# 基于Docker的应用负载均衡优化

作者姓名：郝运 专业班级：2014130601 指导老师：覃章建

# 摘要

随着云技术的发展和普及，云计算无疑是当今互联网行业的关键词。云计算如火如荼的发展与Docker的产生和发展息息相关，Docker的出现让应用的开发、发布和部署呈现出前所未有的便捷和高效。但是Docker只专注于提供容器和镜像，企业需要一个容器的管理平台以对容器进行编排部署、资源调度、负载均衡和服务发现。在众多的管理平台中，Kubernetes有着极其强大的容器编排能力和资源调度能力。

本论文使用Kubernetes的解决方案对基于Docker的应用的负载均衡进行了相关的优化和研究。利用Docker将应用容器化，并采用kubernentes的解决方案实现对容器的调度，编排和动态伸缩。为了实现所涉及的pass平台的高并发性能，后端代码采用Golang语言，前端代码使用HTML，JavaScript，Ajax，数据库采用Mysql。用户在Web界面去管理应用的上下线、一键部署和迭代回滚等

本论文实现的Slyph容器调度系统，性能强大、操作简便，良好支持Devops的开发方式和对应用的动态伸缩。

关键词：Kubernetes；Ajax；Golang

# Application load balancing optimization based on Docker.

**Abstract:** With the development and popularization of cloud technology, cloud computing is undoubtedly the key word in today's Internet industry.The development of cloud computing is closely related to the generation and development of Docker. The emergence of Docker has made the application development, release and deployment more convenient and efficient than ever before.Since Docker focuses only on providing containers and images, the enterprise also needs a management platform for the container cloud to implement the orchestration deployment, resource scheduling, load balancing, and service discovery for the container.Kubernetes has become the leader of many container cloud management systems.

In this paper, the application load balancing optimization is studied and designed.Use Docker to containerize and use the solution of kubernentes to implement container scheduling, orchestration, and dynamic scaling.In order to achieve the high concurrency performance of the pass platform involved, the back-end code USES the Golang language, and the front-end code USES HTML, JavaScript, Ajax and Mysql.Administrators can manage the application lifecycle and iteration rollback through the Web interface.At the same time, this pass platform supports one key deployment.

The Slyph container scheduling system implemented in this paper is powerful and easy to operate. It supports the development mode of Devops and the dynamic scaling of the application.

**Keywords**: Kubernetes; Ajax; Golang

目录

[第一章 绪论 1](#_Toc23210)

[1.1 国内外基于Docker应用负载均衡的现状与发展 1](#_Toc9417)

[1.2 课题研究的目的及意义 2](#_Toc23465)

[第二章 Slyph框架设计 3](#_Toc31811)

[2.1需求设计 3](#_Toc31733)

[2.1.1后端需求分析 4](#_Toc1231)

[2.1.2 前端需求分析 4](#_Toc3363)

[2.2 环境及相关实例 5](#_Toc31733)

[2.2.1 Master 6](#_Toc3363)

[2.2.2 Node 6](#_Toc3363)

[2.2.3 Pod 6](#_Toc3363)

[2.2.4 Deployment 7](#_Toc3363)

[2.2.5 Service 7](#_Toc3363)

[2.2.6 Statefulset 7](#_Toc3363)

[2.3技术栈 7](#_Toc15639)

[第三章 Slyph后端设计 8](#_Toc29285)

[3.1 MVC设计 8](#_Toc31559)

[3.2 脚本文件编写 8](#_Toc13884)

[3.2.1 Dockerfile 8](#_Toc15981)

[3.2.2 Makefile 8](#_Toc1006)

[第四章 Slyph前端实现 10](#_Toc13593)

[4.1框架选择 10](#_Toc2242)

[4.2前端界面设计 10](#_Toc21436)

[4.3功能模块实现 15](#_Toc28311)

[4.3.1管理员登录 15](#_Toc412)

[4.3.2 Pod实例的显示与管理 17](#_Toc22784)

[4.3.3 Deployment的显示与管理 19](#_Toc12273)

[4.3.4 Service的显示与管理 21](#_Toc10356)

[4.3.5 有状态应用的显示与管理 23](#_Toc10356)

[4.3.6 存储的显示与管理 25](#_Toc10356)

[4.3.7 一键部署 27](#_Toc10356)

[4.3.8 日志 29](#_Toc10356)

[第五章 Slyph数据库设计 32](#_Toc18644)

[5.1数据库的对比与选择 32](#_Toc20086)

[5.2部署Mysql容器 32](#_Toc20086)

[5.3数据库表结构设计 32](#_Toc20086)

[5.4 Mysql的连接配置 35](#_Toc20086)

[第六章 系统测试 36](#_Toc22360)

[第七章 总结 37](#_Toc22360)

[致谢 38](#_Toc27898)

[参考文献 39](#_Toc20423)

