，并且扩大了语言支持。目前，除了 Ruby 外，该平台还支持 Node.js、Clojure、Java、Python 和 Scala 等语言，但 heroku 仍然不支持应用开发者访问主机文件系统上的文件。这样就可以更为简单的为用户应用创建多个实例。Heroku 平台可以做到平台上的程序代码与本机上的代码相一致，使开发更轻松。

CumuLogic 是一个专门针对 java 开发者的 PaaS 平台[19]，它可以将一个在本地主机运行的程序直接迁移到云端运行，然后通过 CumuLogic 所给的工具来实现对应用的管理。因此开发者就可以轻松的将自己开发的应用经过本地测试后发送到云端运行，但缺点是只能使用 java 作为开发语言。

BAE 是百度开发出的 PaaS 平台[20]。在该平台之上，开发者无需兼做运维，可以将全部精力投入业务逻辑的开发，最后将程序打包上传。剩下的部署和维护任务就交给 BAE，BAE 会发布开发者的应用并向普通用户提供服务，而且它可以根据负载，自动将应用配置为分布式架构，解决了开发者的性能顾虑。但所有的这些的前提是开发者必须基于 BAE 所提供的服务和语言进行开发，不能自己添加服务。

ACE 是阿里云开发的一款 PaaS 平台[21]，它提供了一个基于分布式的应用程序运行环境，可以将 java，php 等语音开发的应用运行其平台之上。该平台提供了多种基于分布式的服务，可以给开发者的应用提供强大的支持。

SAE 是新浪开发的 PaaS 平台[22][23]，它在国内推出较早，现在使用量也较大，在国内属于较成熟的 PaaS。它的特点是开发者的代码将代码上传到 SAE 后，会自动同步到 SAE 后端的 WEB 服务器，然后使用类似沙箱的容器对应用的代码和数据进行隔离，更进一步会隔离内存和 CPU 资源。不过目前而言，只有支持三种语言的应用：php，java 和 python。

在以上 PaaS 平台中，Cloud Foundry 属于开源 PaaS，剩余的 PaaS 都是以公有云的形式向开发者提供平台服务，这样也大大限制了开发者要被限制于 PaaS 提供者的服务。比如目前都存在的一个缺点就是所有应用智能监听一个 HTTP 端口，应用之间的交互非常困难。比如基本上目前已有的应用都无法直接或者非常少修改的迁移 PaaS 平台上，这因为 PaaS 平台提供的服务版本或者数据库版本与开发的不一样等原因导致的。虽然 PaaS 可以为开发者的应用自动扩展和负载均衡，但负载均衡方式其实选择几乎没有，只要使用共有 PaaS 就必须接受限制，目前已经存在的应用若想迁移到 PaaS 系统之上，那它必须进行一些修改

并在 PaaS 系统上进行测试部署。因此很多企业希望有自己的私有 PaaS 云，企业私有 PaaS 云需要能提供简易的安装，云平台资源随意使用，任意添加定制资源服务。Cloud Foundry 作为开源 PaaS 私有云已经流行与各个大公司，Cloud Foundry 的搭建和维护都需要大量的人员投入大量的时间和经历，光部署就需要定制一套部署工具 BOSH，利用它将整个系统部署于集群中，但是由于 bosh 存在很多欠完善的地方，因此很难利用它在系统集群或者 IaaS 上成功部署，大大增加了使用难度。中小企业无法提供如此的人力物力，他们需要的是能快速搭建系统和应用的随意开发和添加服务。显然以上的 PaaS 都无法满足。

1.2.2 容器技术的国内外研究现状

现有的虚拟化技术主要分为两类：第一类是基于容器的，也叫系统级虚拟化；另外一类是基于硬件虚拟化的，也叫应用级虚拟化。硬件虚拟化需要一个虚拟层在操作系统之上，它可以虚拟化出一套一个操作系统所需的所有硬件，然后在该虚拟化硬件中再安装一个操作系统，它与主机系统完全隔离。使用容器的虚拟化技术是将同一个操作系统内核之上运行多个容器，这些容器之间互相隔离不可访问，容器与主机系统使用同一个操作系统内核。

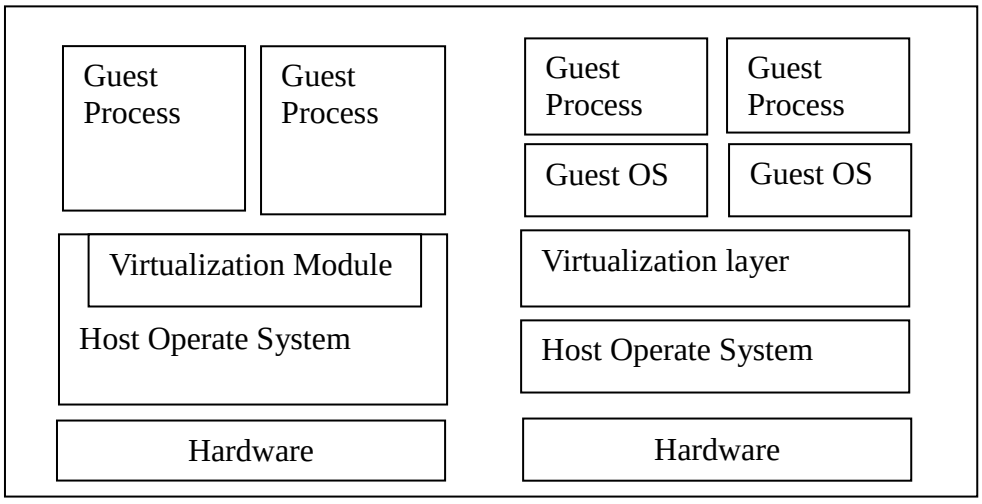

图 1.1 容器技术和虚拟层技术比较

当前，主流的操作系统级虚拟化技术（容器技术）有 OpenVZ，Docker，Lmctfy，Solaris Zones 和 Linux-VServer 等，下面对它们进行简单介绍和分析：

OpenVZ 使用 UBS（User Beancounters），通过公平的 CPU 调度，磁盘分配和 I/O 调度来达到资源管理的目的[24]。UBS 实现了通过控制参数来保证和限制

每个容器的权限和资源使用约束。这种方式可以让我们约束内存使用和各种内核对象，例如 IPC 共享内存段和网络缓冲区。为了促进容器之间的调度公平性，OpenVZ 的 CPU 调度被实现为两个层。第一次会根据调度策略选择合适的容器将 CPU 的资源（时间片）分配给它。第二层会使用检测容器内部的所有进程，并根据进程优先级来调度进程的运行。磁盘分配则可以为每个容器将每个用户或者用户组磁盘设置约束。

在资源隔离上，OpenVZ 使用内核域名空间技术来为资源作隔离。PID 域名空间允许每个容器拥有自身的进程 id。Net 域名空间允许每个容器拥有自身的网络组件，例如路由表，iptables，loopback 端口。IPC 域名空间提供各种 IPC 通信机制的隔离，这些隔离包括共享内存和消息队列等的隔离。Mnt 域名空间提供每个容器它自身的挂载点。UTS 域名空间确保每个不同的容器可以查看和改变他们的主机名。OpenVZ 通过这些域名空间可以创建一个拥有独立的 PID，IPC，FS，Network 和 UTS 空间的容器。

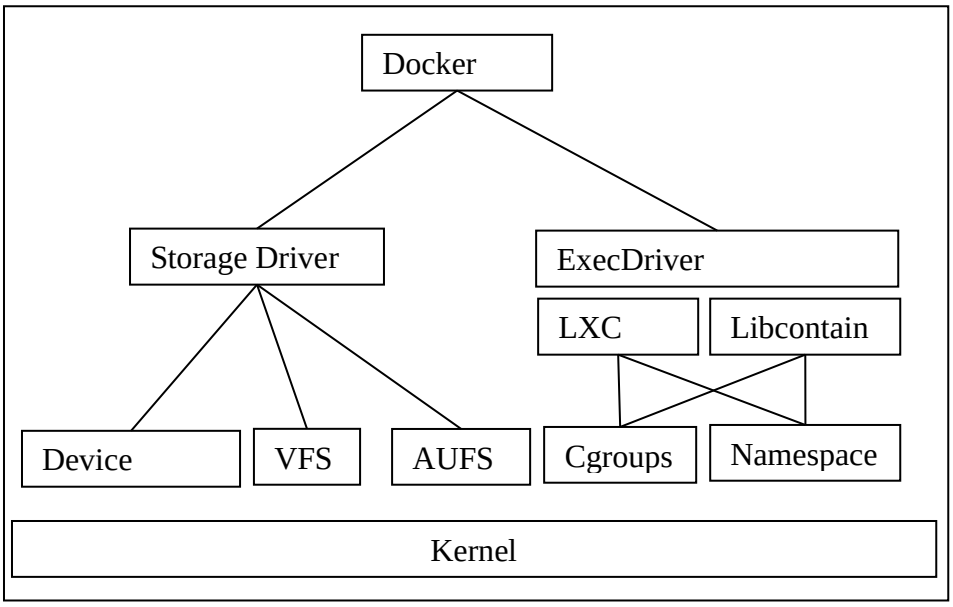

图 1.2 容器技术 Docker 架构

Docker 最早是基于 LXC（Linux Container）[25]来实现的，并在 2013 年三月开源。现在 Docker 已经自己实现一套 libcontainer 来取代 LXC，在配置中可以选择使用 LXC 或者 libcontainer 来实现某些内核操作，但默认配置已经更改为 libcontainer。Docker 的基本架构可以从图 1.2 中看到。LXC 和 libcontainer 实现

了对资源的控制和隔离，存储驱动实现了文件系统的 copy-on-write 技术和镜像管理。LXC 和 libcontainer 通过利用 linux 内核提供的 Cgroups 实现了对容器资源的控制。Cgroups 可以约束和记录一个容器使用物理资源（CPU,内存），这些组合形式组成大量子系统，每个子系统就是一个资源控制器。和 OpenVZ 一样，Docker 使用域名空间来提供容器之间的资源隔离。

Docker 使用的存储驱动机制是基于文件系统的 copy-on-write 技术和镜像管理来实现的，当前 Docker 底层的存储驱动主要支持三种类型的文件系统：AUFS，VFS 和 DeviceMapper。AUFS 是一种利用联合挂载的技术来产生新的文件系统，该技术可以将多个可读写的文件系统通过叠加的方式联合到另外的只读文件系统，生成一个看似一个的文件系统来提供给用户，Docker 将合并于最终文件系统的单个文件系统称做层，将最底下的只读层命名为容器镜像，就是这个原理实现了 AUFS 文件系统 copy-on-write 技术及其镜像管理技术，给 Docker 增加了更加强大灵活的运行机制。

Lmctfy 是 Google 在 2013 年 9 月开始开源的[26]。它可以通过控制和隔离进程资源。这些资源包括 CPU，内存和 I/O 带宽。Lmctfy 的基本架构如图 1.2，Lmctfy 由两层（CL1 和 CL2）组成。CL1 负责容器的创建和维护，并辅助 LC2，CL2 负责资源策略的设置。

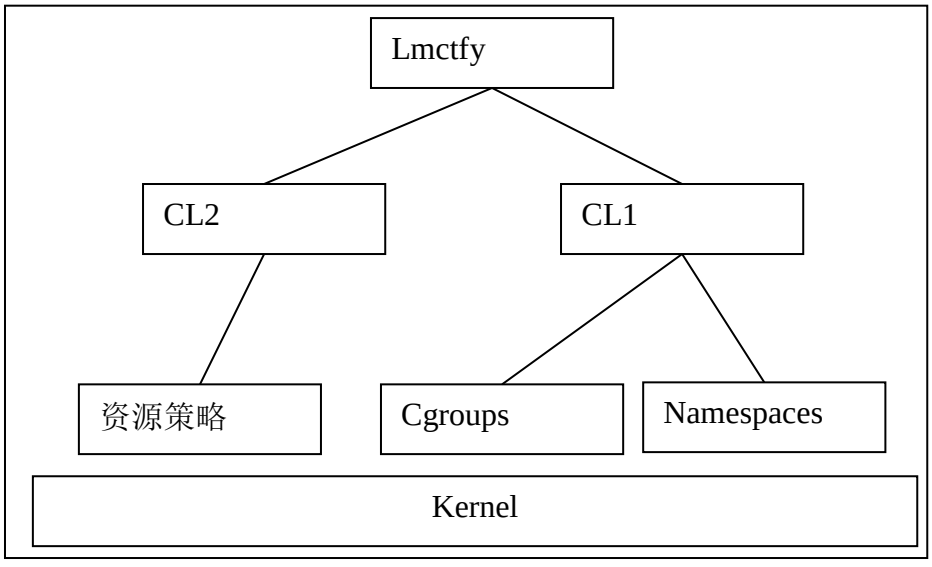

图 1.3 容器技术 Lmctfy 架构

Solaris Zones/Containers [27]项目开始于 2004 年，其目的是提供一个操作系统

级虚拟化的商业解决方案。在那时候，Sun 工程师对 FreeBSD 的 Jails 和 Linux 的 Vserver 解决方案进行了仔细的分析和总结，因为这两个项目的目标和他们的项目类似，都是深度的操作系统集成。另外，这些工程师希望可以创造一系列的可用区管理工具，这样可以使得可用区设置和配置可以被工具来替代。Solaris Zones 的隔离方案是基于将绑定一个空间标识符到一个进程，并利用该标识符限定进程的可视空间。这个空间标识符也被用作限制进程权限，防止其访问其他的非全局空间。资源管理使用 Solaris 内部的标准机制来实现，比如 entitlements（权限）, limits（约束） 和 partition（分区）。

在 Linux 中对类似 FreeBSD Jails 需求生成了 Linux-VServer 项目[28]。官方的第一个 release 版本是在 2003 年。在 Linux-VServer 中，不同的虚拟环境被描述为 VPSs（Virtual Private Servers），他们可以使用 util-vserver 包提供的用户空间工具来管理。每一个 VPS 拥有它自己的上下文，这个上下文中包含了所有与 VPS 相关的信息，比如 VPS 的名字，允许的权限，绑定空间，调度信息等等。另外，甚至每个 VPS 的行为都可以被通过指定上下文空间和标记来调整，这些标记可以用来修改 VPS 的主机名，隐藏网络端口。不过其最大缺陷就是需要使用 Linux-VServer 补丁修改 Linux 内核并重新执行编译指令得到新的内核。

1.3 本课题研究的主要内容

本课题主要的研究内容是利用 Docker 作为应用的载体，为中小企业编写一套能自动快速部署应用的 PaaS 云平台。该 PaaS 系统设计了一套自动选择物理主机机制，使每台物理主机的资源使用率趋于平均水平；设计了一套心跳检测机制，在物理主机宕机的第一时间侦测到，并将其从集群中剥离；设计了一套 Docker 主机自动注册机制，可以让 Docker 主机自动向系统中注册自己实现系统自动扩展。同时，该 PaaS 系统利用 Docker 容器作为应用的载体，使服务可以以容器的形式提供，其他应用可以依托于容器对外提供的功能，快速部署自己，从而让服务功能快速启动。该 PaaS 平台还可以灵活快速的添加定制服务，服务的添加方式与应用的添加动作一致。

本课题为了满足企业对 PaaS 系统的需求，具有如下几个创新点：

1.该 PaaS 系统使用 Go 语言编写，Go 语言编译后的程序性能与 c/c++的程序相当，部署的时候只需要将程序拷贝到服务器，可以直接运行，不需要配置运行环境，下载各种依赖包。对于中小企业来说，非常易用且高效。

2.使用 Docker 容器作为应用载体，对于开发者来说不再需要根据 PaaS 平台所提供的资源进行应用编写，开发者可以随意使用各种资源，在发布的时候，只需要将自己的开发测试环境打成容器镜像，将容器镜像放到 Docker 服务器，并执行容器创建命令，就会自动的启动新容器并运行应用。

3.使用 etcd 作为 Docker 服务器发现，可以自动监听 Docker 服务器的出现和宕机。

1.4 论文结构安排

本文介绍了云计算的背景，分析了 PaaS 平台的国内外现状和容器的国内外现状，最后提出了针对企业的的易部署易使用的 PaaS 系统。

后面章节会依照以下内容进行组织:

第二章介绍了该 PaaS 系统中将要使用的主要技术，有 Docker 相关激技术，主流的服务器调度技术，rethinkDB 和 etcd 的相关技术。

第三章对企业 PaaS 进行需求分析，明确了 PaaS 系统需要实现的功能，并对可能的主要问题进行了介绍，说明了用到的核心技术。

第四章说明了主要问题的解决方案，设计了企业 PaaS 系统的总架构，并对各个功能模块进行分类和设计。

第五章对各个功能模块进行了实现，对主要类和函数进行了仔细讲解，并附上部分关键代码。

第六章对通过运行整套系统，对系统的各个功能进行测试，并对测试结果进行相应的分析。

第七章对全文做出总结，并分析基于 Docker 的企业 PaaS 系统的不足以及今后要改进的地方。

第2章 相关研究工作

2.1 Docker 相关技术

2.1.1 容器的组成

容器是在 linux 环境中提供隔离和资源一种手段。一个操作系统容器需要提供进程与操作系统其他部分的隔离，当然也包含该进程的所有子进程。容器的根本意图是提供与虚拟机一样的隔离级别和安全性的基础上保持与主机系统的轻量级整合。不依赖于硬件模拟的特性使其拥有比全虚拟化更好的性能特性，同时也限制了它对操作系统多样性支持。在 PaaS 系统中，对操作系统的多样性是无需考虑的，因此容器可以很好的提供低损耗隔离特性。

Chroot（change root）是 linux 命令[29]，它可以修改当前进程和其子进程的 root 目录到新的目录。一些容器使用 chroot 在虚拟环境中隔离和共享文件系统。

Cgroups（change groups）是 Linux 内核子系统[30]，它提供了很好的共享资源授权控制，比如 CPU，内存，进程组等等。目前容器技术一般都使用 cgroups 作为系统资源管理机制，通过 Cgroups 设定这些资源的使用量限制。

Kernel namespaces 域名空间可以创建两个容器之间各个隔离级别的栅栏[31]。Pid 域名空间允许每个容器拥有自身的进程 id。Net 域名空间允许每个容器拥有自身的网络组件，例如路由表，iptables，loopback 端口。IPC 域名空间提供各种 IPC 通信机制的隔离，这些隔离包括共享内存和消息队列等的隔离。Mnt 域名空间提供每个容器它自身的挂载点。UTS 域名空间确保每个不同的容器可以查看和改变他们的主机名。

2.1.2 该 Docker 的内部结构

Docker 是一个为开发者和系统管理员建立的一个开源平台，它可以构建，发布，运行分布式应用。Docker 内部主要分为四个模块：Docker 引擎，一个提供便携的轻量级运行时环境和打包的工具，Docker Hub，一个共享的应用程序和自动化工作流程的云服务。Docker 使应用程序可以快速的通过组建来装配得到，免去了开发者，QA 和生产环境之间的冲突。因此，它可以快速发布并在笔记本，数据中心的虚拟机上和云平台上运行，这一切都无需改动程序。

Docker 内部的各个组成模块主要有：Docker Client、Docker Daemon、Docker Registry、Graph、Driver、libcontainer 以及 Docker container[32]。

Docker Client 是 Docker 架构中用户用来和 Docker Daemon 建立通信的客户端。它是一个可以控制管理 Docker 容器的基于命令行的程序，可以通过它运行和配置 container 镜像的启动操作。

DockerDaemon 模块是用于代表 Docker 对外提供各种服务的一个服务进程，它以守护进程的形式存在，可以接受 tcp 请求，http 请求，它的主要任务就是接受并处理 Docker Client 发送的请求。它的工作原理是收到 Client 端发送的消息请求之后利用内部的路由机制，讲消息转发到对应的已经注册的 Handler 来处理该消息。

Engine 模块是是 Docker 系统中最重要的一个模块，它扮演了引擎的角色，所有的任务都经过它来封装成多个 job 执行，job 肯能是运行容器，也可能是停止容器，镜像的 pull 操作也由 engine 生成一个 job 来执行。

Graph 模块是用于将 Docker 系统中已经存在的 Docker 镜像进行管理和维护，它包括镜像的存取操作和存储多个镜像间的依赖关系。

Libcontainer 模块类似于 LXC[33]的一个模块，通过它可以直接调用 linux 内核中的 API，这些 API 包括管理容器的 cgroups，namespace，network，iptables 等，Docker 通过它可以不使用 LXC 而操作容器，这就减少了对第三方包的依赖。

2.1.3 该 Docker 的优势

当前流行的容器主要有 LXC，Warden, Docker, OpenVZ，LXC 虽然已经非常成熟，但是它只提供容器的功能，使用复杂，并且受限与 linux 平台，也没有做到容器之间完全隔离，例如时间。Warden 是 Cloudfoundry 自己开发的基于 cgroup 的一种容器[14]，但是隔离级别较低。OpenVZ 需要修改内核，给内核打补丁。

Docker 的理念是提供一个简单的工具来管理内核技术，比如 libcontainer，LXC，cgroups 和 copy-on-write 文件系统。这使得一些以往的开发模式，系统管理员的游戏规则产生了一些变化，开发者可以直接操纵内核功能而无需知道其复杂的实现。

Docker 使用 libcontainer 来控制 cgroups，同时使用域名空间来隔离主机与容器。使用 AuFS 作为容器的文件系统。AuFS 是一个层级文件系统，他可以覆盖到一个或者多个已经存在的文件系统。当一个进程需要修改一个文件，AuFS 就会创建这个文件的拷贝，这里就实现了 copy-on-write[34]。AuFS 最大的亮点是容器镜像版本控制，这个方式可以在新的镜像中只保存差异部分，从而减少空间

损耗。

表格 2.1 显示的是当前容器的使用状况，可以看到，当前的 PaaS 系统中，Docker 使用非常少。但是在 2014 年 6 月 Docker 发布 1.0 之后，微软，亚马逊，IBM ，Rackspace ，Google，Red Hat，VMware 都使自家的产品对 Docker 开始有了支持，加入了 Docker 的阵营[35][36]。