# 

# 第一章 绪论

## 1.1国内外基于Docker应用负载均衡的现状与发展

负载平衡是一种计算机技术，其目的是为了在多个服务器、网络、CPU、内存或其他资源中分配负载，以达到最高效资源使用、最大吞吐率、最小响应时间，同时避免资源过载其中传统应用的负载均衡方式是利用一台服务器作为负载均衡器，多台服务器供其调度。(王舜，2013)我们可以构建不可变的虚拟机映像，以实现可预测的滚出和回滚，但是vm是重量级的，不可移植的。传统的负载均衡运维难度高，数据迁移慢，硬件成本高，动态扩容复杂已经不适用于当今的云计算时代。

云计算时代中最重要的技术就是虚拟化技术，虚拟化技术有KVM、Xen、Hyper-V，Docker等，其中做的最好的就是Docker。Docker化的应用可以在几秒钟时间内启动， Docker能够高效利用宿主机资源，一台服务器上可以同时运行多个 Docker。（何松林，2017）Docker无疑Developer们的福音，因为它解决了交付过程中的环境问题，开发好的程序只需要打包进Docker一起发送给客户，客户可以开包即用，无需担心繁杂的环境问题。

国内的负载均衡的现状是大部分负载均衡是基于虚拟机的，利用单独的一个硬件设备作为负载均衡器去调度几台分布式的服务器。如，腾讯云提供两种负载均衡的方式，一种负载均衡只负责转发流量，不具备处理请求的能力。这种负载均衡相当于一个网关，将域名与负载均衡实例绑定后，用户访问时就会由负载均衡实例将用户的流量分配给后端的一台虚拟机。另一种是应用型负载均衡，这种负载均衡根据用户的域名的转发规则，将请求转发到指定的后端应用中。

国外的负载均衡现状是基于虚拟机并逐步支持Docker。大部分服务商的负载均衡不仅支持流量的转发、请求的重定向，还支持动态扩容，蓝绿更新。如，AWS的Amazon ELB，ELB通过在多个实例和资源可用区之间自动均衡流量，同时确保只有状态正常的实例收到流量，为应用程序提供容错能力。如果某个资源可用区内的所有实例均为非正常状态，ELB 将把流量路由至另一个资源可用区内的正常实例。

ELB 还支持Docker化应用程序的自动负载均衡，利用ELB与Amazon EC2 Container Service的深度集成只需将Docker服务注册到负载均衡器中，ECS 就会自动管理 Docker 容器的注册和注销。谷歌的Kubernetes是一个可移植的、可扩展的开源平台，用于管理集装箱化的工作负载和服务，这有助于声明式配置和自动化。Kubernetes的服务、支持和工具随处可见。(张迟第，2017)Kubernetes 对应用的负载均衡的方式目前有三种：LoadBlancer Service、NodePort Service、Ingress。Kubernetes将Docker作为Containner放到每个Pod内，通过Deployment控制Pod的动态扩容，并通过Service将服务暴露出来。（Yegulalp，2016）

## 1.2 课题研究的目的及意义

应用的高负载均衡是长久以来的需求，保证应用在大流量高并发请求下正常运行是现阶段应用开发必不可少的环节。Kubernetes在应用的负载均衡方面做的十分出色并且Kubernetes是一个可移植的、可扩展的开源平台，用于管理Docker化的工作负载和服务，有助于声明式配置和自动化开发同时Kubernetes是由强大的开源社区推动开发的，其服务、支持和工具随处可见。（周佳威，2017）

云计算技术发展的十多年以来，有IaaS基础设施及服务、PaaS平台及服务、SAAS软件及服务的三种服务模式，随着容器技术的发展，衍生出了一种新的服务模式：CAAS容器及服务。（仇臣，2016）在应用的开发过程中将应用容器化方便了应用在不同环境中的迁移，同时Docker轻量级的特性使其能在更少的资源消耗下提供更优质的服务。在实际的开发过程中一个应用的服务是分布在不同的容器中的，因此企业需要一个对容器进行管理、编排和调度的系统。（张建 等，2014）

综上所述，在当今大流量的云计算时代，应用的负载均衡已经成为了服务开发中十分重要的一环，要保证用户体验的同时对产品进行迭代是云计算时代开发或不可缺的能力，所以对应用负载均衡的优化是时代发展的迫切需求。

本设计研究了负载均衡技术并使用Kubernetes作为解决方案实现应用的高负载、高可用、平滑迭代和一键部署。本设计适应当今云计算时代的开发需求，十分具有商业价值。

# 第二章 Slyph框架设计

## 2.1需求设计

Slyph容器管理系统主要用于企业应用服务的开发、运维和部署并且实现应用的高高负载均衡。实现管理者可以通过界面实现对应用的生命周期管理、动态伸缩、升级回滚和一键部署。实现非离线式迭代和回滚。