表 2.1 常见开源 PaaS 系统与使用容器

PaaS 提供者	实现支持
OpenShift	Docker with LXC
Heroku	LXC
CloudFoundry	Warden Container
Stackato	Warden Container（基于 Cloud Foundry）
AppFog	Warden Container（基于 Cloud Foundry）
Virtuozzo	基于 OpenVZ
dotCLoud	Docker with LXC

2.2 rethinkdb 相关技术

2.2.1 rethinkDB 的概述

RethinkDB 类似于 MonGoDB 的面向文档的数据库[37],但旨在克服可扩展性和后者的实际限制。 RethinkDB 对文档的存储是 JSON 格式来实现存储的，它可以使用服务器集群进行分布式存储,它支持真正有用的查询如 joins 和 group by 的查询语言，并且易于安装和学习。它可以用来替代 Memcached 作为数据缓存层。或者替代一些类似 Memcached 的存储，比如 Membase，MemcacheDB，TokyoTyrant 及 Schooner Membrain 等[38]。RethinkDB 与很多其他关系型数据一样，可以在高压力情况下仍然拥有高性能，同时还保持高可靠性；因此，它非常适合在某些地方使用。

由于它对 SSD 做了特别的优化，因此它可以解决很多一般方案不能解决的负载能力问题，比如数据量非常庞大，已经大大超出了内存所拥有的存储量，但系统仍然需要拥有较高的性能的场景。在硬件资源无法改变的情况下，希望硬件有能力胜任更加强大的负载能力的场景下，可以使用 rethinkDB，它不仅可以提升缓存的性能，而且可以提升键值存储效率。

2.2.2 rethinkDB 的存储特性

RethinkDB 在早期的时候是作为一个 MySql 的一个存储引擎，该引擎是经过对固态存储专门优化的结果，可以让固态硬盘的利用率达到最高。RethinkDB 现在已经发展为一个单独的存储系统，可以非常快速的对 json 文本进行存储和检索，是一个高效的文档型数据库。

固态硬盘有两个特性：首先，由于没有机械部件，因此消除了旋转等待时间和臂争用，使得它们的随机存取性能显著比旋转的好。其次，由于存储阵列的物理设计，随机写入要求结算页数和再次复制，使得这些操作昂贵，并且降低了驱动寿命。第一个特性使数据的空间局部性相关性较低的性能，消除了需要这样的传统的数据组织如聚簇索引和数据结构空间局部性优化诸如 B-树。第二个特性使得传统的索引技术效率较低，因为 B 树节点设计的地方进行修改，与昂贵的随机写入的固态硬盘相冲突。鉴于这两个特性，需要追加唯一方法来存储数据，通过日志结构文件系统创建。此外，rethinkdb 设计了一个仅追加索引机制，允许索引和未索引数据的一致好评高效的性能。追加的唯一办法是由可通过廉价的随机读取和缺乏必要的聚集索引，并且是由需要通过昂贵的随机写入的固态硬件。高效的仅追加存储打开了一些利润丰厚的特性，不能轻松有效地采用传统的存储引擎实现的可能性。

2.3 etcd 相关技术

2.3.1 etcd 技术的使用

Etcd 是一个分布式键值存储器，它提供了一种可靠的方式在一个计算机集群上存储数据，主要的思想来自 ZooKeeper[39] ，还有一些来自 Doozer。Etcd 可以优雅的处理主节点选举，并容忍网络分区中某些节点的故障。应用程序可以轻松的向 Etcd 读取和写入数据，这一切都通过基于 HTTP 协议的 JSON 格式数据交互。etcd 集群的工作原理基于 raft 共识算法 (The Raft Consensus AlGorithm)[40][41]，它可以保证集群中数据的一致性，当某些节点发生宕机或者无法正常工作时，集群会通过 raft 算法来重新选举新的 leader，新的 leader 可以作为对外的主服务节点，当然也可以从其他节点中获得键值对信息，但是在普通节点中无法获得集群信息，比如集群中有哪些节点，该消息只能由 leader 节点得到。在本文中，使用的是 etcd 服务注册和服务发现的功能，利用它来发现新主机或者主机宕机。使用方法如下图，服务提供者向 etcd 注册自己，并将一些信

息存入 etcd 中，服务发现者会去请求有哪些服务可用，并得到所有可用的服务列表。在本系统中可用服务注册就是 Docker 服务器节点自己想 etcd 注册自己，说明自己可用提供容器创建工作。

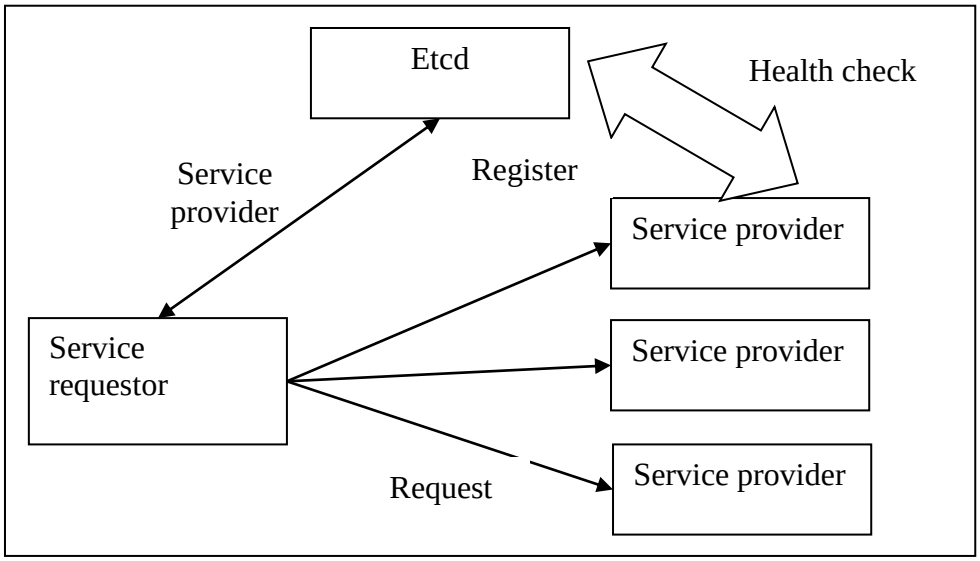

图 2.1 利用 etcd 作为服务发现的框架

第3章 基于Docker的企业PaaS系统模块的需求分析

3.1 业务需求

基于Docker的企业PaaS系统是一个针对于中小企业用户的微型PaaS系统。通过该系统，企业可以将已有的几台内部服务器进行轻松的搭建为私有PaaS系统。通过该系统，可以将多台服务器合并为一个计算和内存池，并对这些资源通过使用 Docker 创建容器来实现资源的二次划分，从而使得所有容器应用都在被分配的资源中运行，并且容器之间的资源被几乎完全隔离。同时也实现了用户利用 Docker 镜像，将应用轻松部署，快速部署，一致性部署。

3.2 用户需求

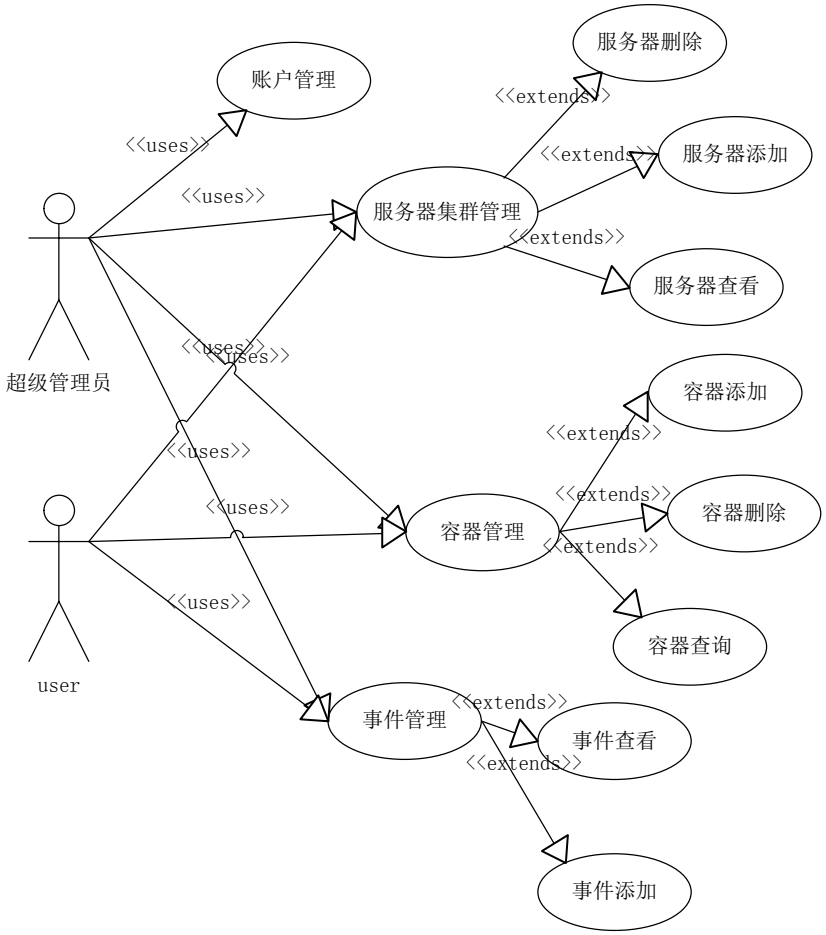

图 3.1 用例图

如图所示，管理员可以借助该系统对账户或者对集群进行管理。

管理员需要能通过账号密码的形式登入 PaaS 系统，查看服务器集群的资源和运行状态。当集群出现人为事故，管理员需要查看历史操作记录。当需要部署应用时，管理员能利用自动化部署来完成批量部署操作。

账户管理:超级管理员可以通过指令对 PaaS 中的用户账号进行管理和维护，比如添加和删除操作。

容器管理：管理员可以对容器进行增改查，以此来实现 PaaS 的目的。

事件管理：管理员需要查看所有操作的历史记录，使其能够快速查看历史操作的正确性。

查看集群节点信息：管理员需要能够查看某节点信息，可以了解某节点信息从而判断故障原因。

服务器集群管理：管理员可以对集群内地节点进行手动删除和添加，从而手动调整自动化添加到一些故障。

3.3 功能需求

基于 Docker 的企业 PaaS 系统的功能需求如下:

账户管理：对 PaaS 系统的管理员账号进行管理和维护，比如增加和删除。

查看历史操作：依据时间对 PaaS 系统的事件日志进行排序显示。

查看集群信息：查看 PaaS 系统的资源状况，比如 CPU 使用率，系统内存使用率等信息。

集群节点管理：对 PaaS 系统中的 Docker 服务器进行管理和维护，运行 Docker 服务器热添加和热删除。

应用管理：运行 n 个容器应用，停止容器应用，删除容器，查询容器应用。

节点监控:使用健康监测模块对 PaaS 系统中的所有 Docker 服务器节点进行跟踪和检测。

3.4 非功能需求

基于 Docker 的企业 PaaS 系统将 Docker 部署于提供服务的服务器中。在 Docker 执行容器部署的过程中，不能影响其他容器应用的 cpu 资源和内存资源。

在 PaaS 系统中添加新主机后，应该在秒级内能加入到 PaaS 系统资源池待用。

PaaS 系统需要拥有简洁的使用方式，使用客户端可以操作集群中的各种功

能。

将客户端与控制器分离，能够提供 remote API，remote API 用于二次开发其他客户端。

PaaS 系统的服务器节点配置最低要求是单核 cpu，512M 内存的主机环境。

单个应用的部署需要秒级执行。

3.5 关键问题

3.5.1 实现系统的主要问题

服务器节点调度。PaaS 系统需要管理多台 Docker 主机，让这些主机协调执行应用部署任务。因此，系统需要一个高效简洁的调度算法来协调 Docker 主机集群工作。传统的调度多是针对计算资源调度或者单独针对磁盘资源进行调度，但本系统需要同时考虑计算资源和内存资源，好比同时考虑任务调度和磁盘调度。本文需要设计一个多种资源统一分配的问题，很多算法都是实验性质的，需要设计一个方便简洁适合中小企业 PaaS 系统的调度算法。该算法需要能综合考虑计算机的 cpu 资源和磁盘资源，在节点调度的时候能根据每台 Docker 主机的实际使用情况给该台主机进行评分操作，需要配置合适的 cpu 和磁盘资源的权值，得到一个比较公正的调度算法。从而使每台服务器能高效快速的运行。

监控服务器存活。PaaS 系统需要管理多台 Docker 服务器，在长时间运行过程中某台主机可能会突然停止运行，如果 PaaS 系统没有感知到该服务器的宕机则会继续让该服务器提供服务，后果很严重。因此，需要设计一套检测集群中所有主机的健康状态，在主机停止工作后及时将其从服务列表中删除，避免继续被分配任务。该部分的难点是如何高效的对每一台已注册的服务器进行跟踪，如果设置检测频率太高会使工作服务器效率下降，如果检测频率太低会使检测宕机的实时性降低。因此选择一个合适的检测间隔也是非常重要的一个问题。

访问权限控制。由于选用的语言没有现成的 restful 框架，也没有访问权限控制框架，主要问题就是需要自己根据其他语言的模式来自己实现访问权限控制的设计。需要能够根据 restful 的设计理念，加上对请求消息的路由控制匹配到相应的认证函数上，使该函数能够该消息的头部进行认证，认证通过再将该消息转给真正的处理函数。

批量创建应用容器。在 PaaS 系统中，固然会有需求需要同时批量创建同一个镜像的多个应用容器。如果每次请求只能创建一个，那效率非常低，而且对

于操作者来说也是非常麻烦的。但是目前 Docker 自身只支持一次请求只启动一个容器，况且在本系统中 Docker 主机是有多台的，如果可以多台主机同时创建，将大大提高系统的性能。所以设计实现一套可以让多台 Docker 主机并发创建指定应用容器数量，并保证每台机器负载都差不多是一个非常复杂的问题。

3.5.2 本系统的核心技术

使用 etcd 作为 Docker 主机注册中心：使用 etcd 来提供 Docker 主机的注册环境，在 etcd 中存有活动状态的所有主机，服务注册方式使用 http 远程注册方式。编写监听器，实现对注册于 etcd 中的 Docker 服务器节点进行监控。当某个节点停止工作后，监听器能迅速侦查到，并将其从健康目录中删除，通知集群控制器停止使用该服务器节点。

权限认证和控制：编写框架，实现对 http 处理前进行权限认证。

Docker 容器调度技术：本文实现了一个批量部署容器应用的算法。设计了一套算法，能够改变已运行的应用的部署个数。在需要拓展的时候，可以设置加大某个应用的部署数量。在需要减少的时候，可以自动删除一些应用容器，减少部署数量。

3.6 本章小结

本章节对企业 PaaS 系统进行了充分详细的需求分析，阐述了完成该 PaaS 系统需要实现的功能以及包含这些功能的模块。同时说明了在实现过程中有可以遇到的一些主要问题，并简要提出了解决针对于这些主要问题的关键技术。为读者更好的理解和把握本系统提供了比较完善的说明，同时也为下文要描述的 PaaS 系统设计，实现和测试做了较平滑的铺垫。

第4章 基于Docker的企业PaaS系统的功能模块设计

本章将对本 PaaS 系统的各个功能模块进行详尽的设计描述，同时根据具体需求，分析系统的功能模块，设计系统的整体框架。结合 PaaS 系统设计范围，本系统设计为两大部分，第一部分为 PaaS 系统客户端，第二部分为 PaaS 系统服务器端。又将 PaaS 系统服务器端程序设计为如下模块：http 服务模块、登入和认证模块、注册和心跳检测模块、账户管理模块、服务器管理模块、容器控制模块、事件管理模块、主控制器模块。在确定了系统的模块组成之后，依据现有的技术要求，将对系统每一个模块在详细设计部分用时序图，类图等进行说明。通过本章的讲解，读者将会对该 PaaS 系统有一个全面的了解，系统各个模块的功能以及各功能是如何实现的，从系统最初的需求分析，功能确定到最终的设计整个流程。

4.1 总架构设计

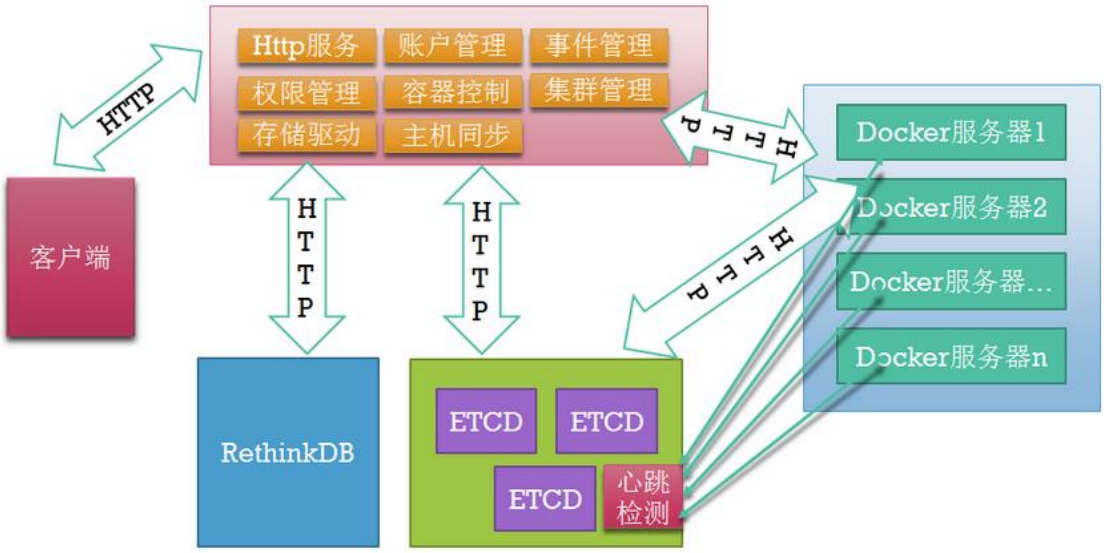

图 4.1 系统结构图

基于 Docker 的 PaaS 系统主要分为三块内容。第一块内容是客户端，用于控制 PaaS 系统的各项操作。第二块内容分两部分，第一部分是 PaaS 系统的主要控制中心，负责应用部署，账户管理，服务器集群管理，事件监听等任务。第二部分是 etcd 注册服务器和心跳检测模块，用于 Docker 服务器向 PaaS 系统注册自己，心跳检测模块用于检测已注册的 Docker 服务器的健康程度。第三块内

容是 Docker 服务器，Docker 服务器提供应用部署的实体，所有应用都通过 Docker 来创建容器，并在容器中部署各个应用，并提供应用服务。

PaaS 控制器的内部模块图如下:

PaaS 控制器将分为 http 服务模块，权限管理模块，账户管理模块，容器控制模块，服务器管理模块，事件管理模块，主控制模块，心跳检测模块，他们的具体设计说明将会在后续小节中说明。

图 4.2 本 PaaS 系统控制模块内部结构图

4.2 关键问题解决方案的设计

1. 服务器节点的调度设计

PaaS 系统中应用部署需要选择合适的服务器节点，目的是选择负载最低的服务器来部署应用，从而达到集群内所有服务器负载最低。本调度策略采用静态调用，不需要实时获取每个容器的资源利用情况，可以降低 PaaS 系统本身的 cpu 和内存的使用量。

Docker 主机的选择实行得分制，对 Docker 集群的所有活动主机进行分数评估，然后对服务器节点根据评分进行排序，选出分数最高的服务器作为应用部署的载体。计算方式如下:

对于每一个节点，有两个关键参数，一是 cpu，二是内存。一般情况下，Cpu 和内存的重要程度也不一样，一般内存可扩展性强，比较量大。因此，在权重划分的时候，将 cpu 权重置为 65，将内存权重置为 35。cpuScore 的值为（1- (已经分配的 cpu+该请求需要的 cpu）/ cpu 总量)*65。memoryScore 的值为（1- (已经分配的 memory+该请求需要的 memory）/ memory 总量)*35。total 的值为 cpuScore+memoryScore。这里，cpuScore 为服务器节点剩余的 cpu 所得到的分数，memoryScore 为服务器节点剩余的内存获得的分数，total 为该服务器节点获得的总分，用于评估其可以用与部署的权重。

2. 监控服务器存活

监控服务器的目的是为了系统部署时因无法访问集群节点而出错。

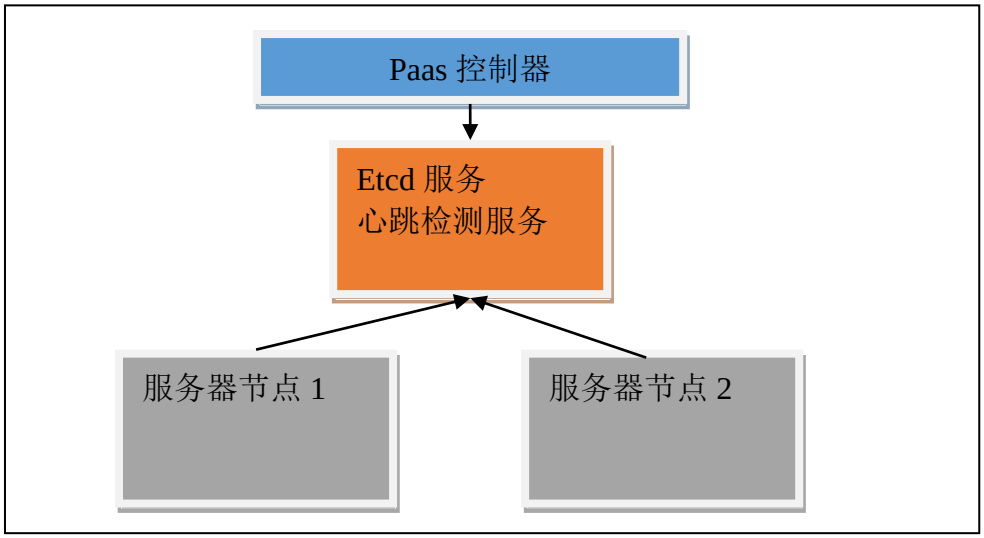

图 4.3 心跳检测框架

监听服务器存活最简单的方法是定时的对每一台服务器节点进行访问。但是在 Go 语言中并没有 ICMP 包，需要自己通过 ip 协议来实现 Ping 工具。以下是对 ICMP 的研究分析：ICMP 的中文名字是控制报文协议，它可以用于检测目标主机是否可以使用 ip 来连通，它的数据结构如下:

回送消息[ECHO]

Type = 8	Code = 0	CheckSum
Identifier		Sequence Number
data		

回送响应消息[ECHO REPLY]

Type = 0	Code = 0	CheckSum
Identifier		Sequence Number
data		

这数据结构中 type=8 表示为发送消息，type=0 则是说明该消息是主机返回的消息。检验和 checksum 需要将整个消息的数据结构进行校验得到的结果，用于在主机收到后查看数据是否正确。Identifier 用于标识是哪一个 ICMP 消息。

3.访问权限控制

为了阻止一般管理员进行一些与职位无关的访问操作，需要对 PaaS 设立访问控制模块。可以分别从是否登入和是否有访问目录权限两方面进行控制。由于 Go 语言自身没有级联的访问框架，下面将设计一套可以连续执行的多级过滤器。

首先建立一个链表，每个节点挂一个处理 request 的 handler。将多个处理 handler 挂载到这样一个链表中，并且按照处理次序设置完成。在 n 节点进行处理 request 的时候，先对 request 正常处理，该级处理完成之后，在处理末尾访问 n+1 节点的处理函数，同时将 request 作为参数传入。这样递归一次，所有的节点就会都处理一次 request。

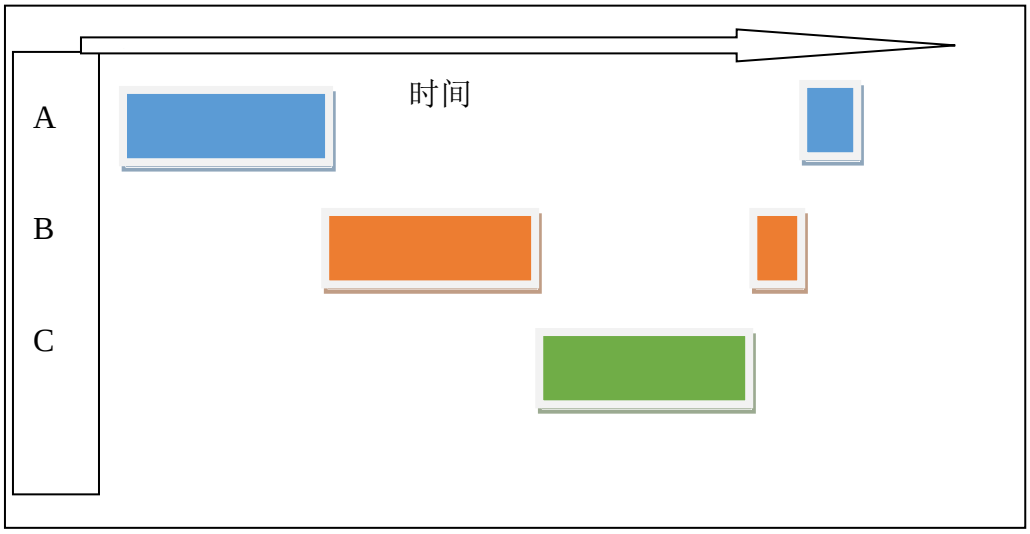

图 4.4 请求链运行过程

在第一级将引入检测是否登入状态，即是否带有已经被分配的 token，以及该 token 是否合法。在第二级将引入 remote api 访问路径控制，针对每个级别的管理员设置其能访问的路径集合。Admin 管理员可以访问 PaaS 系统所提供的所有服务路径，user 级别的管理员能访问除了账户管理相关的所有服务路径，这些路径都存于用户访问表中。

4. 批量创建应用

批量创建应用在 PaaS 系统中是一般都会实现的一个功能，也可以说是 PaaS 系统的核心功能之一。比如同时启动 100 个 tomcat 应用。如果手动逐个添加是非常缓慢的。更进一步，如果使用批处理方式，使用循环函数逐个添加应用。由于每个应用的启动需要一定时间，在循环中只能等待一个应用创建完毕之后

再创建第二个。显然这种循环方式会随着应用部署个数增加而线性增加，等待时间过长。为了更有效的使用系统资源，这里将使用协程对应用部署进行并发部署。为了将部署任务分散到多个服务器同时进行，使用了逐次评分选择服务器来执行部署任务，加快了 PaaS 系统的批量应用创建工作。

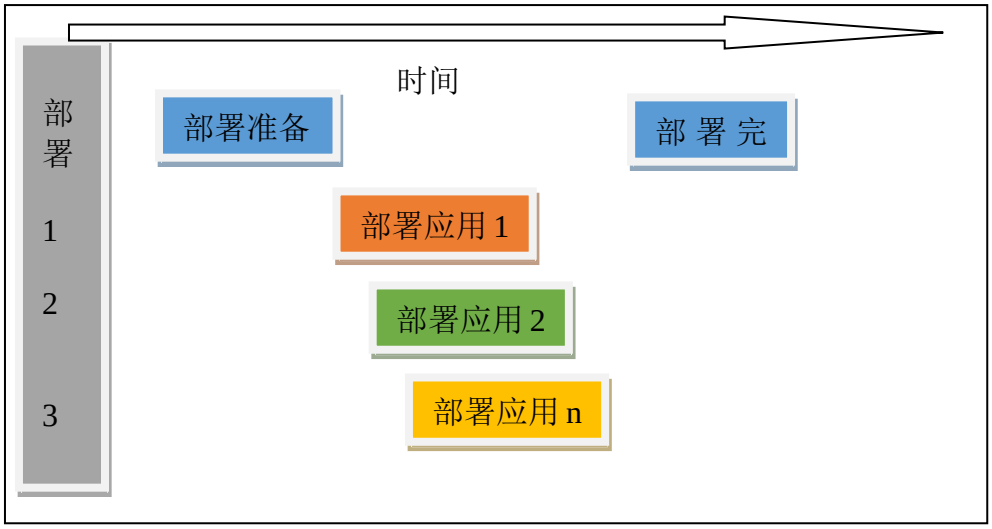

图 4.5 批量部署例子

4.3 系统各功能模块的设计

4.3.1 系统 http 服务模块的设计

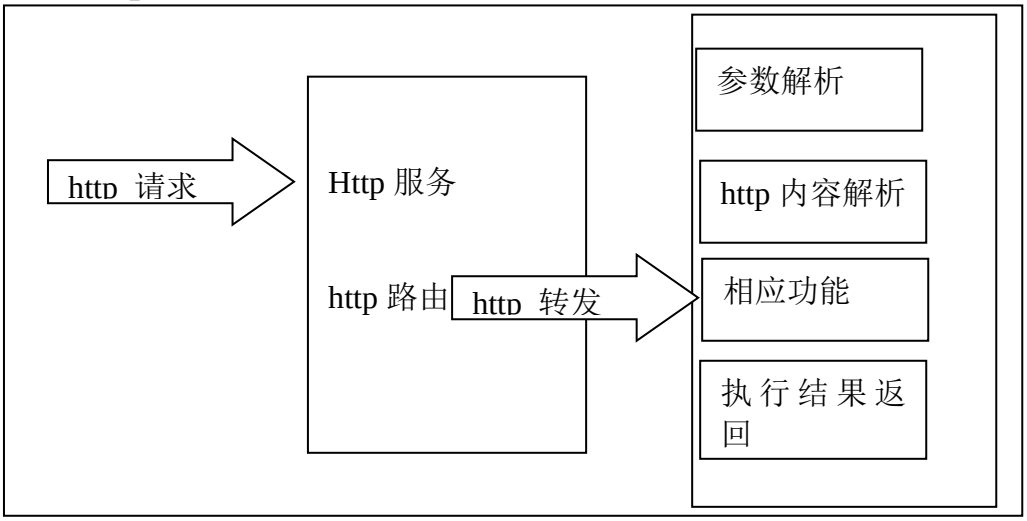

图 4.6 系统 http 服务分析图

http 服务模块的主要功能是监听服务端口，同时接受 http 请求，并把 http

消息请求根据 request 头部的路径转发给对应的功能处理模块。这些功能处理主要包括 http 请求参数解析，http body 解析，实体函数功能的调用，返回执行结果。

在 Go 语言中 http 包监听 http 的消息请求是这样操作的：对于开发者，设置端口唯一需要做的事就是调用 ListenAndServe 这个函数来设置监听端口的，并在参数中传入路由器。端口监听部分函数内部执行过程为它在内部实现了一个 server 实体，然后利用 server 的方法 Listen（tcp 协议，地址端口）来实现对端口的监听，实际上就是使用了 TCP 来实现了一个服务发布。

经过以上操作已经可以监听端口，下一步操作是利用 Serve 函数来设置接受发送到该服务器的消息请求。在 Serve 函数内部有一个无线死循环，在循环内部会对将 Listener 接受得到的请求创建一个连接 Conn，然后对 Conn.serve()开启一个协程，将这次请求的内容作为参数传入协程中，这样，每当一个请求到达之后就会启动一个新的协程去处理，完全达到了高并发的需求。

http 包处理消息请求的过程为：由于 socket 流式传输，因此可以通过读操作将 Client Socket 中的数据读到 HTTP 协议段中的 header 区域，然后检测 header 中消息是属于哪一种方法，若是 POST 请求，需要继续读取 HTTP body 中的数据，然后将 header 和 body 整合为一个 request 交个专门处理请求的 Handler。Handler 获取到 request 对象后会对它进行分析并调用相关函数，最后利用 Client Socket 将数据写到流中返回该消息请求。

正常情况下，在路由完成调用相应函数的时候，需要在函数内部判断请求方法，这增加了函数内部的复杂度。为了在路由的时候使用请求方法来进行路由，使用了自定义路由器。也就是将上文的 DefaultServeMux 用自己的 mux 来做替换。

编写自己的 mux 之前需要先分析官方默认的 mux，经分析得知，路由的功能有两个：第一个是路由表，记载了 http 请求路径及其需要转发到的目的 handler 的关系；另一个是根据路由表信息将消息请求分发到相应的功能函数来处理。

Go 默认的路由添加是通过函数 http.Handle 和 http.HandleFunc 等来添加，底层都是调用了 DefaultServeMux.Handle(pattern string, handler Handler),这个函数会把路由信息存储在一个 map 信息中 map[string]muxEntry，这样以上的第一个功能就被实现了。

在监听函数监听端口收到 tcp 连接请求之后会丢给 DefaultServeMux 处理，

DefaultServeMux 使用 ServeHTTP 函数来接受 HTTP 包监听到的 request 请求，并把 responseWriter 的 handler 也作为参数传递给 ServeHTTP 函数。在该函数内部，它会搜索含有请求路径的 map，如果找到该请求路径，则调取 map 对应的功能函数来执行处理该请求。Mux 的第二个功能就是如此操作的。

这里的 mux 主要采用了 Gorilla 的设计。这个 mux 的主要特性有：1.请求可以被根据各种方式来匹配路由。包括主机的 URL，路径，路径前缀，http 头，请求值，http 方法，也可以使用自定义匹配器。2.主机 URL 和路径使用一个变量来表示，该变量内部含有正则表达式，可以根据表达式来进行路径匹配。3.路由器可以被用作子路由器，这对定义一个路由组非常有帮助，每个路由组可以共享一些常用配置，比如主机，路径前缀或者其他重复的属性值。

在本 PaaS 平台中主要运用 http 请求方法来过滤消息，即在每一个路径路由的时候给其添加方法的过滤，如 GET，POST，PUT，DELETE 等。

4.3.2 登入和认证模块的设计

登入认证模块分两部分内容，一是账户登入认证，二是用户访问权限认证。

账户登入认证是账户在登入之后会给其分配一个 token，用户只要使用该 token 与 PaaS 系统进行通信，才能访问 PaaS 系统的功能模块。

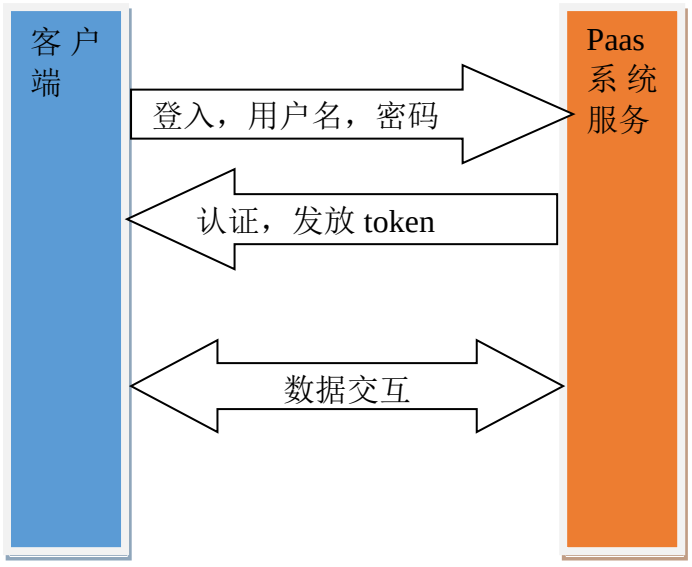

图 4.7 登入过程图

在服务器端，密码的存储使用了 sha1 哈希算法进行了转换，该算法具有不

可逆性，也就是说无法通过转换后得到的字符串恢复密码，将转换后得到的字符串作为密码存储存储在服务器数据库。此方式确保了 PaaS 系统中没有真正的密码。

用户访问权限认证会使用一个 map 来存储每一级别用户的权限，在本 PaaS 系统中，将 admin 用户的权限设置为所有，将一般 user 的权限设置为某些目录的访问权限。这要求在请求进入路由处理前就完成权限认定。

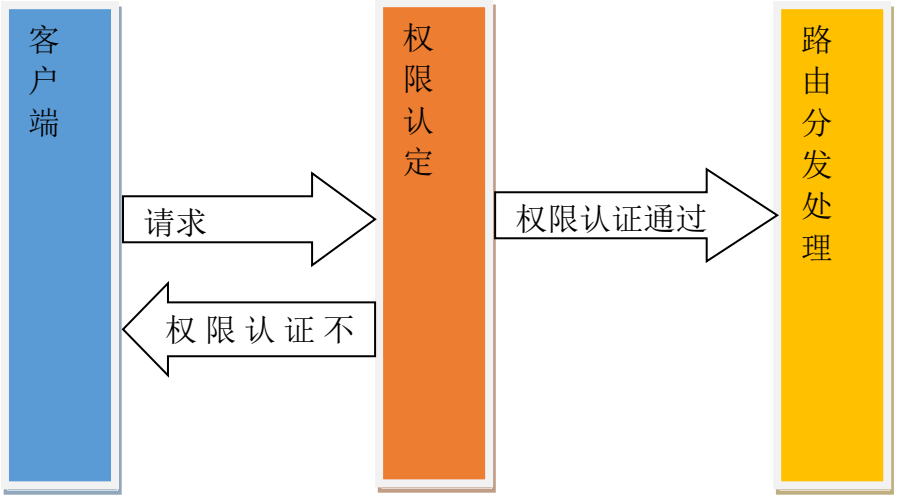

图 4.8 权限认证图

4.3.3 注册和心跳检测模块的设计

心跳检测模块用于监测 PaaS 系统中的所有 Docker 主机的健康状况。服务器节点在启动之后会向 etcd 服务器将自身注册到 PaaS 系统中，注册的主要数据为 ip 地址及 Docker 服务端口，cpu 数量，内存容量。心跳检测程序会定时向 etcd 服务器拉取服务器节点数据，并将它们放入自身的检测池中。

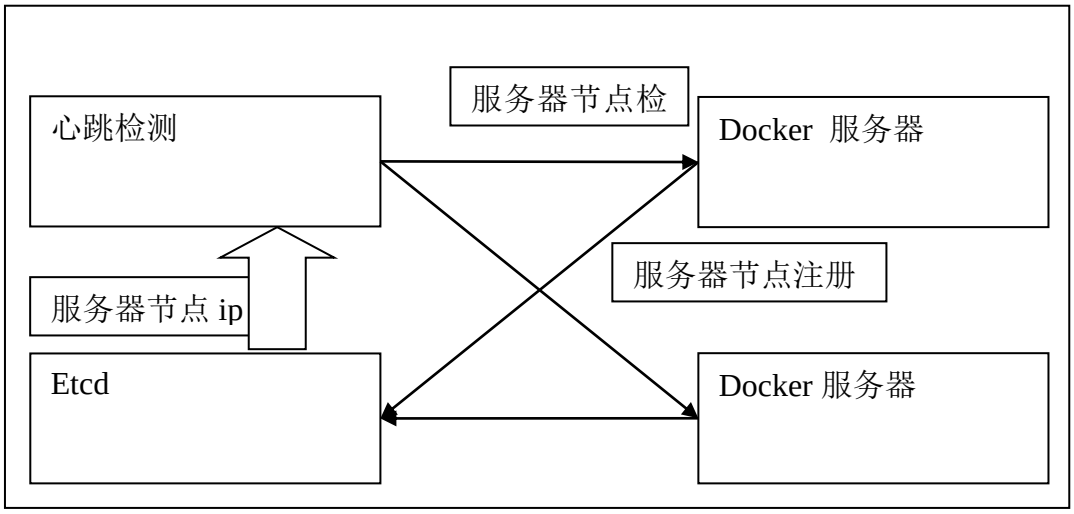

图 4.9 注册和心跳检测框架

心跳检测程序内部由主程序获得所有服务器节点的地址。然后对每个服务器开一个协程，让各个协程独立运行。当部署完成之后，主程序将进入休眠状态或者死循环状态，在协程内部，有一个滴答式定时器，该定时器每隔一定时间会唤醒程序执行程序。这里的唤醒使用到了 Go 内部的协程通信机制: channel。channel 就是通道，用来传递协程之间的数据。

这个滴答定时器在新建的时候开启一个协程，用于时间记录。同时也不影响原程序的正常运行，在时间到达的时候，通过 channel 将通告发送会原程序段。若源程序段有接收语句，则在该消息未到达之前，程序段会进入阻塞状态。在滴答定时器的通知到了之后，程序段接受该通知，并往下执行程序。滴答定时器重新进入新的定时周期，在下次时间间隔到达之后再次呼唤。

服务器处理程序主要通过 ping 程序来检测服务器 ip 是否可达，若可达，则继续执行，若不可达，则等待一段时间后再次检测是否可达，若还是不可达则进入不可达程序处理。不可达处理程序会将已经处于宕机的 Docker 节点从活动主机列表中删除。

4.3.4 账户管理模块的设计

账户管理模块拥有三个功能，就是对账户进行查看、添加、删除操作。其中，账户查看功能会将 PaaS 系统中保存的所有账户的基本信息返回；添加账户会先检测数据库中是否已经有该用户名，然后在没有相同用户名的情况下，向数据库中添加该账户。删除账户会将该用户的数据从数据库中清除，该账户就

无法使用该 PaaS 系统。账户的数据格式为如下:

Account struct 账户类

ID string

Username string

Password string

Tokens *Auth.AuthToken

Role *Role

Role struct 角色类

ID string

Name string

AuthToken struct token

Token string

UserURL string

其中 ID 为字符串类型，Username 是管理员在系统中的登入名，PaaSword 是管理员的登入密码，Tokens 存储了管理员在登入 PaaS 系统的时候分配给他的 token，Role 是管理员的角色类型。

查看账户会首先把消息发送到 PaaS 系统的 http 服务模块，服务模块直接将 http 请求交给路由器。路由器首先会将请求交给登录认证模块，检测该用户受否有 token，该 token 是否与数据库中的 token 一致。当确认该用户已经登入之后将该请求移交给权限检测模块。权限检测模块会根据登入用户的用户名查询该用户的角色。并比较用户请求的访问路径是否在该用户对应角色的可访问路径表中存在，若不存在则返回权限错误信息，若存在则继续将该请求转发给下一级 handler，也就是账户管理模块。账户管理模块收到该请求，会从数据库中查询账户表，获取所有账户的信息，并将它们转化为账户对象数组返回给用户。

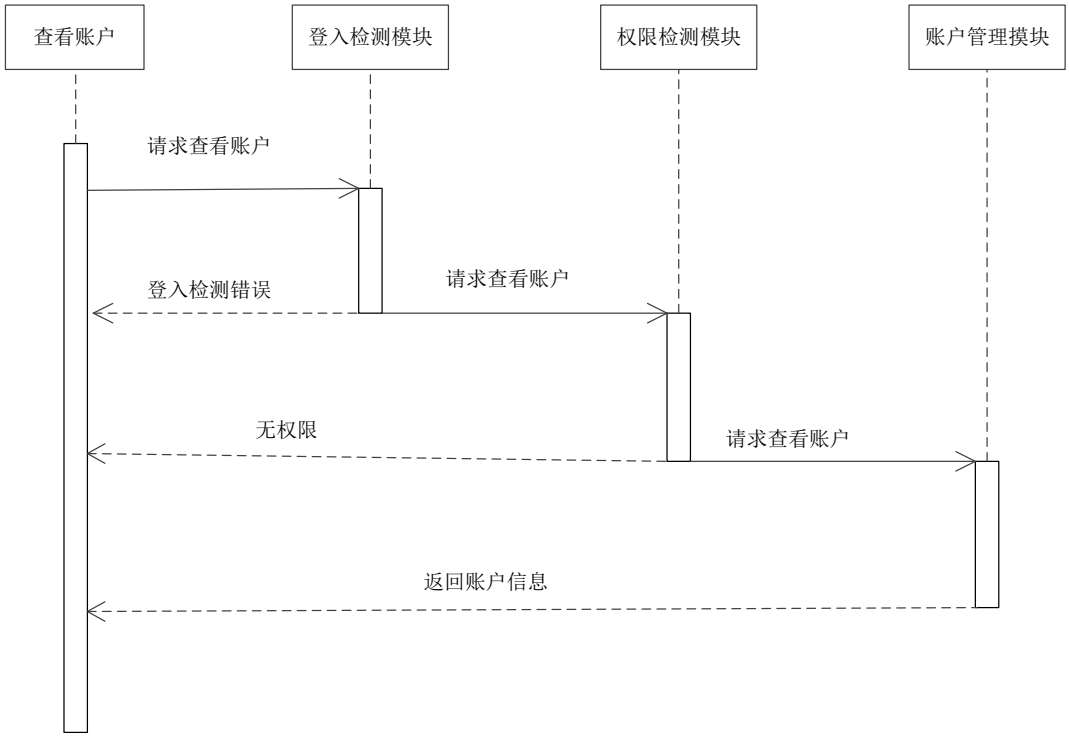

图 4.10 查看账户顺序图

添加账户会首先把消息发送到 PaaS 系统的 http 服务模块，服务模块直接将 http 请求交给路由器。路由器首先会将请求交给登录认证模块，检测该用户受否有 token，该 token 是否与数据库中的 token 一致。当确认该用户已经登入之后将该请求移交给权限检测模块。权限检测模块会根据登入用户的用户名查询该用户的角色。并比较用户请求的访问路径是否在该用户对应角色的可访问路径表中存在，若不存在则返回权限错误信息，若存在则继续将该请求转发给下一级 handler，也就是账户管理模块。账户管理模块收到该请求，会从数据库中查询账户表，获取所有账户的信息，并检测请求添加的用户名是否已经在系统中存在，如果该用户已经在系统中就告知用户名重复，否则就将该用户名存储到 rethinkDB 数据中。

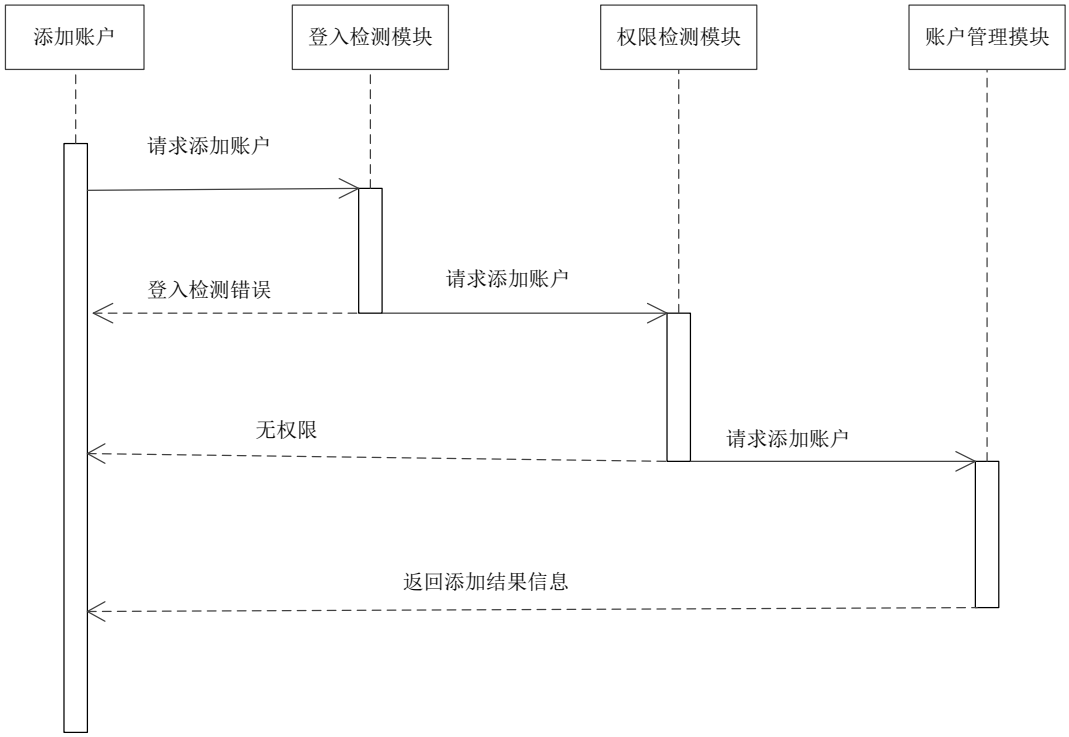

图 4.11 账户添加顺序图

删除账户会首先把消息发送到 PaaS 系统的 http 服务模块，服务模块直接将 http 请求交给路由器。路由器首先会将请求交给登录认证模块，检测该用户受否有 token，该 token 是否与数据库中的 token 一致。当确认该用户已经登入之后将该请求移交给权限检测模块。权限检测模块会根据登入用户的用户名查询该用户的角色。并比较用户请求的访问路径是否在该用户对应角色的可访问路径表中存在，若不存在则返回权限错误信息，若存在则继续将该请求转发给下一级 handler，也就是账户管理模块。账户管理模块收到该请求，会从数据库中查询账户表，判断是否用户存在，若存在则删除，返回删除结果。

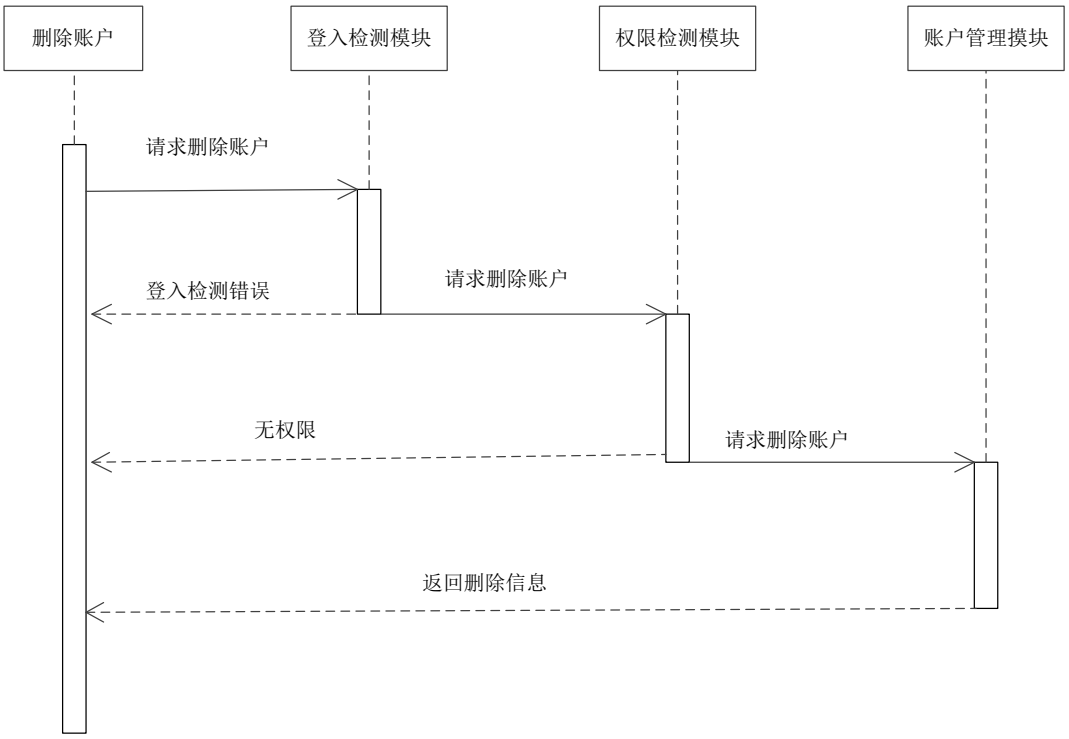

图 4.12 账户删除顺序图

4.3.5 服务器管理模块的设计

服务器管理模块主要功能就是对集群中的 Docker 主机进行管理控制，在本 PaaS 中对 Docker 主机的控制主要分为查看、添加 Docker 主机、删除 Docker 主机。服务器节点查看会直接检索数据库中的所有服务器节点，并生成服务器节点对象，再组成数组返回给用户。添加服务器节点会将 HTTP 包中的 json 数据解码为主机节点对象，然后检测对象中数据的合法性，并在数据库中增加该节点对象，最后刷新集群节点池，使新加入的 Docker 主机也提供服务。服务器节点删除是根据服务器节点的 id 来删除的，集群管理模块收到需要删除的 id 后会访问数据库将其从数据库中删除，同时会把该 Docker 主机从主机资源池里删除，使其停止提供服务。Docker 服务器节点的数据结构如下:

Engine struct 本地服务器类

ID string

SSLCertificate string

SSLKey string

CACertificate string

Engine *citadel.Engine

Health *Health

DockerVersion string

Engine struct 集群控制器中的服务器类

ID string

Addr string

Cpus float64

Memory float64

Labels []string

client *Dockerclient.DockerClient

clientAuth *Dockerclient.AuthConfig

eventHandler EventHandler

Engine 中 SSLCertificate 为 ssl 证书，SSLKey 为 ssl 公钥，CACertificate 为 CA 证书。这几个属性主要用于连接 https 的 Docker 主机节点，将会生成一个 X509 对象来连接 Docker 主机。Engine 中的子 Engine 为 Docker 主机节点的实例，里面包含 Docker 主机节点的 ID，http 地址（Addr），cpus，内存容量（Memory），服务器标签（Labels），客户端 handler（client），这里的 client 使用了 Docker 官方提供的 client API。客户端用户信息（clientAuth），使 Docker 服务器节点可以利用该账户登入到 Docker registry，可以不设。EventHandler 为 Docker 服务器发生事件所调用的 hander，会通知 PaaS 系统发生事件。Health 存储了该 Docker 主机节点的健康状态。在本 PaaS 系统中，该功能实际上是不需要的，因为服务器节点都是在局域网内，无需远端跨网络连接。

查看服务器节点:

查看服务器节点与账户查看过程一样，需要经过登入检测和权限检测。最终到达集群服务器管理模块，该模块会去访问数据库，获得最新的服务器节点的列表信息，并将它们生成服务器节点的对象列表，最后将该列表返回给用户。

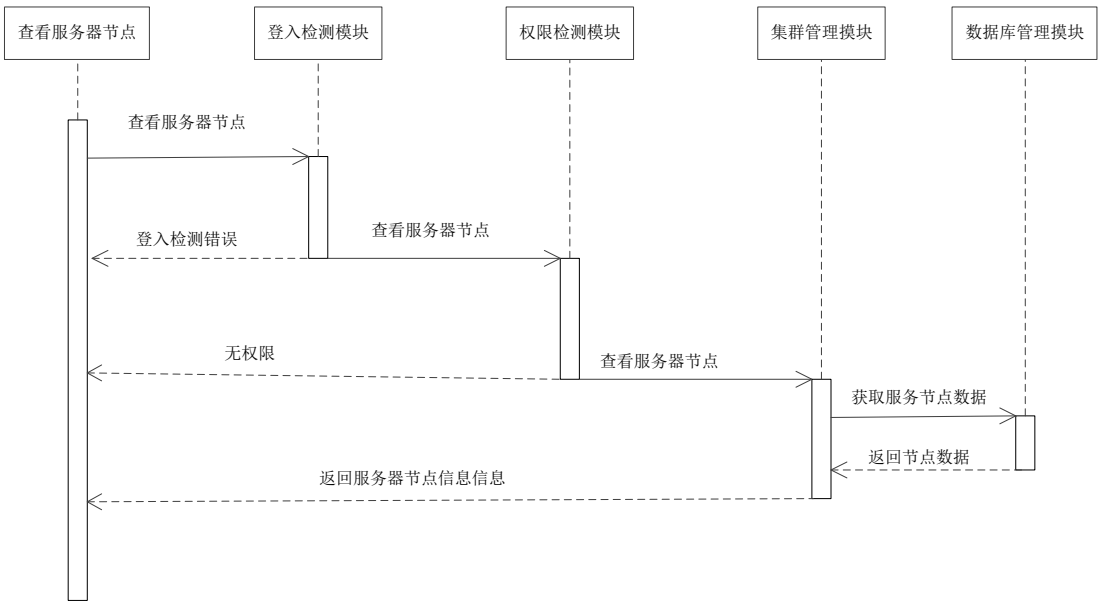

图 4.13 节点查看顺序图

添加服务器节点:

添加与查看过程一样，需要经过登入检测和权限检测。最终到达集群服务器管理模块，在这个模块里，会将 http body 里的服务器节点数据实例化为一个服务器节点对象，并将该节点直接存于数据库中，存数据库的时候可能会出现该节点已经存在，此时返回的消息会直接报错；如果不存在，则将 Docker 主机信息存于 RethinkDB 数据中，返回存储成功消息。

删除服务器节点:

删除与查看过程一样，需要经过登入检测和权限检测。最终到达集群服务器管理模块，该模块会去访问数据库，获得最新的服务器节点的列表信息。并将请求中的服务器节点 ID 与系统中存有的服务器列中 id 进行逐一比较，若有，则向数据库发送删除请求。若没有，则直接返回 id 错误信息。

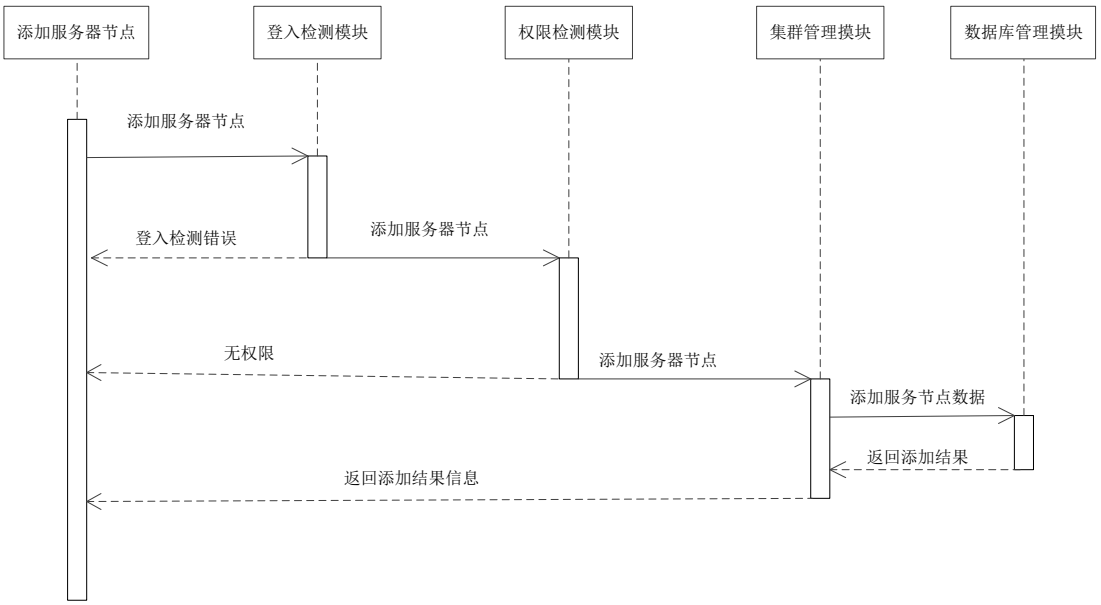

图 4.14 节点添加顺序图

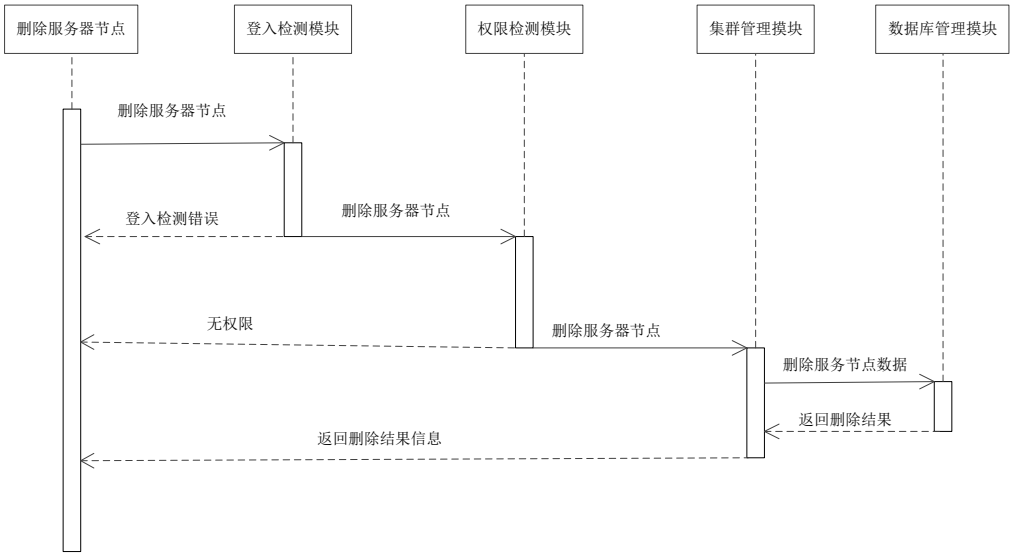

图 4.15 节点删除顺序图

4.3.6 容器控制模块的设计

容器控制模块是本系统的重要模块，该模块负责调度和管理服务器节点中的 Docker，提供应用的部署，以及应用的其他的控制。其提供的主要功能有查看容器，运行 n 个容器，重置容器某种容器数量，停止容器，启动容器，重启容器，销毁容器，查看容器日志。查看容器分为两种，一种是查看所有容器信息，这个功能会返回所有容器的信息；另外一种是查看单个容器信息，这个功能需要用户输入容器 id 号。运行 n 个容器这个功能会根据请求的应用名字和需要创建的个数在服务器集群中选择合适的服务器来创建请求数量的应用。重置容器数量这个功能会根据请求的应用和数量重新调整该容器应用的数量，如果数量不够，会创建缺少的数量；如果已有容器太多，会选择一些容器进行销毁操作，直到数量符合要求。停止容器会暂停一个容器，使其服务停止。启动容器会让一个停止的容器重新提供服务，重启容器重启某个容器。摧毁容器的操作分为两个步骤，第一是停止应用运行，第二步是删除容器数据。查看容器日志可以查看某个容器应用所产生的日志，日志可选标准输出和错误输出两种。

容器在系统中的数据结构如下:

Container struct 容器类

ID string // ID 是容器的 ID

Name string // Name 是容器的名字

Image *Image // Image 是容器被创建时所使用的镜像

Engine*Engine // Engine 是运行这个容器的服务器节点

State string// State 是容器的运行状态，运行或者停止

Ports []*Port // Ports 容器端口和主机端口的映射关系

Image struct 容器镜像类

Name string // Name 是 Docker 镜像名字

Cpus float64 // 在 1024 范围之内表示可用 cpu 的百分比

Cpuset string // Cpuset 用于设置容器运行与那几个 cpu 之上，单个或者多个

Memory float64// Memory 是为容器运行时提供的内存容量，单位是 MB

Entrypoint []string // Entrypoint 是对容器的运行时添加命令参数，不会被覆盖

Environment map[string]string// Envionrment 给容器设置启动后的环境变量

Hostname string// Hostname 给容器设置启动后的主机名

Domainname string// Domainname 设置容器的域名字

Args []string// Args 传给启动容器的命令行参数

Type string // Type 指定容器的类型，服务，应用等

Labels []string // Labels 是容器标签，与服务器节点的标签一致

BindPorts []*Port // BindPorts 确保容器对于特定的端口拥有访问权

UserData map[string][]string // UserData 是用户的数据传入容器内

Volumes []string // Volumes 是在服务器节点中的数据卷

Links map[string]string // Links 是可以内部连接到在本服务器节点的其他运行中的容器

RestartPolicy RestartPolicy // RestartPolicy 设置容器停止后的重启策略

Publish bool // Publish 设置是否暴露所有端口

查看容器:

查看容器与其他请求过程一样，需要经过登入检测和权限检测。然后请求到达容器管理模块，容器管理模块会将请求发送 Docker 客户端模块，该模块负责向远端 Docker 服务器请求容器数据。Docker 主机接收容器信息请求后返回容器数据列表。由于本地容器数据结构比 dokcer 客户端容器数据结构复杂，所以在收到数据之后还需要进一步封装为本地容器数据结构。需要在原有的容器数据结构中添加镜像结构和服务器节点结构。

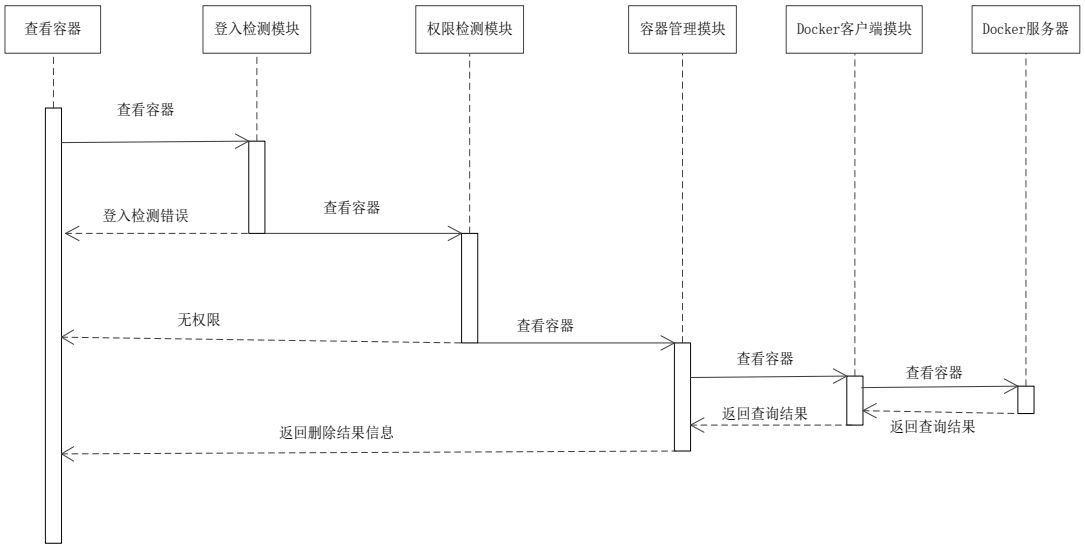

图 4.16 容器查看顺序图

运行 n 个容器:

运行容器与其他请求过程一样，需要经过登入检测和权限检测，然后请求到达容器管理模块。容器管理模块接受到请求之后首先会将请求中的 body 数据解码为镜像对象，然后开启多个协程，每个协程负责启动部署一个容器镜像。在协程内部，会根据镜像属性中的镜像应用类型来选择服务器过滤器（分几种，其中一种是根据镜像当中的 labels，只要两者有 label 标签一致，则该 Docker 主机可以允许该容器镜像。另一种是根据服务器节点中是否已经含有运行该）。拥有可运行服务器节点列表之后，再使用调度器对容器镜像的部署进行调度，向调度器输入可运行的容器列表和容器镜像对象；调度器会返回装载允许该容器镜像的服务器节点。然后容器管理模块会使用 Docker 客户端向该 Docker 节点发送运行命令，服务器节点收到该命令之后会添加 job，运行该容器镜像，返回运行结果。