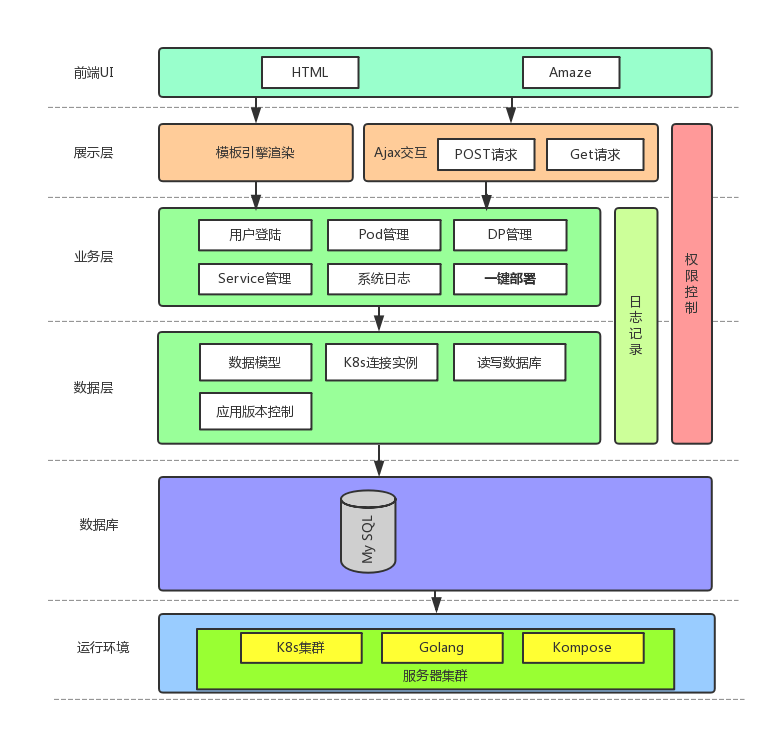


图2-1 Slyph系统架构图

### 2.1.1后端需求分析

1. 权限验证

A.登入权限：用户使用Slyph时权限验证。

B.界面访问：无权限用户访问Slyph返回错误界面。

2、错误处理

A.操作错误：执行错误操作会回显报错信息。

B.系统错误：系统出现错误会写入数据库，写入过程出错会打印到console。

3、K8s组件控制

A.Pod实例控制：控制Pod实例的查看、创建和删除。

B.Deployment实例控制：控制Deployment实例的查看、创建、删除、滚动更新和回滚。

C.Service实例控制：控制Service实例 的查看、创建和删除。

4、日志记录

A.登入日志：记录用户登入的名称和网络地址。

B.操作日志：记录用户对Slyph进行的操作。

5、一键部署

A.Up：编写/v1/auto/up/接口实现一键部署服务。

B.Down：编写/v1/auto/down/接口实现一键下线服务。

C、集成Swagger

Slyph集成swagger实现界面化API调用便于测试。

### 2.1.2 前端需求分析

1、登入界面和404界面

A.登入界面用户输入名称和密码登入。

B.用户直接访问Slyph资源则返回错误界面。

2、K8s组件控制界面

A.Pod实例的监控、下线和部署界面。

B.Deployment实例的监控、下线、部署和升级界面。

C.Service实例的监控、下线和部署界面。

3、一键部署界面

A.部署和下线界面。

B.应用的监控界面。

4、日志界面

日志界面显示Slyph系统日志和用户操作日志。

Slyph系统主要功能为：

A.用户登录

B.应用及其负载均衡的部署

C.一键式部署和下线

D.日志收集

## 2.2 环境及相关实例

Slyph系统服务依赖的运行环境如下：

Docker 1.10 or later

Kompose 1.12

Kubeadm 1.10.0

Etcd 3.0

K8s 1.09.0 or later

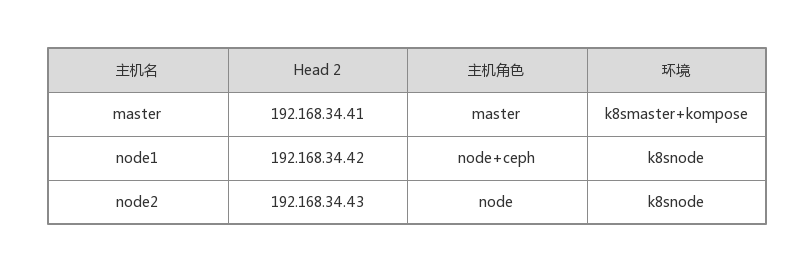


图2-2-1 集群环境

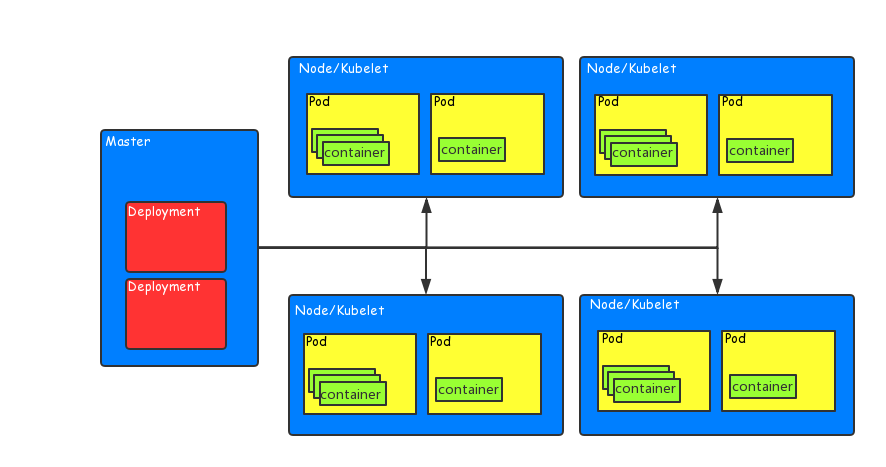


图2-2-2 Kubernetes架构

### 2.2.1 主节点

在服务器集群中，需要一台服务器作为Kubernetes集群的主服务器，负责调度从服务器Node，同时在本设计中，主服务器同时承担Kubernetes Master和Kompose执行者的角色所以需要安装Kubernetes Master和Kompose环境。

### 2.2.2 从节点

在服务器集群中，应用本身运行在Pod内，而Pod则运行在各个Node内，Node由Master控制并由Master根据资源情况将合适的Pod调度到Node中运行,Node节点上运行的服务包括Docker，kubelet和kube-proxy。

### 2.2.3 应用

Kubernetes中的Pod是其最基本的单位，Pod运行在节点上，Pod中包含一个或多个容器(Docker)，同一个Pod内的容器共享网络资源。（Butler，2016）在开发的过程中，我们实际的服务就是运行在这一个个Pod之中，Pod的生命周期是短暂的，实际开发中我们不会去单独启动一个Pod，而是利用Deployment去调度管理Pod。

### 2.2.4 应用管理器

Deployment在Kubernetes中的作用是控制和管理Pod。它能够根据你所做出的描述去创建对应数量的Pod和监控其是否健康并且保证Pod的健康数量。Kubernetes的实例之间是强解耦的，Deployment和Pod的绑定是通过Label绑定的。(杜军,2016)

### 2.2.5 负载均衡

Pod的生命周期是短暂的，因此Pod的虚拟IP是随着其创建和销毁变化的，所以我们不可能通过直接访问Pod的IP的方式去访问服务。Kubernetes中的Service是一种抽象的负载均衡，Service和Pod的关联也是通过Labels。（Asay，2016）当Labels指定了对应的Pod，Service会为每个Pod生成对应的Endpoint，在访问Service的时候，Service会为你自动路由到对应的Pod。(唐瑞，2017)

### 2.2.6 有状态应用

Kubernetes中的Pod时无状态的，在实际的开发中往往需要将应用使用中的数据保存下来，Statefulset类似于Deployment，其功能为对Pod调度和控制，同时，它还支持通过配置第三方挂在卷将应用中的数据持久化到数据中心如NFS服务器中。

## 2.3技术栈

HTML5：HTML5是HTML最新的版本，其制定的目的是取代HTML 4.01和XHTML 1.0标准，HTML5减少了应用对插件的依赖并且拥有良好的跨平台性。（黄永慧 等，2013）

Go：Go语言由谷歌开发和维护，其高并发性能使其十分适用与网站开发。Go语言有自己的编码排版工具，使得代码维护变得十分轻松。

jQuery：jQuery是一个强大的JavaScript类库，拥有兼容多浏览器、免费开源、快速、小型、功能丰富等特点。

Amaze UI：Amaze UI前端框架包含多种组件，这种基于组件式的开发是的开发者不用深入了解前端技术就可以对应用UI进行开发。

Ajax技术：Ajax实现了用户在界面上产生事件后无需等待事件的完成，事件将会由JavaScript脚本异步处理，然后将处理结果返回给前端界面。

# 第三章 Slyph后端设计

## 3.1 MVC设计

Controller层，根据所实现的功能，设计出对应的Controller对用户的请求信息进行校验和返回处理后的信息。

Model层根据Controller需求设计出对应的结构体和结构体成员，并在models中实现Controller的逻辑代码如deployment的创建、删除、滚动更新和回滚。

View层为了方便用户使用，根据服务类型列出基本菜单，在每个服务下根据功能列出子菜单，资源查看界面通过Ajax请求后端对应接口返回的JSON数据在前端进行渲染。

## 3.2脚本文件编写

### 3.2.1 Dockerfile文件

Dockerfile是一个文本，其中包括了一系列的指令，每一条指令构建一层并描述怎样构建该层，Docker将按序执行操作创建镜像。

FROM registry.docker-cn.com/library/golang:1.8

WORKDIR /go/src/k8s

COPY . /go/src/k8s

ENV KUBERNETES\_MASTER http://192.168.34.41:8080

ENV TZ Asia/Shanghai

MAINTAINER haoyun

EXPOSE 8888

CMD ["Slyph"]

### 3.2.2 Makefile文件

Makefile是一系列shell语句的集合。编写好Makefile后只需要使用make [target] 指定所要执行的步骤即可完成自动化编译。make all 执行程序的编译、构建和Docker镜像的打包过程。

#! /usr/bin/make -f

targetImagename = slyph

.PHONY: all

all: package

.PHONY: test

test: build

.PHONY: package

package: build createImage

.PHONY: build

build:

go build -o Slyph main.go

.PHONY: createImage

createImage:

docker build -t $(targetImagename) .