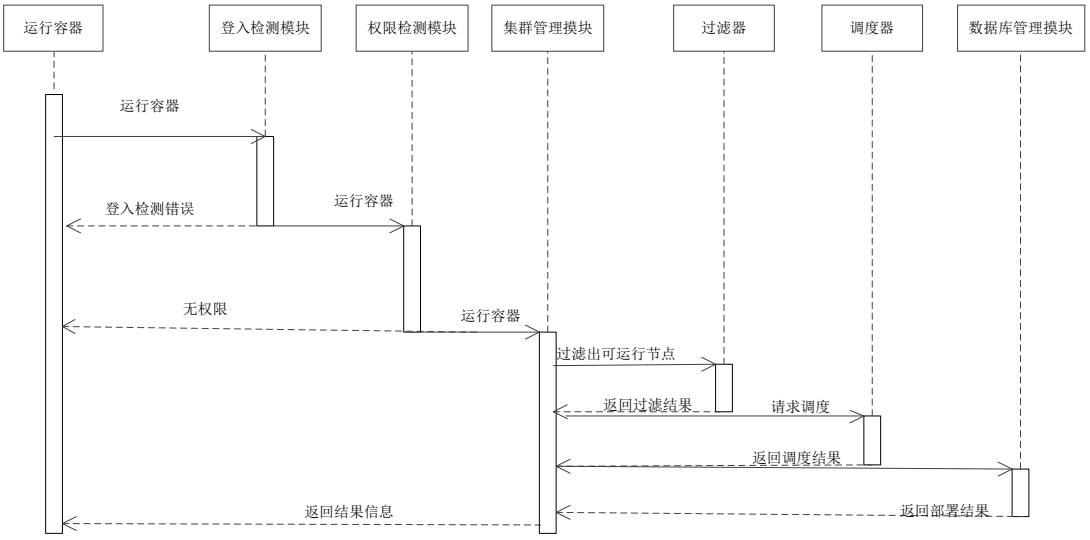

图 4.17 容器运行顺序图

重置容器某种容器数量:

重置容器数量与其他请求过程一样，需要经过登入检测和权限检测，然后请求到达容器管理模块。容器管理模块接受到请求之后首先会将请求中的 body 数据解码为容器对象。接着容器管理就会去搜集与之一样的容器数目，判别标准为容器的运行参数，内存分配容量，容器镜像类型，若这三种一样则视为与

需要重置的容器一样,统计请求容器在 Docker 主机集群中的同种容器运行个数,然后比较输入的需要容器个数和已存在的容器个数。如果已存在个数大于需要个数,则需要删除部分容器。删除的规则是从已存在容器列表的第一个开始删,直到删除需要删的数量为止。删除动作需要由删除功能来实现。如果已存在容器数小于需要容器数,则需要重新运行欠缺的容器数,该操作会调用运行容器命令。

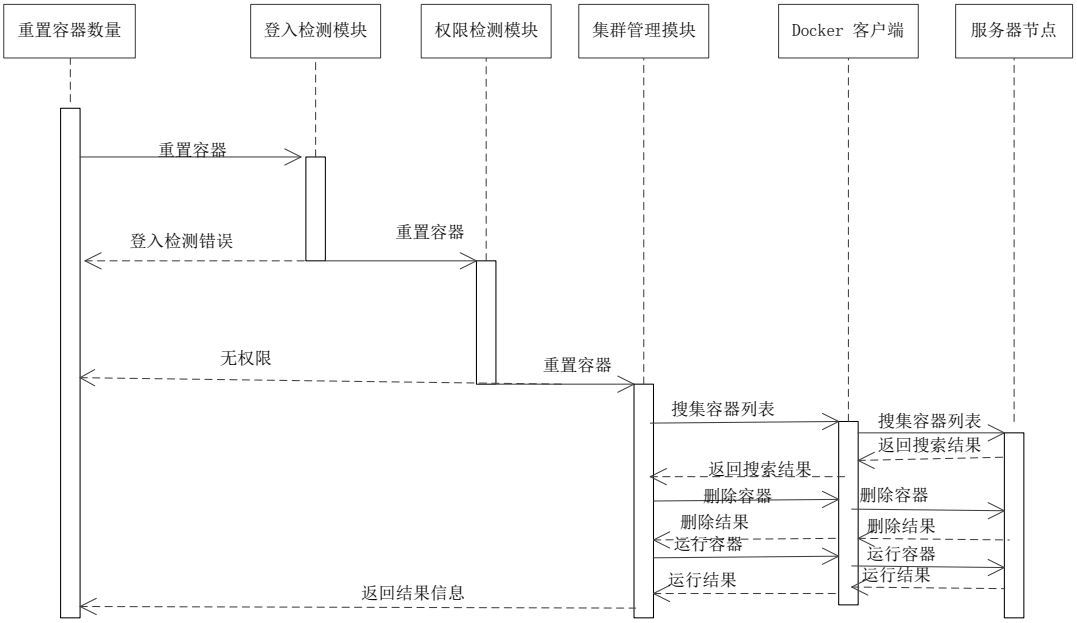

图 4.18 容器重置顺序图

停止容器:

停止容器与其他请求过程一样,需要经过登入检测和权限检测,然后请求到达容器管理模块。容器管理模块会利用容器对象中记录的服务器节点信息向运行该容器的服务器节点发送停止请求。服务器节点收到请求,执行停止容器工作,返回执行成功消息。

启动容器:

启动容器与其他请求过程一样,需要经过登入检测和权限检测,然后请求到达容器管理模块。容器管理模块会利用容器对象中记录的服务器节点信息向运行该容器的服务器节点发送启动请求。Docker 主机接收消息请求后,执行启

动容器过程，返回执行结果。

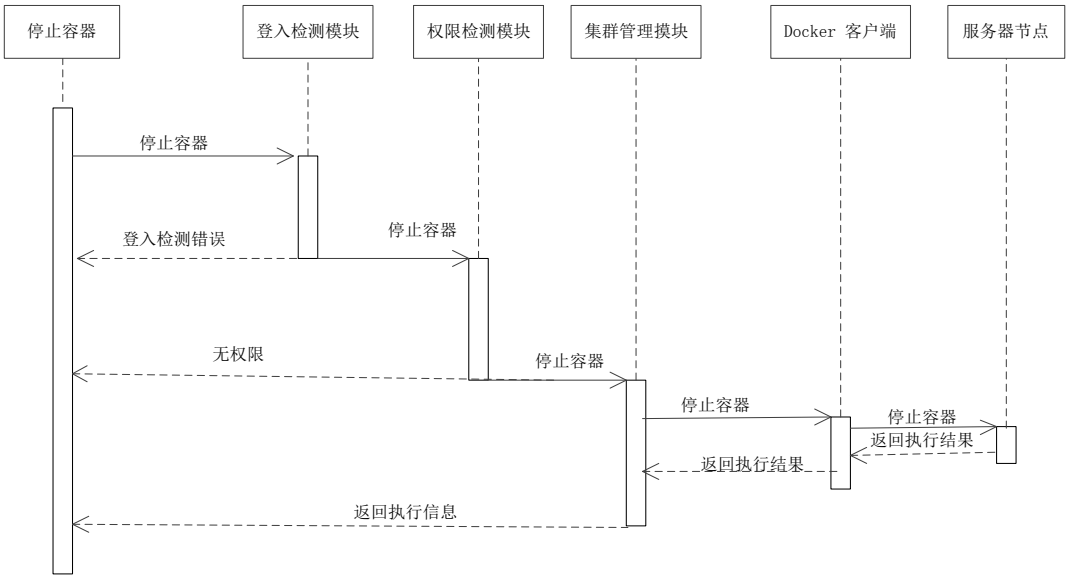

图 4.19 容器停止顺序图

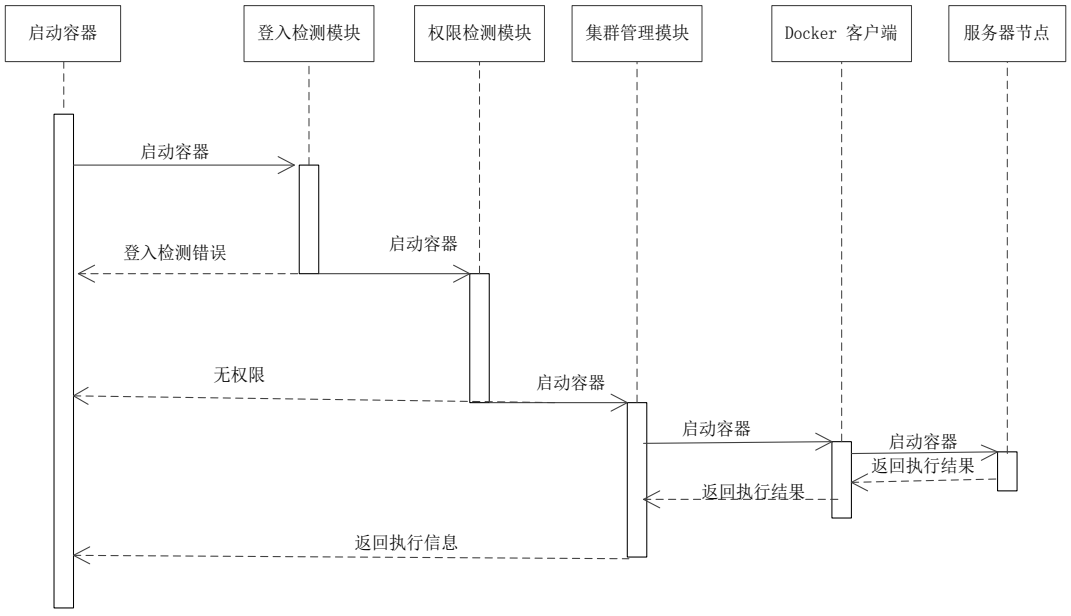

图 4.20 容器重新启动顺序图

重启容器:

重启容器与其他请求过程一样，需要经过登入检测和权限检测，然后请求到达容器管理模块。容器管理模块会利用容器对象中记录的服务器节点信息向运行该容器的服务器节点发送重启请求。服务器节点收到请求，执行重启容器工作，返回执行成功消息。

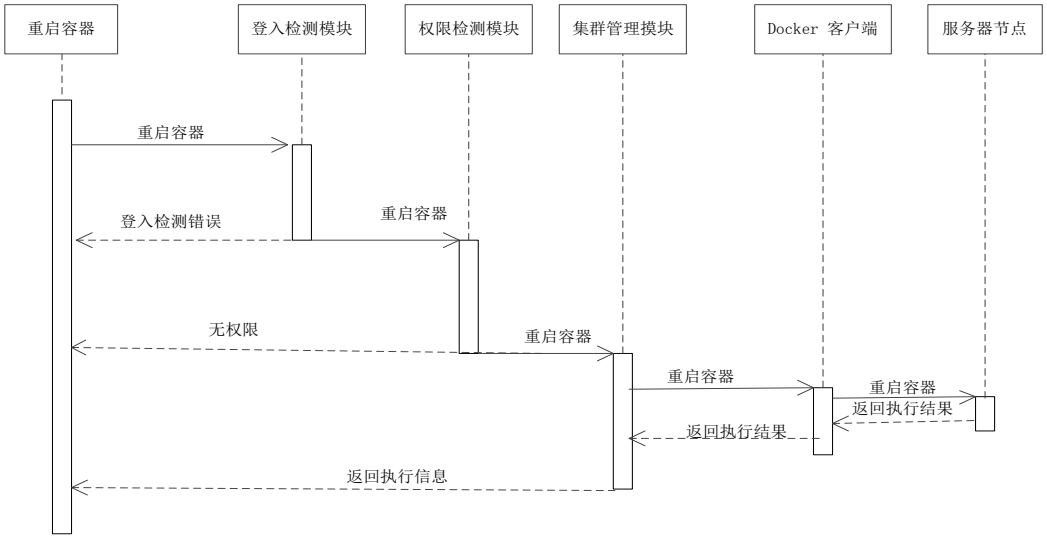

图 4.21 容器重新启动顺序图

销毁容器:

销毁容器与其他请求过程一样，需要经过登入检测和权限检测，然后请求到达容器管理模块。容器管理模块会利用容器对象中记录的服务器节点信息向运行该容器的服务器节点发送 kill 请求。Docker 主机接收消息请求，执行 kill 容器工作，返回执行结果；然后再向 Docker 主机发送 remove 请求；Docker 主机接收消息请求，执行 remove 容器工作，返回执行结果。

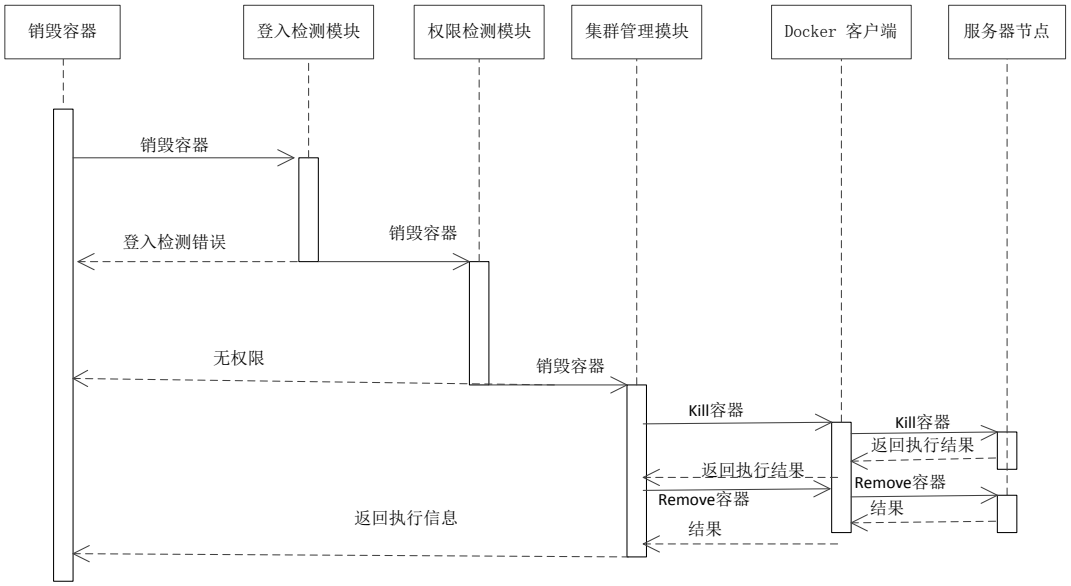

图 4.22 容器销毁顺序图

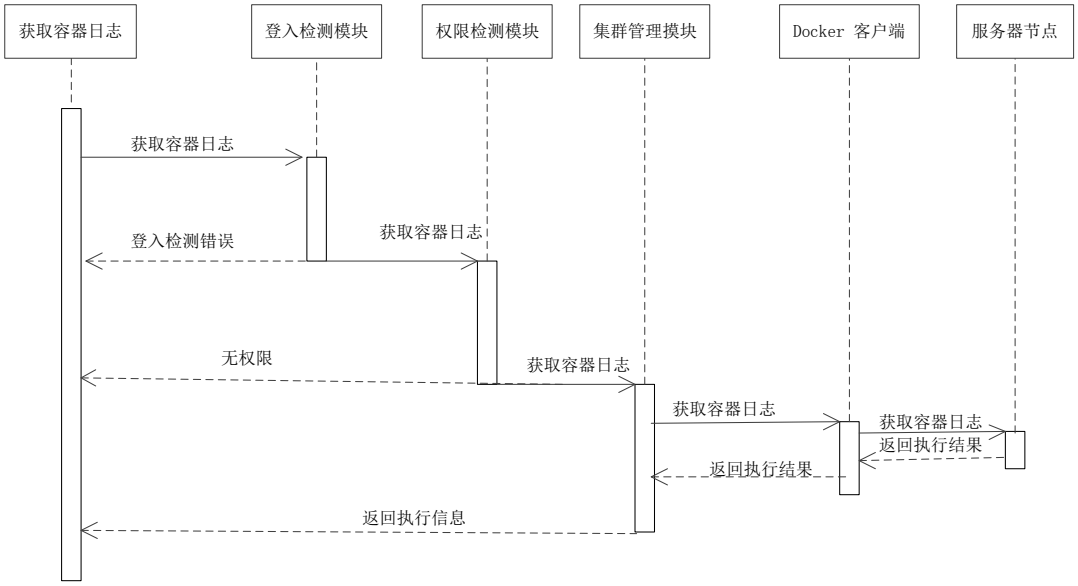

图 4.23 容器操作日志查看顺序图

查看容器日志:

销毁容器与其他请求过程一样，需要经过登入检测和权限检测，然后请求

到达容器管理模块。容器管理模块会利用容器对象中记录的服务器节点信息向运行该容器的服务器节点发送 log 请求。服务器节点收到请求，执行查找容器日志工作，返回执行成功消息。

4.3.7 事件管理模块的设计

事件管理模块主要负责各种事件的记录，比如容器的添加删除，服务器节点的添加删除，账户的添加删除等待，并将它们结构化后存入数据库。事件数据主要来源与两部分，一部分是从 Docker 服务器节点中监听 Docker 服务获得；另一部分通过监听 PaaS 的各类操作获得。两部分的数据结构如下:

来自服务器节点:

```
Event struct
Id        string
Status    string
From      string
Time      int64
```

转化为 PaaS 系统的事件类型为:

```
Event struct
Type          string
Container *citadel.Container
Engine        *citadel.Engine
Time          time.Time
Message       string
Tags          []string
```

在以上 PaaS 系统数据结构中 type 为该事件类型，比如添加和删除；container 是容器的结构体，engine 是服务器节点的结构体，time 是事件发生的时间，message 是事件说明字符串，tags 是事件的标签。

从服务器节点上传的事件数据可能如下:

```
{"status": "create", "id": "dfdf82bd3881","from": "ubuntu:latest", "time":1374067924}
```

其中 status 是事件的类型，id 是容器的 id，from 是容器使用的镜像的名字，time 是事件发生的名字。对于服务器节点事件的监听主要使用了回调函数的机

制和协程的机制。为每一个服务器节点开启一个协程，并在协程内设置回调函数 handle 供事件发生时调用。通过此方式就可以对所有 Docker 主机进行事件监听。

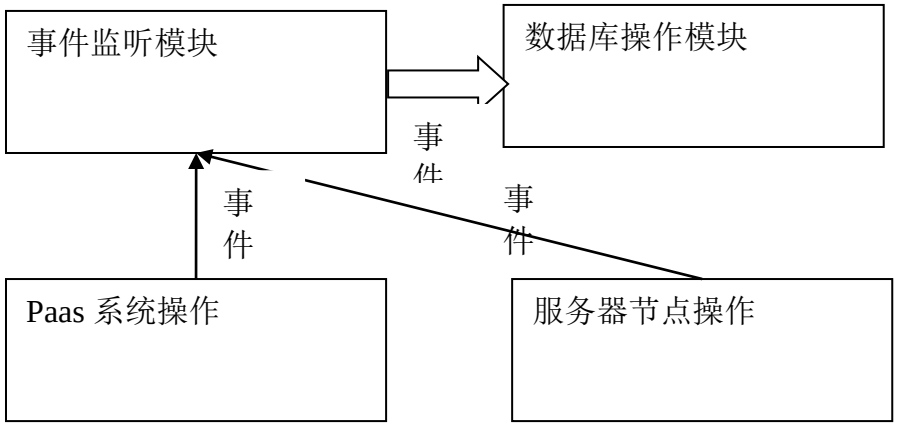

图 4.24 事件监听框架图

4.3.8 主控制器模块的设计

主控制器好比是所有模块的管理者，会将各个模块组合成一个完成的 PaaS 系统对外提供整体服务。控制器的数据结构如下:

```
Controller struct
cController     *Container
cCluster        *Cluster
cEngine         *EngineContro
cAccount        *AccountOpt
cAuth           *Authenticator
cEvent          *EventOpt
dbname           string
sesssion        *r.Session
engines          []*Engine
```

在主控制器中，有容器控制模块对象，集群控制模块对象，服务器节点控制模块对象，账户控制模块对象，权限认真模块对象，事件操作模块对象，数据库名字，数据库 session 对象，服务器节点列表。

控制器创建:

创建控制器需要给控制器提供 rethinkDB 的 ip、端口号和库名称。在新建函数中会如下操作 1.使用数据库模块连接数据库，连接成功后得到一个 session 对象实体，供以后对数据库进行操作。2.生成控制器对象，给新的对象分配内存空间，并对内部的成员变量初始化。3.调用 DB 初始化函数将 rethinkDB 中的各种表建立起来。4.调用控制器对象的初始化函数来初始化调度器、Docker 主机连接器、设置监听器等操作。

数据库初始化:

数据的初始化很简单，建立四张表，一张账户表，一张 Docker 服务器表，一张事件表，一张角色表。由于使用的是 rethinkDB，因此不需要设定表中的数据字段和类型，因为它是一个 document 类型的数据库。建表的过程是这样的:先检测表是否存在，若存在则继续下一张表，若不存在则建立该表。

控制器的初始化: 1.调用集群控制器对象获得集群中所有 Docker 主机节点的数据信息。2.遍历主机节点数组，如果主机对象中的证书不为空则创建 TLS 文件（由 CACertificate、SSLCertificate、SSLKey 生成），然后连接服务器节点获得一个包含 httpClient 的 client 对象，将这个对象填充到服务器节点对象中，用于以后操作 Docker 主机。3.初始化集群控制器，对其添加资源调度器和主机信息过滤器。4.初始化每个 Docker 主机的事件监控 handler。5.初始化容器控制器，将集群控制器注册到容器管理器，使容器控制器可以调用集群控制器的调度服务。

4.3.9 客户端的设计

客户端设计为基于命令行，方便使用脚本进行自动化控制。设计思路是先设计一个 proxy，该 proxy 屏蔽了所有与 PaaS 系统的 http 操作，暴露出的只是各种功能函数接口，包括账户类的，服务器节点类的，容器类的等等。然后针对该 proxy 编写各种命令行交互的功能，使命令行数据转化为结构化对象，再利用 proxy 与 PaaS 系统进行通信。