.PHONY: clean

clean:

rm -f Slyph

docker rmi $(docker images|grep slyph|awk '{print $1}')

# 第四章 Slyph前端实现

## 4.1 框架选择

近几年来前端发展的十分迅速，可以说是百家争鸣。在类库方面，jquery无疑是其中的霸主，在框架方面近几年比较热门的框架有Vue、React和Angular这三种。本项目对以下几种流行的框架进行了对比筛选：

（1）AntDesign：AntDesign是由阿里开发的一款基于React的UI库，其丰富的组件和官方示例让它在企业级应用开发中十分热门，但是其要求对前端具有深刻的理解和丰富的知识储备，入门门槛较高。考虑到所需要学习的组件和语法太多，并没有相应的需求，所以不予采用。

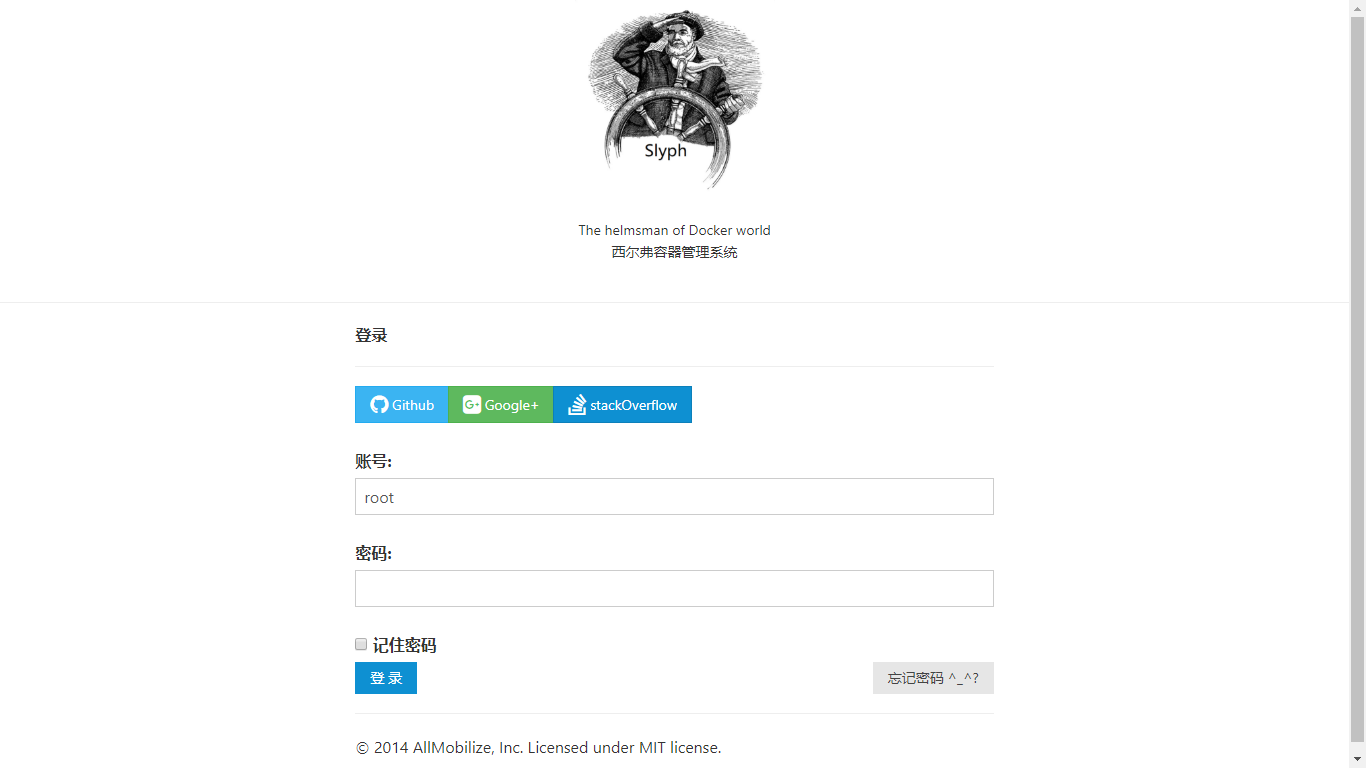
（2) AmazeUI：Amaze UI前端框架包含多种组件，这种基于组件式的开发是的开发者不用深入了解前端技术就可以对应用UI进行开发，因此采用AmazeUI。

## 4.2前端界面设计

前端界面是直接与用户交互的地方，界面设计需要让用户能明确的区分每个模块的区别与联系，让界面处在用户的掌控之中，用户可以直接通过界面操作应用，同时要对用户的操作及时做出反馈提高用户体验。

本系统前端的首要需求是便于用户对应用实例的操作和监控，前端以用户为中心，界面结构切合用户操作习惯。

系统界面分为菜单栏和操作区域，登入界面如图4-1所示，主页界面如图4-2所示。



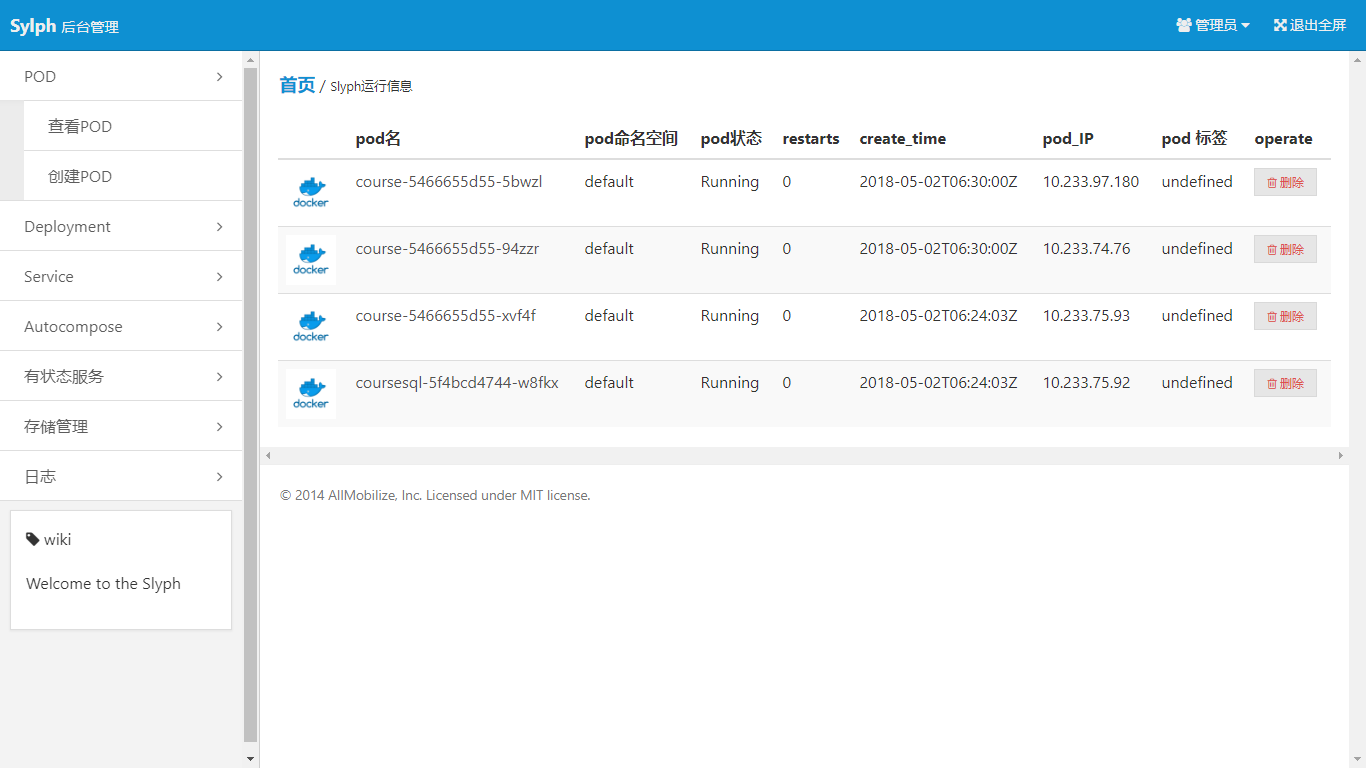
图4-1 登入界面

图4-2主页

用户在登入后会跳转到主页，主页默认显示的是对POD资源的查看，用户可以通过左侧的菜单选择对POD等资源的查看与创建，每个模块所对应的功能包含在其二级菜单内，对应界面如图4-3、4-4、4-5、4-6、4-7、4-8所示。