Proxy 所含有的功能包括: 容器查看，创建容器，销毁容器，启动容器，重置特定容器数量，查看容器日志，服务器节点查看，服务器节点添加，服务器节点删除，集群信息查看，事件查看，账户查看，账户添加，账户删除，登入。

工作流程为: 1.登入，获得 PaaS 分配的 token，将 token 存入本地。2.使用

其他功能，使用功能时会导入 token，放入 http 头部，与 PaaS 系统进行通信。

4.4 本章小结

本章首先对 PaaS 系统从总体上进行系统架构，然后将系统又细分成多个功能模块进行详细设计。先将系统架构分为 PaaS 系统客户端和服务器端，然后对服务器端上各个功能模块详细设计说明。分别对八个主要模块从实现流程的设计，到数据结构的定义及其在数据库中的存储字段名称都做了详细的说明。每个模块都用时序图进行了阐述和详细说明。

第5章 基于 Docker 的企业 PaaS 系统模块的实现

5.1 实现环境

该 PaaS 系统是在 liteide X25 开发平台上上开发实现的，对应 Go 版本为 Go1.4.1 版本。开发环境的操作系统为 Windows 7 64 位。

5.2 核心模块的实现

5.2.1 主控制器的实现

Controller 是 PaaS 系统主要的控制器，它的实现主要使用了代理模式。具体的就是利用控制器为桥梁，将 http 服务于后端各个管理模块连接在一起。Http 服务模块的所有服务都通过主控制器来访问，在主控制器中，代理后端管理模块的同时，会在代理使用后端之前进行一些预处理，比如在添加容器时，调用事件管理模块，向数据库中插入添加容器事件。

主控制器的类结构如下:

```
Controller
cController    *Container
cCluster       *Cluster
cEngine        *EngineContro
cAccount       *AccountOpt
cAuth          *Authenticator
cEvent         *EventOpt
dbname         string
sesssion       *r.Session
engines        []*Engine
func (m *Controller) initDb() error
func (m *Controller) init()
func (m *Controller) Authenticate(username string, password string)
func (m *Controller) NewAuthToken(username, urlagent string)
func (m *Controller) InspectAccount() []*Account
func (m *Controller) AddAcount(acc *Account) error
func (m *Controller) DeleteAccount(ID string) error
```

func (m *Controller) Listcontainer(all bool) []*citadel.Container

func (m *Controller) InspectContainer(id string) (*citadel.Container, error)

func (m *Controller) RunContainer(img *citadel.Image, count int, pull bool)

func (m *Controller) ScaleContainer(container *citadel.Container, count int)

func (m *Controller) RestartContainer(container *citadel.Container) error

func (m *Controller) StartContainer(container *citadel.Container) error

func (m *Controller) StopContainer(container *citadel.Container) error

func (m *Controller) DestroyContainer(container *citadel.Container) error

func (m *Controller) ContainerLogs(container *citadel.Container)

func (m *Controller) AddEngine(e *Engine) error

func (m *Controller) ListEngines() []*Engine

func (m *Controller) DeleteEngine(id string) error

func (m *Controller) InspectCluster() (*ClusterInfo, error)

func (m *Controller) GetEvents(count int) ([]*Event, error)

主控制器的新建需要使用全局函数 NewController 来实现，该函数是一个全局函数，定义如下:

func NewController(dbAddr string, dbName string) (*Controller, error)

向 NewController 传入数据库地址及其端口和数据库名字，就可以得到一个 Controller 实体，返回的形式是指针的形式。如果有新建控制器出错，则返回控制器的指针为空，错误在 error 中说明。Error 是一个 struct，它内部封装了错误的描述说明。

成员变量介绍如下:

cController: 是 Docker 容器的控制器对象，用于调用 Docker 主机各项服务。

cCluster: 是 Docker 主机集群管理工具的对象，提供对可用 Docker 节点的过滤和调度服务。

cEngine: 是 Docker 主机节点管理对象，用于对 Docker 主机的管理操作。

cAccount: 是账户管理对象，用于对各类级别的管理员账户进行控制。

cAuth: 是登入验证模块，用于对请求的 token 检测。

cEvent : 是事件管理对象，用于对 Docker 主机和 PaaS 控制系统产生的历史操作数据进行归一化，然后统一存入数据库。

Dbname: 用于保存当前操作数据库名称。

Sesssion: 是数据库会话对象，通过它可以执行各类 sql 操作。

Engines: 已有服务器节点列表。

成员方法介绍如下:

initDb(): 初始化数据库，建立不存在的表。

init(): 完成对主控制器的初始化配置，主要为让每个服务器节点对象都连接到 Docker 服务器，初始化注册资源调度器和过滤器。

Authenticate(username string, password string): 该方法直接代理了 auth 模块的 authenticate 函数，实现了对用户名密码的验证。

NewAuthToken(username, urlagent string): 该方法直接代理了 auth 模块的 newauthtoken 方法，实现了 token 的生成。

InspectAccount(): 该方法代理了账户管理模块的 InspectAccount 函数，查看所有账户信息。

AddAcount(acc *Account): 该方法代理了账户管理模块的 addacount 函数，但是在代理函数 AddAcount 中添加了事件的添加，并用事件管理对象进行存储。创建事件对象关键代码如下:

```
evt := &Event{
        Type:       "add-account",
        Time:       time.Now(),
        Message: fmt.Sprintf("name=%s", acc.Username),
        Tags:       []string{"cluster", "security"},}
```

DeleteAccount(ID string): 该方法代理了账户管理模块的 DeleteAccount 函数，在代理函数中添加了"delete-account"类型的事件并存于数据库中。

Listcontainer(all bool): 该方法直接代理了账户管理模块的 Listcontainer 函数，将在账户管理模块进行说明。

InspectContainer(id string): 该方法直接代理了账户管理模块的 InspectContainer 函数，将在账户管理模块进行说明。

RunContainer(img *citadel.Image, count int, pull bool) : 该方法直接代理了容器管理模块的 RunContainer 函数，会启动容器部署应用，详细会在容器管理模块说明。在代理函数中添加了" run-container"类型的事件并存于数据库中。

ScaleContainer(container *citadel.Container, count int): 该方法直接代理了容器管理模块的 ScaleContainer 函数，会重置容器应用数量。在代理函数中添加了

" scale-container"类型的事件并存于数据库中。

RestartContainer(container *citadel.Container): 该方法直接代理了容器管理模块的 RestartContainer 函数，会重启一个正在运行的容器，如果容器原来就停止状态，该函数还是可以将容器启动。在代理函数中添加了" restart -container"类型的事件并存于数据库中。

StartContainer(container *citadel.Container): 该方法直接代理了容器管理模块的 StartContainer 函数，会启动已经停止的容器应用。在代理函数中添加了" start -container"类型的事件并存于数据库中。

StopContainer(container *citadel.Container): 该方法直接代理了容器管理模块的 StopContainer 函数，会暂停容器应用的运行。在代理函数中添加了" stop -container"类型的事件并存于数据库中。

DestroyContainer(container *citadel.Container): 该方法直接代理了容器管理模块的 DestroyContainer 函数，会销毁该容器实体。在代理函数中添加了" destroy -container"类型的事件并存于数据库中。

ContainerLogs(container *citadel.Container): 该方法直接代理了容器管理模块的 ContainerLogs 函数，会查询一个容器的所有日志。

AddEngine(e *Engine): 这个函数代理了服务器节点管理模块的 AddEngine 函数，它实现了向 PaaS 系统中添加服务器节点功能。在代理方法中添加了" add-engine"类型的事件并存于数据库中。在代理函数最后还重置了主控制器，使主控制器能在初始化函数中对未管理的服务器节点进行添加管理。

ListEngines(): 这个会直接返回主控制器对象中的 Docker 主机节点列表。

DeleteEngine(id string): 这个函数代理了服务器节点管理模块的 DeleteEngine 函数，它实现了向 PaaS 系统中删除一个 Docker 服务器节点。在代理方法中添加了" delete-engine"类型的事件并存于数据库中。最后通知更新主控制器中的 Docker 主机列表。

InspectCluster(): 该函数是查看集群当前的状态，包括 cpu 总量及其使用量，内存总量及其使用量，容器个数，Docker 服务器个数，容器镜像个数。关键代码如下:

```
for _, e := range c.engines {
        c, err := e.ListContainers(false)
        for _, cnt := range c {
```

```
            reservedCpus += cnt.Image.Cpus
            reservedMemory += cnt.Image.Memory}
        i, err := e.ListImages()
        containerCount += len(c)
        imageCount += len(i)
        totalCpu += e.Cpus
        totalMemory += e.Memory}
```

首先循环遍历控制器中的所有服务器节点，在循环体中利用 listcontainer（false）函数获取该服务器节点中所有正在运行的容器，如果发生错误，则继续下一个循环；否则所有正在运行的容器列表，提取每一个容器的 CPU 容量和内存容量。结束循环后利用 listimages 获取该节点上的容器镜像个数。最后利用 e.cpus 和 e.memory 获取服务器节点的所有 cpu 资源和内存资源容量。

GetEvents(count int)该函数代理了事件模块对象的 GetEvents 函数，可以返回指定个数的事件。

PaaS 主控制模块的初始化流程如下:

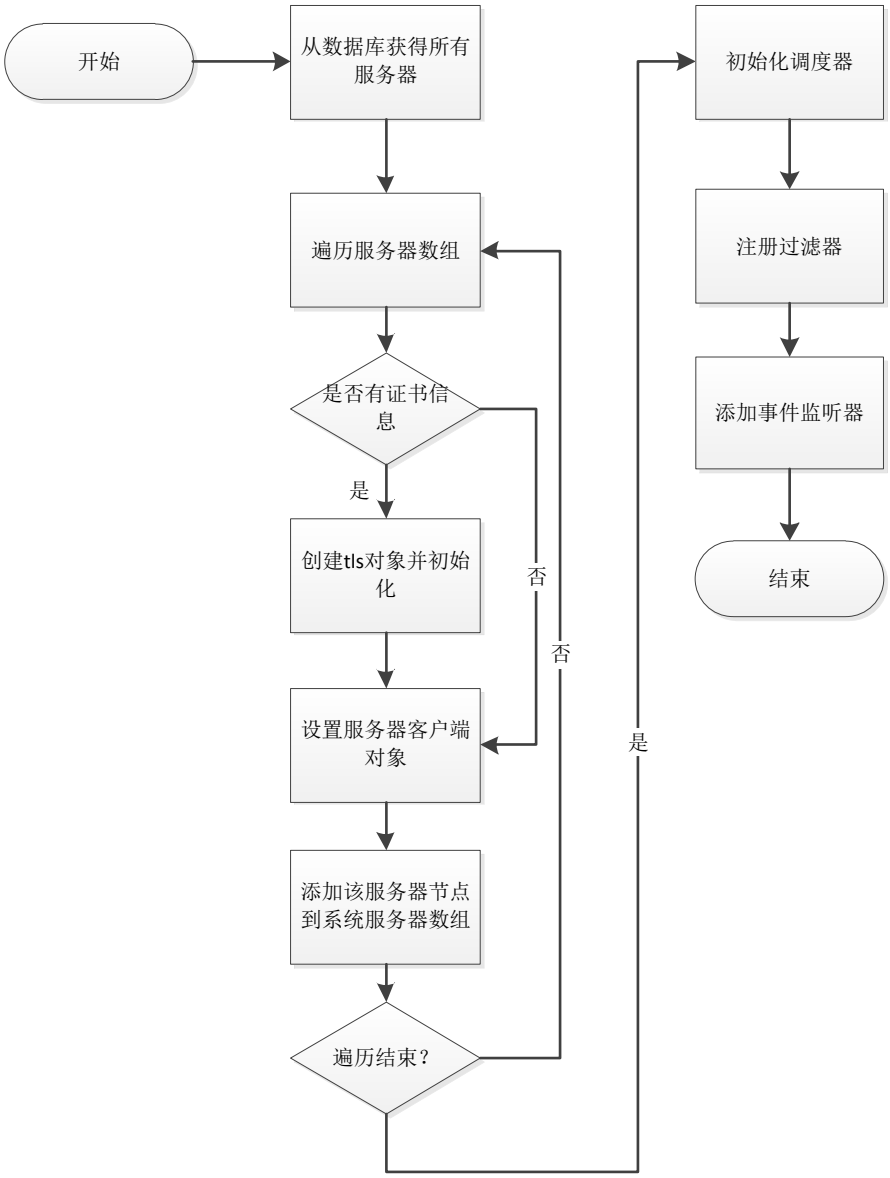

图 5.1 本 PaaS 系统主控模块初始化

5.2.2 系统 http 服务模块的实现

http 服务模块监听获取用户的消息请求，消息的格式被规定为 json 格式。处理过程为先会调用认证模块来认证用户是否登入，会调用权限认证模块确认用户对 remote api 的访问权限。然后根据 remote api 的路径将请求路由到相应的函数中，在对应的函数中会对请求进行解析，比如地址参数解析，http header 解析，http body 解析等，然后将解析得到的数据传给主控制器进行处理。

http 服务的初始过程为: 新建一个登入服务路由器，用于转发登入相关请求。

新建一个账户路由器，通过它可以让密码修改的消息请求得以转发到相应的函数。创建一个 API 路由器，用于转发普通的消息请求。在前面添加一个权限认证路由器，在权限认证路由器中已经调用了登入认证服务，因此没有添加登入认证路由器。新增总路由器，将/auth，/account，/api 这三个请求路径转发到之前三个路由，将端口监听服务打开，将所有请求直接转给总路由器处理，该过程的关键代码如下:

```
defaultRouter := http.NewServeMux()
defaultRouter.Handle("/auth/", userRouter)
defaultRouter.Handle("/account/", authRouter)
defaultRouter.Handle("/api/", remoteApiAuthRouter)
```

开启监听服务的关键代码如下:

```
http.ListenAndServe(":5002", defaultRouter)
```

给路由添加一层权限认证的关键代码如下:

```
remoteApiAuthRouter := negroni.New()
remoteUserAuth := Auth.NewAuthRequired()
remoteApiAuthRouter.Use(negroni.HandlerFunc(remoteUserAuth.HandlerFuncWithNext)) //赋予它 ServeHTTP 函数
remoteApiAuthRouter.UseHandler(remoteApiRouter)
```

添加请求方法转发的关键代码如下:

```
remoteApiRouter.HandleFunc("/api/account", inspectAccount).Methods("GET")
```

http 服务模块提供的 remote api 如表 5.1，在括号中的 id 为容器 id，将会在调用函数内被解析出来。

表 5.1 路由表

请求方法	路径	调用方法
GET	/api/account	inspectAccount
POST	/api/account	addAccount
DELETE	/api/account	deleteAccount
POST	/api/engines	addEngine
GET	/api/engines	listEngine
GET	/api/engines/{id}	inspectEngine
DELETE	/api/engines/{id}	deleteEngine
GET	/api/cluster/info	inspectCluster
GET	/api/events	events
GET	/api/containers	listContainer
POST	/api/containers	runContainer
GET	/api/containers/{id}	inspectContainer
DELETE	/api/containers/{id}	destroyContainer
GET	/api/containers/{id}/start	startContainer
GET	/api/containers/{id}/stop	stopContainer
GET	/api/containers/{id}/restart	restartContainer
GET	/api/containers/{id}/scale	scaleContainer
GET	/api/containers/{id}/logs	containerLogs
POST	/auth/login	loginPaaS
POST	/account/passwdchange	passwdchange

接下来介绍各个各个方法的执行过程:

inspectAccount: GET 请求会返回调用主控制器的 InspectAccount 函数，得到账户信息对象之后，设置返回文本类型为 application/json，并将账户信息对象编码为 json 消息返回请求。关键代码如下:

```
accounts := m.InspectAccount()
w.Header().Set("content-type", "application/json")
json.NewEncoder(w).Encode(accounts)
```

addAccount: POST 请求会收到 json 格式，该函数会将其解码为账户对象，

调用主控制器的 addcount 将该对象添加到系统中，最后返回状态码为 204 的消息。

deleteAccount: DELETE 请求会收到 json 格式，该函数会将其解码为账户对象，调用主控制器的 deleteAccount 将该对象添加到系统中。最后返回状态码为 204 的消息。

addEngine: POST 消息请求会收到 Docker 主机节点对象对应的 json 格式信息，这个函数会将它解码为 Docker 主机节点对象，并新建对应的 health 对象，对象内的 status 设置为 pending，responsetime 设置为 0，然后将该 health 对象设置到 Docker 主机节点对象，利用主控制器的 addengine 函数把该 Docker 主机对象加到 PaaS 系统服务中，最后返回状态码为 201 的消息。

listEngine: GET 请求，该函数会调用主控制器的 listEngine 函数，得到服务器节点列表，并将该列表编码为 json 格式返回请求。

inspectEngine: GET 请求，该函数会首先解析出路由路径中的 engine id，然后在 engine 列表中找对应的服务器节点，找到后将其返回。解析 engine id 的关键代码如下:

```
vars := mux.Vars(r)
id := vars["id"]
```

先用 vars 函数解析一次路径,然后从解析好的 map 中获取关键字为 id 的值。

deleteEngine: DELTE 请求，该函数会首先解析出路由路径中的 engine id，然后调用主控制器的 deleteEngine 函数将该服务器节点删除。

inspectCluster: GET 请求，该函数会调用主控制器的 inspectCluster 函数获取集群的状态信息，设置 http 头部格式为 application/json，并将集群信息对象将其转为 json 格式写到 http body 中，返回该请求。

Events: GET 请求，该函数需要从 url 中解析传入的数量参数，然后调用主控制器的 Getevents 函数获取相应数量的事件。获取参数的关键代码如下:

```
count := r.FormValue("count")
n, err := strconv.Atoi(count)
```

listContainer: GET 请求，该函数调用主控制器的 listcontainer 函数，获得所有容器的列表，设置 http 头部格式为 application/json，并将所有容器的列表转为 json 格式写到 http body 中，返回该请求。

runContainer: POST 请求，它需要从 url 中解析两个参数，首先是布尔型的

pull，用于指示是否需要 pull 操作；另一个是 count，用于控制 container 运行的个数。然后将 http body 中的容器镜像消息解码为 Docker 镜像对象，调用主控制器的 runcontainer 函数运行指定个数的 Docker 容器。

inspectContainer：GET 请求，该函数会从 path 中解析出 container id，根据这个 id 调用主控制器的 inspectcontainer 函数获取该容器 id 的信息，编码为 json 文本返回。

destroyContainer：DELETE 请求，该函数会从 path 中解析出 container id，根据这个 id 调用主控制器的 destroyContainer 函数获取删除该容器，返回 204 消息。

startContainer：GET 请求，该函数会从 path 中解析出 container id，首先调用主控制器的 inspectcontainer 函数根据这个 id 获得容器的 contianer 对象，然后再调用主控制器的 startcontainer 函数启动容器，返回 204 消息。

stopContainer：GET 请求，该函数会从 path 中解析出 container id，调用主控制器的 stopContainer 函数停止该容器的运行服务，返回 204 消息。

restartContainer：GET 请求，该函数会从 path 中解析出 container id，调用主控制器的 restartContainer 函数停止该容器的运行服务，返回 204 消息。

scaleContainer：GET 请求，该函数会从 path 中解析出 container id，从 url 中解析出 count 参数，首先调用主控制器的 inspectcontainer 函数根据这个 id 获得容器的 contianer 对象，然后再调用主控制器的 scaleContainer 函数重置容器数量，返回 204 消息。

containerLogs：GET 请求，该函数会从 path 中解析出 container id，首先调用主控制器的 inspectcontainer 函数根据这个 id 获得容器的 contianer 对象，然后再调用主控制器的 containerLogs 函数获得日志数据，由于从 dockclient 模块中返回的数据对象为 ReadCloser 接口，该接口定义了 read 函数，因此使用 copy 函数将 readCloser 中的数据拷贝到 ResponseWriter 中。

loginPaaS：POST 请求，该函数会解码用户信息，生成用户对象，检验账户明和密码，确认密码后新建一个 token 返回。

Passwdchange：POST 请求，该函数会从解析出用户名和密码，先更加用户名找到该用户，然后将该密码转换为 sha1 编码后更新数据库中的用户数据。返回 204 消息。

5.2.3 登入和认证模块的实现

登入模块的登入检测类的 HandlerFuncWithNext 函数为对外函数，该函数在传入路由器的时候被强制转换为 HandlerFunc 接口，该接口默认实现了 serveHttp 函数，但是该函数内部的实现就是调用传入的函数。也就是说传入的对象类型是函数，该函数有个方法叫做 serveHttp，在 servehttp 的内部实现中调用了调用它的函数对象，从而实现了基于接口的函数回调。

检测类的类如下:

AuthRequired

deniedHostHandler http.Handler

func (a *AuthRequired) handleRequest(w http.ResponseWriter, r *http.Request)

func (a *AuthRequired) HandlerFuncWithNext(w http.ResponseWriter, r *http.Request, next http.HandlerFunc)

该类的新建需要使用一个全局的新建函数:

func NewAuthRequired() *AuthRequired

该函数会新建一个 authrequired 对象，然后初始化一个实现了 http.handler 接口的对象，在这里将一个带有传入参数 http.ResponseWriter 和 *http.Request 类型的函数强制转换为了 http.HandlerFunc 对象，因为该对象已经实现了 servehttp 方法。所以在使用的时候可以用 deniedHostHandler.serveHttp(w,r)的方式调用原传入的函数，并不需要知道原函数的名字。

HandlerFuncWithNext 提供对外的服务，在调用时会传入一个 http.handlerfunc 的函数对象。因此在该函数内部调用了两个函数，一个是本地的 handleRequest 函数，提供本地的功能，另外一个是传入的函数，实现另外一种功能。

handleRequest: 是本地的主函数，该函数实现了从 request 请求的头部获取 key 值为 X-Access-Token 的值，这个值中包含了用户名和 token，格式为“用户名:token”。然后通过用户名从数据库中找到对应的用户实体对象，将从数据库中得到的 token 和请求中的 token 进行比较，如果一致，则该用户合法，继续将其请求往下一级传输。若这一级检测到 token 不一致，则直接通过 responseWriter 写会请求报错，代码如下:

http.Error(w, "unauthorized", http.StatusUnauthorized)

访问权限认证结构和登入认证结构一致。类视图如下:

AccessRequired

deniedHostHandler http.Handler

acl map[string][]string

func (a *AuthRequired) handleRequest(w http.ResponseWriter, r *http.Request)

func (a *AuthRequired) HandlerFuncWithNext(w http.ResponseWriter, r *http.Request, next http.HandlerFunc)

该类的新建需要使用一个全局的新建函数:

func NewAccessRequired(m *manager.Manager) *AccessRequired

该函数会新建一个 AccessRequired 对象,然后初始化一个实现了 http.handler 接口的对象，在这里将一个带有传入参数 http.ResponseWriter 和 *http.Request 类型的函数强制转换为了 http.HandlerFunc 对象，它的作用是向 ResponseWriter 中写出错内容通知给用户。因为该对象已经实现了 servehttp 方法，Serverhttp 内部实现就是调用对象本身,所以在使用的时候可以用 deniedHostHandler.serveHttp（w,r）的方式调用原传入的函数，并不需要知道原函数的名字。新建的时候也会初始化另外一个对象 acl，它就是一个 map，对应着角色到权限的映射，默认如下:

```
acl["admin"] = []string{"*"}
acl["user"] = []string{
    "/api/containers",
    "/api/cluster/info",
    "/api/events",
    "/api/engines",}
```

HandlerFuncWithNext 提供对外的服务,在调用时会传入一个 http.handlerfunc 的函数对象。因此在该函数内部调用了两个函数，一个是本地的 handleRequest 函数，提供本地的功能，另外一个是传入的函数，实现另外一种功能。

handleRequest: 是本地的主函数，该函数实现了从 request 请求的头部获取 key 值为 X-Access-Token 的值,这个值中包含了用户名和 token,格式为“用户名:token”。然后使用登入认证模块进行检验其 token 是否合法，若用户合法，继续将其请求往下。若这一级检测到 token 不一致，则直接通过 responseWriter 写回请求报错。接下去的操作是在 PaaS 系统的管理员列表中查找到他，并查看他的角色名称，然后在角色 map 中查找该管理员的权限列表，从 map 中通过键值找到该角色的所有权限路径后，在路径中进行前缀匹配，若匹配成功则继续下一

级传输，若匹配失败，则 request 请求到此处为止。

5.2.4 注册和心跳检测模块的实现

心跳检测模块有三个部分组成，Docker 服务器端，心跳模块，PaaS 系统主控制端。这第三个部分的主要功能是让 PaaS 系统中的服务器节点列表始终是活跃的，如果有发生宕机事件，心跳检测模块会第一时间发现，从 etcd 中的 Docker 主机列表中删除它。若有新节点添加，则 Docker 服务器端程序会想 etcd 中注册自身。PaaS 系统控制器端会发现 etcd 中的 Docker 主机节点的增减，同时同步到自身维护的 Docker 主机节点列表。

心跳检测模块的执行过程需要以下几个步骤:

1.启动运行 etcd

单机启动 etcd 的命令为 etcd --listen-client-urls http://ip: 端口。这样，通过 http 请求就可以对 etcd 中的键值进行操作了。

2.PaaS 系统存有的 Docker 主机节点装入 etcd 中

PaaS 系统的数据库中存在先前添加的 Docker 主机节点列表。因此，当 PaaS 系统启动时需要向 etcd 中注册自己数据库中已有的 Docker 主机节点列表。注册的方式为向 engines 目录中添加以 ID 为 key，地址为 value 的键值对。关键代码如下:

```
for _, engine := range engines {
    key := engine.ID
    key = "engines/" + key
    value := engine.Engine.Addr
    value = value[7:len(value)]
    resp, err := client.Set(key, value, 0)}
```

遍历系统中的服务器节点列表，将每个节点列表中的 ID 存到 key 中，将每个服务器节点地址存到 value 中，通过发送一个 put 请求来实现对 key-value 的注册，这里的 set 就是封装了 put 请求。

3.服务器节点注册

当服务器启动后，希望自身自动添加到 PaaS 系统中提供服务 Docker 服务，就需要用到服务注册程序。服务注册程序实现的是将主机的 cpu 数量，内存容量和 ip 地址自动检测到，并将其以一定数据结构注册到 etcd 中，待 PaaS 系统发现

它并将其添加到 PaaS 系统中。

检测 cpu 数量主要是通过读取 linux 系统中的/proc/cpuinfo 文件，并通过解析得到 cpu 数量，关键代码如下:

```
reg, _ := regexp.Compile("processor")
data, err := file2string(filepath)
num := reg.FindAllString(data, 16)
corenum := len(num)
```

检测内存容量主要是通过读取 linux 系统中的/proc/meminfo 文件，并通过解析得到内存容量，关键代码如下:

```
reg, _ := regexp.Compile("MemTotal:[ ]+[0-9]+ kB")
data, err := file2string(filepath)
num := reg.FindAllString(data, 16)
if len(num) != 0 {
    for i := 0; i < len(num[0]); i++ {
        if '0' <= num[0][i] && num[0][i] <= '9' {
            temp := num[0][i] - '0'
            memsize = memsize*10 + int(temp)
        }       }   }
```

获取主机 IP 使用的方法是向外界发送一个 udp 请求，然后通过这个连接对象来得到主机 IP，关键代码如下:

```
conn, err := net.Dial("udp", "baidu.com:80")
defer conn.Close()
ip := strings.Split(conn.LocalAddr().String(), ":")[0]
```

4.心跳检测程序

心跳检测程序会从 etcd 中定时获取所有服务器节点列表，并与自己的监听服务器列表进行比对，如果有增加，则向监听列表中添加该节点。监听的法则是给每个节点开启一个协成，并使用 timeticker 进行定时 ping 目标主机，从而检测目标主机是否存活。

从 etcd 获取 Docker 主机地址信息列表，转为 engine 对象列表的核心代码:

```
var EngC []Engine
data, err := client.Get(ENGINE_DIR, false, true)
```

```
for _, Node := range data.Node.Nodes {
    key := strings.Split(Node.Key, "/")
    value := strings.Split(Node.Value, ":")
    e := Engine{key[2], value[0]}
    EngC = append(EngC, e)}
```

5.PaaS 控制器的 Docker 主机列表数据同步

PaaS 控制器需要定时与 etcd 存储的 Docker 主机节点列表同步。由于 Docker 服务器注册会向 etcd 中添加节点，或者由于心跳检测模块监听失败会想 etcd 中删除节点。因此 PaaS 需要添加一个定时器并轮询的检测 etcd 中节点列表是否有增减，如有就需要进行同步。

5.2.5 账户管理模块的实现

账户管理模块用来操作管理员信息，通过操作数据库和角色权限列表来实现对管理员账户的管理，账户管理类视图如下:

```
AccountOpt
session *r.Session
dbname    string
func (a *AccountOpt) InspectAccount() []*Account
func (a *AccountOpt) DeleteAccount(Id string) error
func (a *AccountOpt) AddAccount(acc *Account) error
```

账户管理操作类含有两个参数：session 和 dbname，session 用于记录与数据的会话 handler，dbname 记录数据库的名称。

接下来对账户管理类的主要函数进行介绍:

InspectAccount（）: 该类实现了从数据库中将所有账户的信息给提取出来，并封装为 Account 对象数组，其关键代码如下:

```
accs := []*Account{}
res, _ := r.Db(a.dbname).Table(tableAccount).Run(a.session)
res.All(&accs)
```

DeleteAccount(Id string): 该类实现了通过账户 id 将用户从数据中删除，具体动作为先通过该 id 从数据库检索该 id 是否存在，若不存在则返回错误。若存在则继续执行 delete 指令将指定 id 用户从数据库删掉。

AddAccount(acc *Account): 该类实现了将账户信息存入账户数据库的操作，具体操作为先通过数据库检查将要的添加的账户名字是否已经存在于已有用户中，若存在，返回错误。若不存在则将密码经过 SHA1 编码后保存到数据库。其中检测用户已存在的关键代码为:

r.Table(tableAccount).Filter(map[string]string{"username":acc.Username}).Run(a.session)

其中密码用 sha1 编码的关键代码为:

```
hash := sha1.New()
hash.Write([]byte(str))
sum:=hash.Sum([]byte("5i3QfJvpvvn3vQ0aXQeF")
hex.EncodeToString(sum)
```

最后一句是将字节码转为字符串，方便存于 rethinkDB 数据库，如果不转为字符串，经过数据库的一次存和一次取操作，数据就不是原来的数据了。以上代码中的 5i3QfJvpvvn3vQ0aXQeF 是干扰码，会在每个 sha1 计算得到的编码串前加入它们，然后将这整个字串作为密码存入数据库。

5.2.6 事件管理模块的实现

事件管理模块用于记录 PaaS 系统中所经历一些操作日志。事件的来源主要来自两部分，一部分是来自 Docker 服务器，这部分是用监听器去监听的。另外一部分是来自 Docker 主控器，主控制器会在一些重要操作中添加事件存入数据库待查。事件管理类视图如下:

```
EventOpt
session *r.Session
dbname  string
func (EH *EventOpt) Handle(e *citadel.Event) error
func (EH *EventOpt) saveDockerEvent(e *citadel.Event) error
func (EH *EventOpt)SaveEvent(e *Event)error
func (EH *EventOpt)GetEvents(count int)([]*Event,error)
```

事件管理类 EventOpt 有两个成员变量，一个是 session，是连接数据库的会话 handler; 另一个 dbname 记录连接的数据库名称。

事件管理类的成员函数介绍如下:

Handle(e *citadel.Event): 这个函数实现了实现了 EventHandler 接口，该接口的定义如下:

```
EventHandler interface {
    Handle(*Event) error}
```

该函数会被注册到 Dockerclient 的事件监听中，一旦有事件发生，就会主动调用接口中的 handle 函数，在这里会调用 eventOpt 的 Handle 函数，在收到 event 对象后，会调用 saveDockerEvent 函数将该事件生成一个新的 event 对象。该生成关键代码如下:

```
evt := &Event{
    Type:      e.Type,
    Time:      e.Time,
    Container: e.Container,
    Engine:    e.Engine,
    Tags:      []string{"Docker"},}
```

由 Docker 主机节点收集的所有事件都会打上 tag 为 Docker。

最后调用 SaveEvent 函数将事件函数存于数据库中。

在 PaaS 主控制器中如果想存事件，则在主控制器会将事件的对象生成，然后直接调用 saveEvent 函数将该事件直接存于数据库中。

GetEvents(count int): 该函数用于对外提供获取事件操作。该函数需要输入一个个数参数，指定需要返回的事件个数，该个数排序是按从新到旧的顺序排的。该函数的执行过程是: 该函数会首先查询事件表中的事件个数，关键代码如下:

```
var n int
number,_:=r.Table(tableEvent).Count().Run(EH.session)
number.One(&n)
```

然后将事件表中的事件个数与传入的参数进行对比，如果数据不够则报错。如果数据量够，则继续查询指定个数的事件，关键代码如下:

```
events:=[]*Event{}
res, err :=r.Table(tableEvent).Limit(count).Run(EH.session)
res.All(&events)
```

所有事件转换成 Event 对象之后返回该请求。

5.2.7 服务器管理模块的实现

服务器管理模块用来实现服务器节点的数据库相关的操作，通过对传入参数进行检测后对数据库进行相应的操作，从而得到返回值。服务器管理类视图如下:

session *r.Session

dbname string

func (m *EngineContro) GetAllEngine() []*Engine

func (m *EngineContro) AddEngine(e *Engine) error

func (m *EngineContro) DeleteEngine(id string) error

服务器管理操作类含有两个参数，一个是 session，用于记录与数据的会话 handler; 另外 dbname 用于记录数据库名称。

接下来对账户管理类的函数进行介绍:

GetAllEngine( ): 该函数实现了从数据库中将所有服务器节点的信息给提取出来，并封装为 Engine 对象数组，其关键代码如下:

```
engs := []*Engine{}
    res, _ := r.Db(m.dbname).Table(tableEngine).Run(m.session)
    res.All(&engs)
```

AddEngine(e *Engine): 该函数实现了将服务器节点通过 engine 对象添加到数据控中进行持久化存储。

DeleteEngine(id string): 该函数实现了通过 id 号从数据库中删除服务器节点，删除前需要比对数据库中是否存在该 id 节点，若存在则删除，若不存在则直接返回。

5.2.8 容器控制模块的实现

容器控制模块是该 PaaS 系统中的核心模块，该类实现了关于容器的所有功能，它的类视图如下:

Container

mux sync.Mutex

engines map[string]*citadel.Engine

schedulers map[string]citadel.Scheduler

resourceManager citadel.ResourceManager

func NewContainer(c *cluster.Cluster) *Container

func (m *Container) Listcontainer(all bool) []*citadel.Container

func (m *Container) InspectContainer(id string) (*citadel.Container, error)

func (m *Container) RunContainer(img *citadel.Image, count int, pull bool) ([]*citadel.Container, error)

func (m *Container) ScaleContainer(container *citadel.Container, count int)

func (m *Container) StartContainer(container *citadel.Container) error

func (m *Container) RestartContainer(container *citadel.Container) error

func (m *Container) StopContainer(container *citadel.Container) error

func (m *Container) DestroyContainer(container *citadel.Container) error

func (m *Container) ContainerLogs(container *citadel.Container)

func (m *Container) IdenticalContainers(container *citadel.Container)

func (m *Container) getContainersByName(name string)

该类中含有为了多客户访问而添加了互斥量 mux，还有服务器列表 engines，服务器过滤器 schedulers，资源调度器 resourceMangager。

下面介绍成员函数:

Listcontainer(all bool) ：函数实现了从各个服务器节点中搜集容器信息，并将它们组成容器对象数组，其中的参数 all 是用于控制是否将未运行的容器也搜集起来。该函数内部实现是通过遍历已有的服务器节点列表，然后对每个服务器对象调用其客户端 ListContainers 函数，从而得到容器信息并添加到容器数组当中返回。

InspectContainer(id string): 该函数实现了对一个容器 id 进行查询得到数据信息它的信息结构如下:

{"id":"714e01115f76ffa9d468379ba23414abec0a0c93a96cc8766077ce76f1a89f82","name":"/backstabbing_elion","image":{"name":"ubuntu:14.04","cpus":0.1,"memory":20,"hostname":"714e01115f76","type":"service","labels":[""],"restart_policy":{"name":"no"},"network_mode":"bridge"},"engine":{"id":"1921687101","addr":"http://192.168.7.101:2375","cpus":2,"memory":1533,"labels":["Docker"]},"state":"stopped"}

关键代码如下:

```
for _, container := range containers {
    if strings.HasPrefix(container.ID, id) {return container, nil}}
```

由于只知道 id，不值得其服务器信息，因此需要首先获得所有容器数据，然后通过遍历所有容器，匹配其 id 是否一致，将 id 一致的容器信息作为查询结果发送给用户。

RunContainer(img *citadel.Image, count int, pull bool) 该函数实现了将容器镜像部署到服务器中，通过 count 参数控制部署的个数，通过 pull 控制是否需要进行 pull 操作。其内部实现为：首先创建了一个同步器，用于同步所有任务都执行完成。通过协程 Gofunc 将 n 个容器运行指令并发执行。在 start 中，会遍历所有的服务器节点，通过过滤器过滤出可以运行该容器镜像的节点并组成数组，然后通过资源调度器调度可以运行的容器中选出一台来运行部署该容器镜像。其中过滤器的关键代码如下:

```
for _, c := range containers {
    if c.Image.Name == fullImage {return true}}
```

资源调度器的执行过程为：首先进行遍历可用的容器，然后对每个容器中的容器进行遍历，算出每个容器的得分，然后对分数进行排序，得到资源占用最少，可用资源最多的服务器节点进行返回。

ScaleContainer(container *citadel.Container, count int) ：该函数实现了对已运行的容器的个数进行重新制定。内部实现为首先利用 IdenticalContainers 函数获得所有的与输入容器对象一致的已运行容器列表，然后比较两者数量。若比较结果为需要运行的数量大，则调用 RunContainer 运行差额个数的容器镜像。若比较结果需要删减运行容器数量，则调用 DestroyContainer 函数将多余个数的容器进行销毁。

StartContainer(container *citadel.Container): 该函数实现的是将一个已经停止运行的容器启动,传入时需要将容器的 Docker 主机信息也一起传入,这个 Docker 主机信息可以由主机查询得到。

RestartContainer(container *citadel.Container): 这个函数实现的是将一个容器重新启动，调用的是该容器对应的服务器节点对象中存储的客户端来操作的。

StopContainer(container *citadel.Container) ：该函数实现的是将一个已经运行的容器暂停运行，内部实现是通过该容器对象中服务器客户端对象来操作的。

DestroyContainer(container *citadel.Container) ：该函数实现的是将一个容器进行销毁操作，不管其是否运行。其内部实现为先使用 Docker 客户端的 kill 命令将服务器上的该容器杀死。然后使用 dokcer 客户端的 remove（container）将

该容器从容器列表中删除，从而实现将该容器实体从磁盘中清除，删除后无法使用 start 函数再启动。

ContainerLogs(container *citadel.Container) ：该函数实现的是从一个服务器节点中获取传入容器的所有 log 数据，包括标准输出和错误输出。其内部实现为：调用客户端发送获取 Docker 主机日志的请求，将 Docker 主机返回的消息 body 直接传给客户端。关键代码如下:

```
req, err := http.NewRequest("GET", client.URL.String()+uri, nil)
    req.Header.Add("Content-Type", "application/json")
    resp, err := client.HTTPClient.Do(req)
```

IdenticalContainers(container *citadel.Container) 该函数作为 scale 的辅助函数，其实现的功能为从所有容器中找出与传入 container 匹配的容器，并组成数组返回。其匹配依据为启动参数，分配内存，镜像类型。

5.2.9 客户端的实现

客户端实现了使用命令行对远端 PaaS 系统进行控制和操作，它可以调用 PaaS 系统中 http 服务模块所提供的所有功能。其内部实现如下:

1.创建一个代理 PaaSProxy 封装所有 http 请求，然后在基于基础封装之上实现所有功能函数。请求的基础封装如下:

```
func execRequest(method, path string, content []byte, header map[string]string,
acceptStatus int) (*http.Response, error) {
    bodyBuff := bytes.NewBuffer(content)
    req, err := http.NewRequest(method, path, bodyBuff)
    for key, value := range header {req.Header.Add(key, value)}
    resp, err := http.DefaultClient.Do(req)
    if resp.StatusCode != acceptStatus {
        tmp, err := ioutil.ReadAll(resp.Body)
        data := fmt.Sprintf("%d\n%s", resp.StatusCode, tmp)
        return nil, errors.New(data)}
    return resp, nil}
```

该函数需要传入请求的方法，路径，body 内容，http 头部内容，期望返回的结果值。在内部实现中，首先创建一个 request 对象，传入方法，路径，内容

生成 request 对象。然后给该 request 对象设置 header 值，一般这里需要设置 token。

2.交互

交互这块实现的是与命令行界面的交互，将命令行的输入转换为内部数据，同时生成对象，调用相关方法，实现相应功能。将指令绑定到函数的关键代码如下:

```
CommondLogin = cli.Command{
    Name:     "login",
    Usage:    "login to DockerPaaS to get token",
    Action: Login,}
```

在 name 中设置指令名字，在 usage 中设置该命令的说明，在 action 中设置调用函数名。当用户在命令行中敲入 login 指令时就会启动 login 函数。

通过从命令行中读入数据，然后生成对象，调用代理中的相关函数。登入函数中要求从命令中敲入用户名及其密码，然后将用户名和密码使用代理的 login 函数进行登入操作，得到 token 后存入本地待下次使用其他命令时使用。

其他交互功能实现也基本差不多，在此省略。

5.3 本章结论

本章给出了 PaaS 系统的各个模块的具体实现，包括类视图和对类函数进行说明。还将部分关键代码附上并说明其执行过程。

第6章 基于 Docker 的企业 PaaS 系统的测试

在系统测试过程中，及时发现系统设计及实现过程中的问题和程序代码的错误，保证系统功能完整，正确，运行可靠，并且能达到相关性能指标。本章采用逐步递增增范围的方式进行，先对各个模块进行单元测试，再对集成后的系统进行功能验证测试，验证功能的正确性和完整性。

6.1 测试方案

6.1.1 测试目标

测试 PaaS 系统可以满足所有功能性需求，在以上需求同时满足之后再尽量满足非功能性需求。

保证 PaaS 系统能够稳定可靠的长时间运行，无需用户经常维护。

6.1.2 测试范围

测试范围包括登入和认证模块，注册和心跳检测模块，账户管理模块，服务器管理模块，账户管理模块，http 服务模块，容器控制模块，客户端。单元测试在开发的时候已经完成，本测试主要采用模块功能测试，测试所有功能是否正常运行。

6.1.3 测试环境

测试环境为有 Windows7 64 位，CPU I5 双核 4 线程，内存 8G 作为 PaaS 系统的主控制器运行场所，在该机上的 virtual box 虚拟出两台 ubuntu14.04 64 bit 的主机。第一台主机是双核处理器，1.5G 内存，作为 Docker 服务器。第二台主机是单核处理器，内存为 1G，作为 rethinkDB 和 etcd 的运行系统，还负责心跳检测模块。

6.2 系统测试

6.2.1 功能测试

1.登入和认证模块测试

通过用户名 clare，密码 659559700 登入 PaaS 平台：127.0.0.1：5002，如果密码正确将得到 PaaS 平台分配的 token，利用该 token 与 PaaS 进行消息请求。