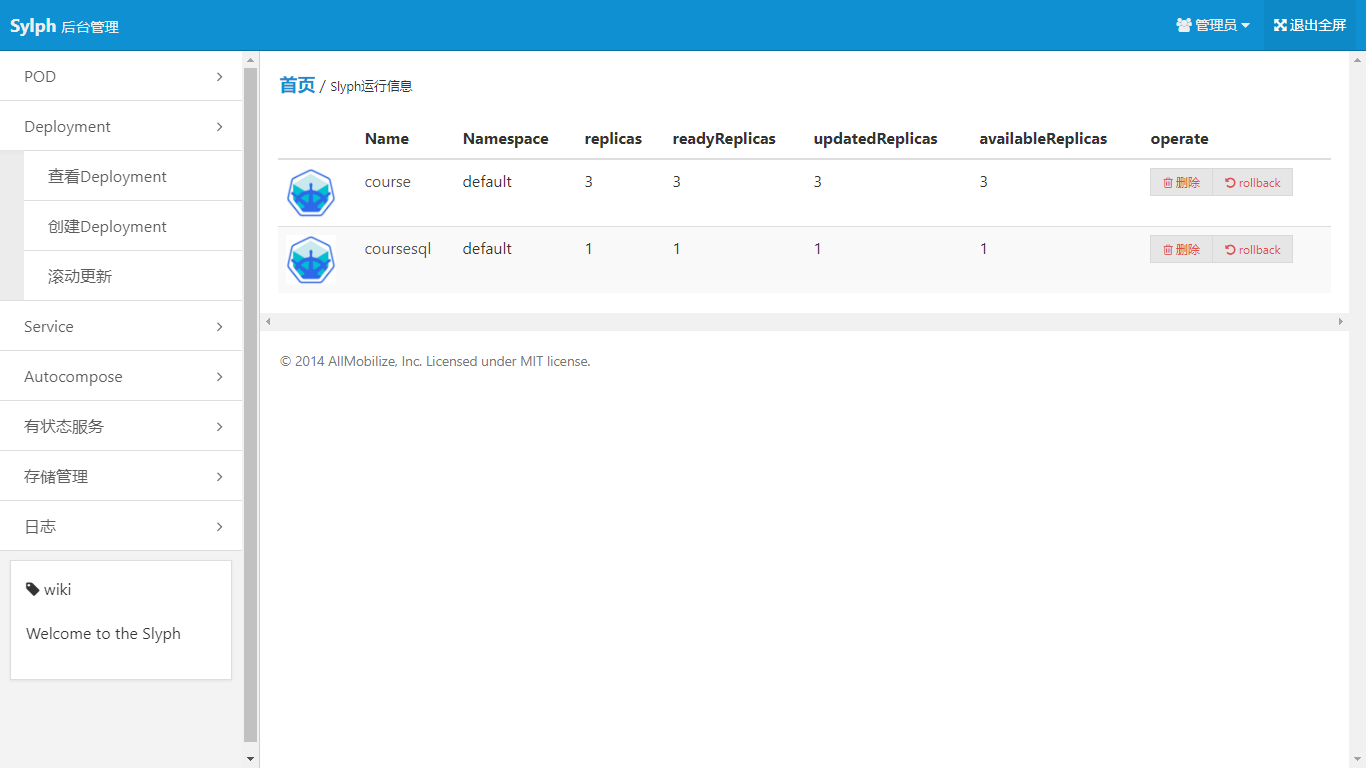


图4-3 Deployment操作界面

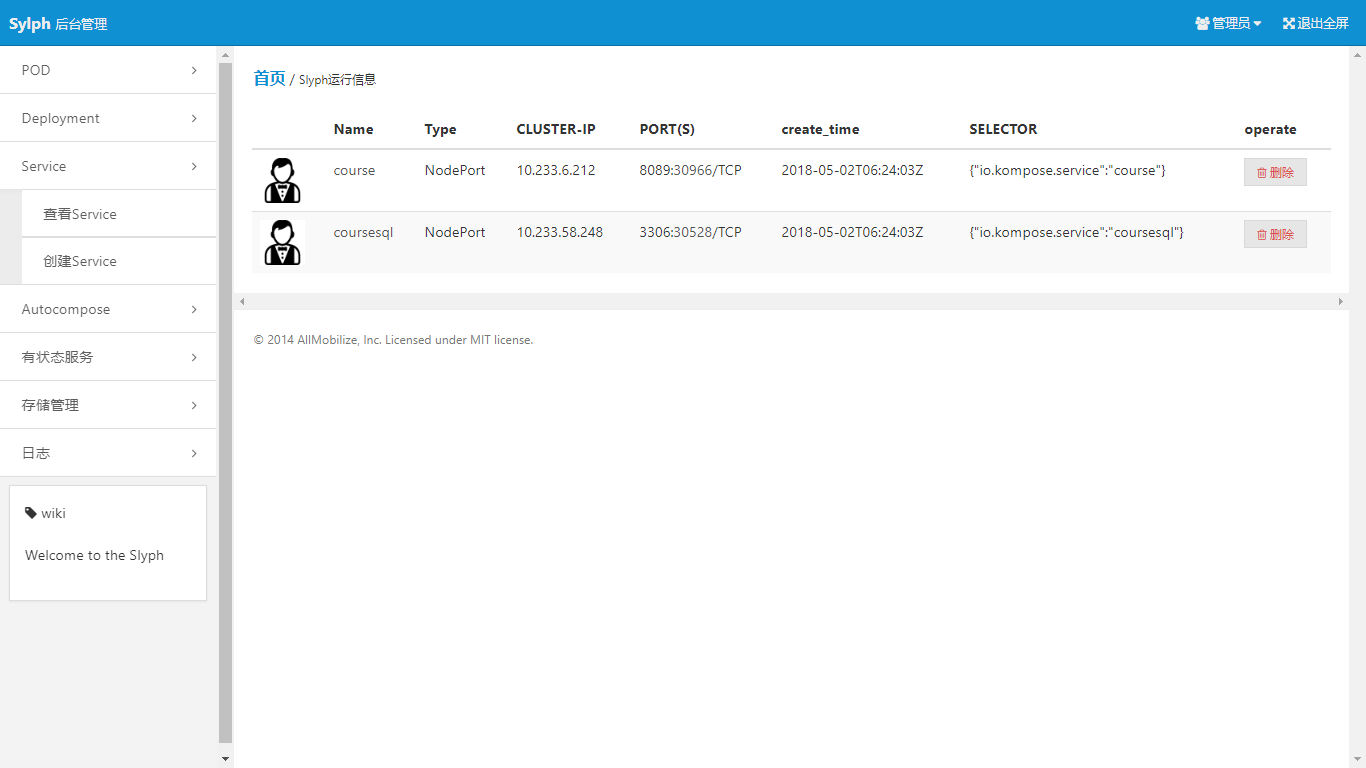