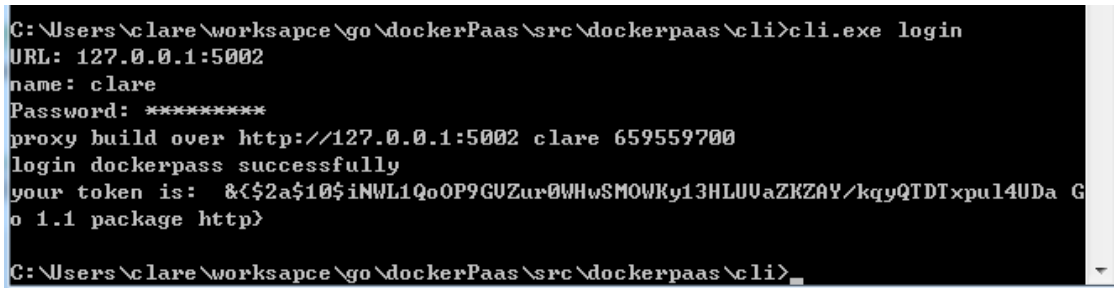

图 6.1 登入

2.注册和心跳检测模块

服务器注册时服务器节点会自动运行 reg 程序，该程序会将服务的 ip，服务端口，cpu 数量，内存容量发送到 etcd 中，注册自己。

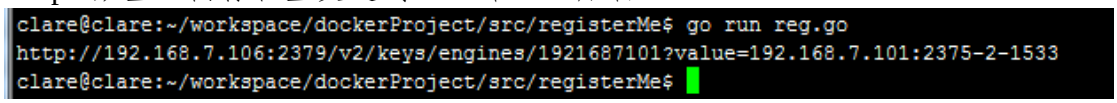

图 6.2 服务注册

注册成功后，心跳检测模块就会侦测到 etcd 中有服务器注册，然后读取该服务器 ip 信息，并将其加入监听列表。

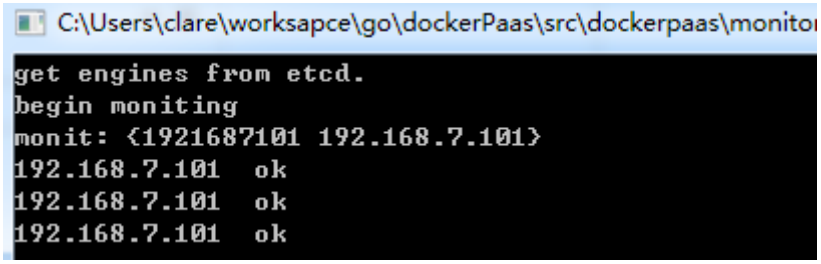

图 6.3 服务器监听

PaaS 主控器也会检测到 etcd 中有服务注册，并将该服务器加入 PaaS 中，开始使用该服务器节点的服务。

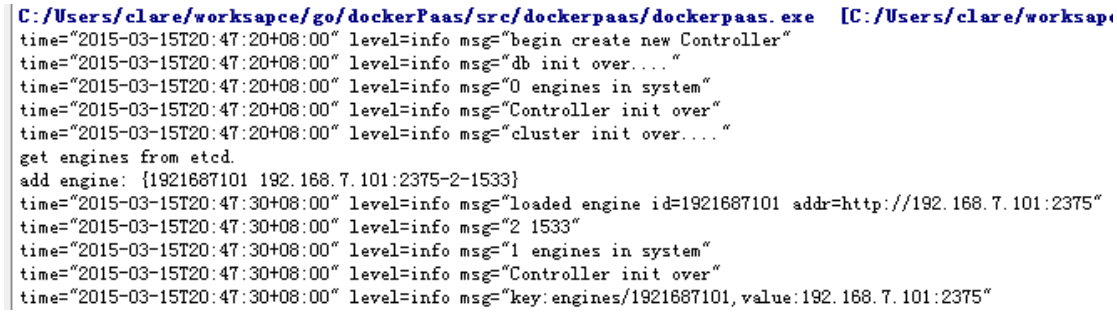

图 6.4 服务器节点加入 PaaS

3.账户管理模块测试:

向 PaaS 系统中添加 clare2 账户，然后再将其删除，从而测试张含韵功能完整性。

```
C:\Users\clare\worksapce\go\dockerPaas\src\dockerpaas\cli>cli.exe accounts
0 6789 clare

C:\Users\clare\worksapce\go\dockerPaas\src\dockerpaas\cli>cli.exe aadd --id 1234
 --name clare2 --password 123456 --role user

C:\Users\clare\worksapce\go\dockerPaas\src\dockerpaas\cli>cli.exe accounts
0 1234 clare2
1 6789 clare

C:\Users\clare\worksapce\go\dockerPaas\src\dockerpaas\cli>cli.exe adelete clare2

delete account:  clare2

C:\Users\clare\worksapce\go\dockerPaas\src\dockerpaas\cli>cli.exe accounts
0 6789 clare

C:\Users\clare\worksapce\go\dockerPaas\src\dockerpaas\cli>
```

图 6.5 账户管理测试

4.服务器管理模块测试:

该模块的功能已经在心跳检测模块测试中体现出来。

Docker 主机也可以手动添加，下面进行测试:

```
C:\Users\clare\worksapce\go\dockerPaas\src\dockerpaas\cli>cli.exe engines
1921687101 http://192.168.7.101:2375

C:\Users\clare\worksapce\go\dockerPaas\src\dockerpaas\cli>cli.exe eadd --id 1921
687106 --addr http://192.168.7.101:2375

C:\Users\clare\worksapce\go\dockerPaas\src\dockerpaas\cli>cli.exe engines
1921687101 http://192.168.7.101:2375

C:\Users\clare\worksapce\go\dockerPaas\src\dockerpaas\cli>cli.exe eadd --id 1921
687106 --addr http://192.168.7.101:2375
```

图 6.6 系统 Docker 主机手动管理测试

5.事件模块测试:

获取最近十条 PaaS 系统操作日志。

```
C:\Users\clare\worksapce\go\dockerPaas\src\dockerpaas\cli>cli.exe events 10
 add-account name=clare
 add-account name=clare
 add-account name=clare
 add-account name=clare
 delete-engine id=1921687102
 add-account name=clare
 add-account name=clare
 add-account name=clare
 add-engine addr=http://192.168.7.102:2375
 add-account name=clare
```

图 6.7 事件监听模块测试

6.容器控制模块测试:

运行一个 httpd 服务，PaaS 系统会自动分配一个主机上的端口绑定到 Docker 内部的 80 端口。

```
C:\Users\clare\worksapce\go\dockerPaas\src\dockerpaas\cli>cli.exe containers
634c429d7d34249670f5f67caa5a9be265614ee165c8fb1677d8fc99eaf450f3 httpd:2 http://
192.168.7.102:2375 running &{tcp 0.0.0.0 22668 80}
3155cee9b28cf3d5b0d1d69bffdc8303c3802edf531b99d985cc5cd9c036f907 httpd:2 http://
192.168.7.102:2375 running &{tcp 0.0.0.0 5510 80}
e1f1f954fbe5a127bae276ccee4e6bb03f72e8881aa162466a0df78f84829aa6 httpd:2 http://
192.168.7.102:2375 running &{tcp 0.0.0.0 5505 80}

C:\Users\clare\worksapce\go\dockerPaas\src\dockerpaas\cli>cli.exe run --name htt
pd --port tcp/::80 --memory 100 --pull false
pro:tcp,hostport:21121,docker port:8005e03fe8c68917be01b0532a03baf9d8d3e88cfb1e4
e8751a2a48d032fffd155

C:\Users\clare\worksapce\go\dockerPaas\src\dockerpaas\cli>cli.exe containers
05e03fe8c68917be01b0532a03baf9d8d3e88cfb1e4e8751a2a48d032fffd155 httpd:2 http://
192.168.7.102:2375 running &{tcp 0.0.0.0 21121 80}
634c429d7d34249670f5f67caa5a9be265614ee165c8fb1677d8fc99eaf450f3 httpd:2 http://
192.168.7.102:2375 running &{tcp 0.0.0.0 22668 80}
3155cee9b28cf3d5b0d1d69bffdc8303c3802edf531b99d985cc5cd9c036f907 httpd:2 http://
192.168.7.102:2375 running &{tcp 0.0.0.0 5510 80}
e1f1f954fbe5a127bae276ccee4e6bb03f72e8881aa162466a0df78f84829aa6 httpd:2 http://
192.168.7.102:2375 running &{tcp 0.0.0.0 5505 80}
```

图 6.8 本 PaaS 服务运行测试

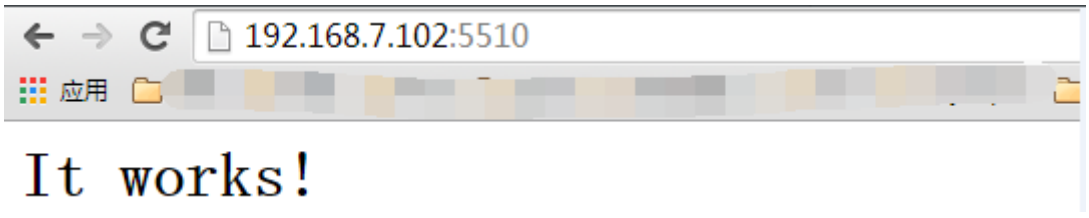

图 6.9 运行 httpd 服务展示

7.容器数量重置测试:

测试内容为由五个 httpd 服务重置为 2 个，然后又重置为 4 个，结果如下:

```
C:\Users\clare\worksapce\go\dockerPaas\src\dockerpaas\cli>cli.exe containers
c4cf3f446bd65cc68936b00524c2667c2da140d335d465b2ad5c5f3acd30bdd6 httpd:2 http://
192.168.7.102:2375 running &{tcp 0.0.0.0 21898 80}
d13cc0ba9b7d23ff5d423e88cce20038edd8a9c6c1b25c24942bffc518d75942 httpd:2 http://
192.168.7.102:2375 running &{tcp 0.0.0.0 21876 80}
8a45dd251d3b37b35f4d798f02dafb3696a7712b25161b360fbe991f8f9909dc httpd:2 http://
192.168.7.102:2375 running &{tcp 0.0.0.0 5918 80}
3155cee9b28cf3d5b0d1d69bffdc8303c3802edf531b99d985cc5cd9c036f907 httpd:2 http://
192.168.7.102:2375 running &{tcp 0.0.0.0 5510 80}
e1f1f954fbe5a127bae276ccee4e6bb03f72e8881aa162466a0df78f84829aa6 httpd:2 http://
192.168.7.102:2375 running &{tcp 0.0.0.0 5505 80}

C:\Users\clare\worksapce\go\dockerPaas\src\dockerpaas\cli>cli.exe scale e1f1f954
fbe5a127bae276ccee4e6bb03f72e8881aa162466a0df78f84829aa6 2

C:\Users\clare\worksapce\go\dockerPaas\src\dockerpaas\cli>cli.exe containers
3155cee9b28cf3d5b0d1d69bffdc8303c3802edf531b99d985cc5cd9c036f907 httpd:2 http://
192.168.7.102:2375 running &{tcp 0.0.0.0 5510 80}
e1f1f954fbe5a127bae276ccee4e6bb03f72e8881aa162466a0df78f84829aa6 httpd:2 http://
192.168.7.102:2375 running &{tcp 0.0.0.0 5505 80}

C:\Users\clare\worksapce\go\dockerPaas\src\dockerpaas\cli>cli.exe scale e1f1f954
fbe5a127bae276ccee4e6bb03f72e8881aa162466a0df78f84829aa6 4

C:\Users\clare\worksapce\go\dockerPaas\src\dockerpaas\cli>cli.exe containers
e489ee8cbe6f8f92229c4d22b9b06a653389397b26ad5fb097b0b5d1292ac3b8 httpd:2 http://
192.168.7.102:2375 running &{tcp 0.0.0.0 13548 80}
871dca3d62fafff29ca485dfbe633dbb6a8485a17e2400b10780dc883afb9973 httpd:2 http://
192.168.7.102:2375 running &{tcp 0.0.0.0 13516 80}
3155cee9b28cf3d5b0d1d69bffdc8303c3802edf531b99d985cc5cd9c036f907 httpd:2 http://
192.168.7.102:2375 running &{tcp 0.0.0.0 5510 80}
e1f1f954fbe5a127bae276ccee4e6bb03f72e8881aa162466a0df78f84829aa6 httpd:2 http://
192.168.7.102:2375 running &{tcp 0.0.0.0 5505 80}
```

图 6.10 重置 httpd 服务数量测试

6.3 测试评价

通过功能测试对系统所有功能进行测试后，证明系统已经具备满足需求分析阶段提出的功能性需求。针对非功能性需求，系统无法进行完善的性能测试，稳定性测试经过 24*7 稳定运行，满足了简单易用稳定的需求，基本符合验收要求。

6.4 系统界面展示

```
C:\Users\clare\worksapce\go\dockerPaas\src\dockerpaas\cli>cli.exe
NAME:
   dockerPaas - use docker as paas main component

USAGE:
   dockerPaas [global options] command [command options] [arguments...]

VERSION:
   0.1

AUTHOR:
  clare - <helloavr@gmail.com>

COMMANDS:
   containers   show all the containers's information
   container    container id could get the container info
   run          run a container
   destroy      destroy a container
   stop         stop a container
   start        start a container
   restart      restart a container
   scale        scale container
   log          get container's log
   engines      get all engines's list
   engine       get one engines's info
   eadd         add one engine
   edelete      delete one engines's info
   clusterinfo  get cluster's info
   accounts     get all account's list
   account      get one account's info
   aadd         add one account
   adelete      delete one account
   events       events count,get count events
   login        login to dockerpaas to get token
   help, h      Shows a list of commands or help for one command

GLOBAL OPTIONS:
   --help, -h                   show help
   --generate-bash-completion
   --version, -v                print the version
```

图 6.11 系统主界面

6.5 本章小结

本章首先对系统的测试方案进行描述，说明测试目标、范围、环境的配置等。并进行了系统的各个模块的功能性测试和 PaaS 系统的非功能性测试。并在最后进行了总结分析。证明系统已经基本满足需求分析阶段提出的要求，最后给出了系统的总运行界面。

第7章 结论

为了满足中小企业对于 PaaS 系统的需求,在对基于 Docker 的用例分析和流程规划的基础上,本文着重探讨了如何利用 Docker 和及其他组件的集成和编码。笔者主要参与了系统的需求分析，设计，以及 PaaS 系统所有模块的功能实现和测试工作。

本文通过国内外对 PaaS 和容器技术的研究现状以及其发展趋势分析，整理了 PaaS 系统的用例描述和业务流程，通过深入剖析现有 PaaS 云平台的优缺点以及其对中小企业的不恰之处。决定整合已有的一些开源程序以及设计编写集成和程序,按照模块化的设计理念,利用总分方式开发了基于 Docker 的企业 PaaS 系统。在开发过程中，采用模块化组件的开发方式，先开发每个功能模块，然后利用总控制器代理各个模块组成一个完整功能的系统，并且使系统的架构更加清晰，同时使模块与模块之间的耦合性降到最低。在开发过程中，采用边开发边测试的方式保证代码准确性，模块完成之后采用功能测试来确定模块集成的可用性。

在系统的设计和实现过程中，为了满足云计算平台分布式的特点以及中小企业资源有限的特点，本文提出了一些方法来实现用户的需求:1.利用 etcd 作为服务器注册和发现的工具，这样做使得系统的耦合性降低，心跳检测，服务器节点同步，服务器注册可以在三个地方进行。2. 使用 rethinkDB 作为系统信息数据库，这样做可以将一个数据对象直接利用 json 编码器编码为 json 对象保存与数据库中，同时也拥有高效的查询性能，做到了部署轻松，使用简单，性能高效的特点。3.本文提出了使用 Docker 容器作为应用服务的载体，很好的解决了中小企业运行环境灵活多变，服务需求多变的特点，将应用及其运行环境和依赖统一打包为 Docker 镜像的方式进行发布,无需做复杂的适配和调试。在 Docker 上运行该 Docker 镜像就完整的复原了开发测试时的应用服务。

在 PaaS 系统的测试途中也发现了几个问题需要改善: 容器在重启后会导致 ip 的变化，这虽然对外部没有影响，因为外部是通过主机端口映射到内部主机端口，但是对于内部链接访问会有很大问题。系统没有在应用级做容错处理，因此如果主服务器宕机则控制器也会停止工作，因此需要在此之上做冗余操作。本文虽然使用的是 PaaS 的理念，但是其自身并没有做各种应用服务，而是让用

户自己选择服务容器的启动即应用。因此，需要使用者对容器镜像相关知识有一定的了解。

参考文献

[1]Bhardwaj S, Jain L, Jain S. Cloud computing: A study of infrastructure as a service (IAAS)[J]. International Journal of engineering and information Technology, 2010, 2(1): 60-63.
[2]王禹华. PaaS 云管理系统设计与实现[D]. 北京邮电大学, 2013.
[3]徐鹏, 陈思, 苏森. 互联网应用 PaaS 平台体系结构[J]. 北京邮电大学学报, 2012, 35(1): 120-124.
[4]Joha A, Janssen M. Design Choices Underlying the Software as a Service (SaaS) Business Model from the User Perspective: Exploring the Fourth Wave of Outsourcing[J]. J. UCS, 2012, 18(11): 1501-1522.
[5]罗军舟, 金嘉晖, 宋爱波, 等. 云计算: 体系架构与关键技术[J]. 通信学报, 2011, 32(7): 3-21.
[6]陆钢. 电信运营商云计算 PaaS 发展关键问题探析[J]. 广东通信技术, 2011, 31(7): 2-5.
[7]蔡文君, 裴培, 杨巧霞. 浅析 PaaS 平台在电信运营商业务支撑系统中的应用[J]. 邮电设计技术, 2012, 7: 003.
[8]Zahariev A. Google app engine[J]. Helsinki University of Technology, 2009.
[9]Redkar T, Guidici T. Windows Azure Platform[M]. New York: Apress, 2009.
[10]Wilder B. Cloud Architecture Patterns: Using Microsoft Azure[M]. " O'Reilly Media, Inc.", 2012.
[11]Van Vliet J, Paganelli F, van Wel S, et al. Elastic Beanstalk[M]. " O'Reilly Media, Inc.", 2011.
[12]Collison D. Distributed Design and Architecture of Cloud Foundry[J]. 2012.
[13]彭麟. Cloud Foundry 技术全貌及核心组件分析[J]. 程序员, 2013 (1): 104-107.
[14]徐铖. Cloudfoundry 中 warden 框架的设计与实现[D]. 江苏: 南京大学, 2013.
[15]Fawcett A. Force. com Enterprise Architecture[M]. Packt Publishing Ltd, 2014.
[16]Sapir J, Wood S. The Executives Guide to force. com: Shadow IT and Citizen Developers in the Age of Cloud Computing[J]. 2012.
[17]Hanjura A. Heroku Cloud Application Development[M]. Packt Publishing Ltd, 2014.
[18]Kemp C, Gyger B. Professional Heroku Programming[M]. John Wiley & Sons, 2013.
[19]Cohen B. PaaS: New Opportunities for Cloud Application Development[J]. Computer, 2013 (9): 97-100.
[20]BAE, http://developer.baidu.com/cloud/rt.
[21]ACE,http://www.aliyun.com/product/ace.
[22]SAE,http://sae.sina.com.cn/.
[23]丛磊. Sina App Engine 架构——云计算时代的分布式 Web 服务解决方案[J]. 程序员, 2010 (11): 59-62.
[24]Che J, Yu Y, Shi C, et al. A synthetical performance evaluation of openvz, xen

and kvm[C]Services Computing Conference (APSCC), 2010 IEEE Asia-Pacific. IEEE, 2010: 587-594.
[25]Merkel D. Docker: lightweight linux containers for consistent development and deployment[J]. Linux Journal, 2014, 2014(239): 2.
[26]Tang X, Zhang Z, Wang M, et al. Performance Evaluation of Light-Weighted Virtualization for PaaS in Clouds[M]AlGorithms and Architectures for Parallel Processing. Springer International Publishing, 2014: 415-428.
[27]Daniels J. Server virtualization architecture and implementation[J]. Crossroads, 2009, 16(1): 8-12.
[28]des Ligneris B. Virtualization of Linux based computers: the Linux-VServer project[C]High Performance Computing Systems and Applications, 2005. HPCS 2005. 19th International Symposium on. IEEE, 2005: 340-346.
[29]Lessard P. Linux process containment: A practical look at chroot and user mode Linux[J]. 2003.
[30]HandiGol N, Heller B, Jeyakumar V, et al. Reproducible network experiments using container-based emulation[C]Proceedings of the 8th international conference on Emerging networking experiments and technologies. ACM, 2012: 253-264.
[31]Xavier M G, Neves M V, Rossi F D, et al. Performance evaluation of container-based virtualization for high performance computing environments[C]Parallel, Distributed and Network-Based Processing (PDP), 2013 21st Euromicro International Conference on. IEEE, 2013: 233-240.
[32]Docker 架构图,
http://www.infoq.com/cn/articles/Docker-source-code-analysis-part1.
[33]Helsley M. LXC: Linux container tools[J]. IBM devloperWorks Technical Library, 2009.
[34]Hitz D, Malcolm M, Lau J, et al. Copy on write file system consistency and block usage: U.S. Patent 6,892,211[P]. 2005-5-10.
[35]唐丹，金海，张永坤. 集群动态负载平衡系统的性能评价[J]. 计算机学报, 2004, 27(6): 803-811.
[36]游小明,罗光春. 云计算原理与实践[M].2014.
[37]Walsh L, Akhmechet V, Glukhovsky M. Rethinkdb-rethinking database storage[J]. 2009.
[38]陈莉莹，双锴. NoSQL 数据库综述[D]. 北京: 北京邮电大学网络技术研究院, 2012.
[39]Hunt P, Konar M, Junqueira F P, et al. ZooKeeper: Wait-free Coordination for Internet-scale Systems[C]USENIX Annual Technical Conference. 2010, 8: 9.
[40]开源的服务发现项目 Zookeeper，Doozer，Etcd.
http://www.tuicool.com/articles/e6viimI,2014.
[41]Ongaro D, Ousterhout J. In search of an understandable consensus alGorithm[C]Proc. USENIX Annual Technical Conference. 2014: 305-320.

作者简历

教育经历:

2009 年 9 月到 2013 年 6 月就读于浙江工业大学，计算机科学与技术和自动化双专业，获得工学学位。

2013 年 9 月到 2015 年 6 月就读于浙江大学，软件工程专业，获得硕士学位。

工作经历:

2014 年 4 月到 2015 年 2 月在上海 SAP 参加 SDE 实习。

致谢

感谢我的校内导师施青松老师，感谢他在我撰写论文的过程中提出宝贵的意见，帮助我提高论文质量。

感谢我的校外导师 James 和 Frank，以及实习单位的各位同事，是你们给我机会接触 Docker 和 cloudfoundry，给了我很多理论和实践上的指导。

感谢父母在我毕业设计期间给予精神上的支持，使我能顺利完成本毕业设计。

陈东东

于浙江大学软件学院