图4-4 Service操作界面

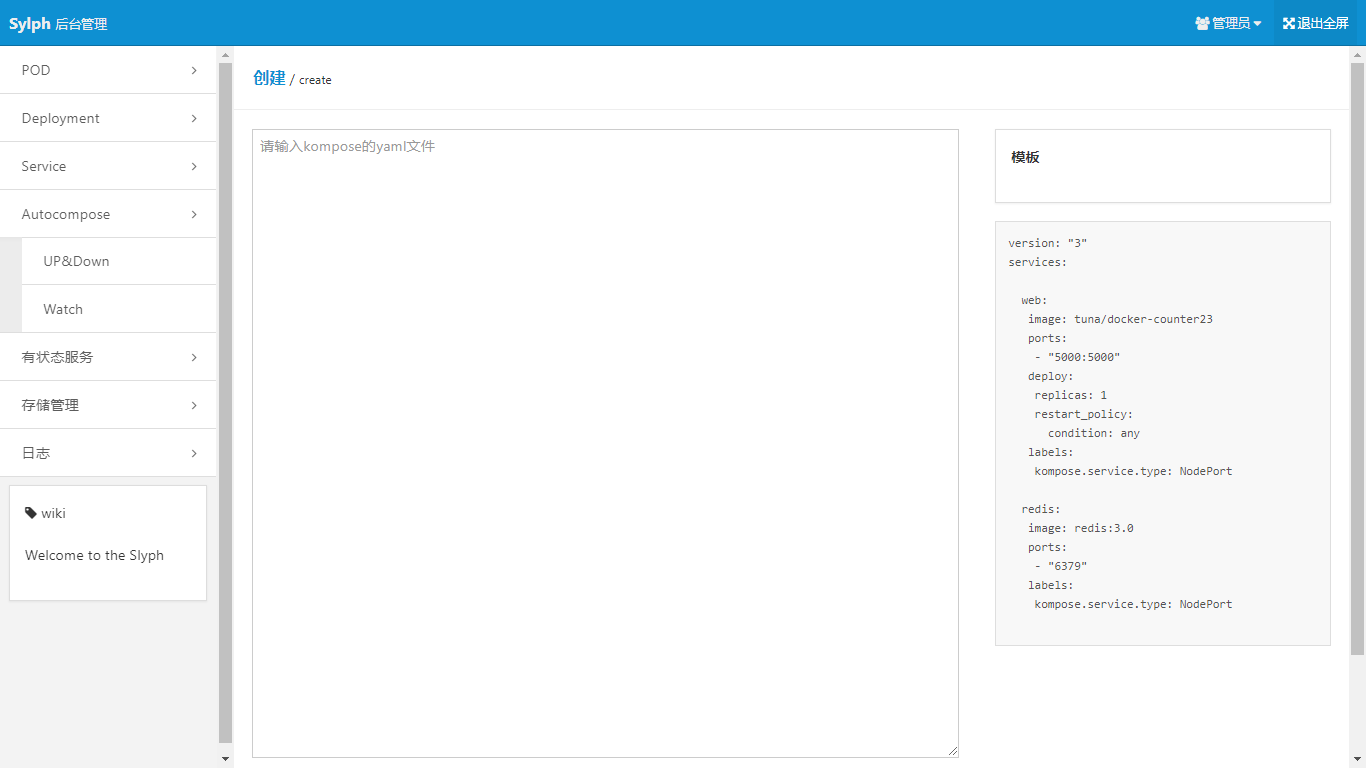


图4-5 一键部署操作界面

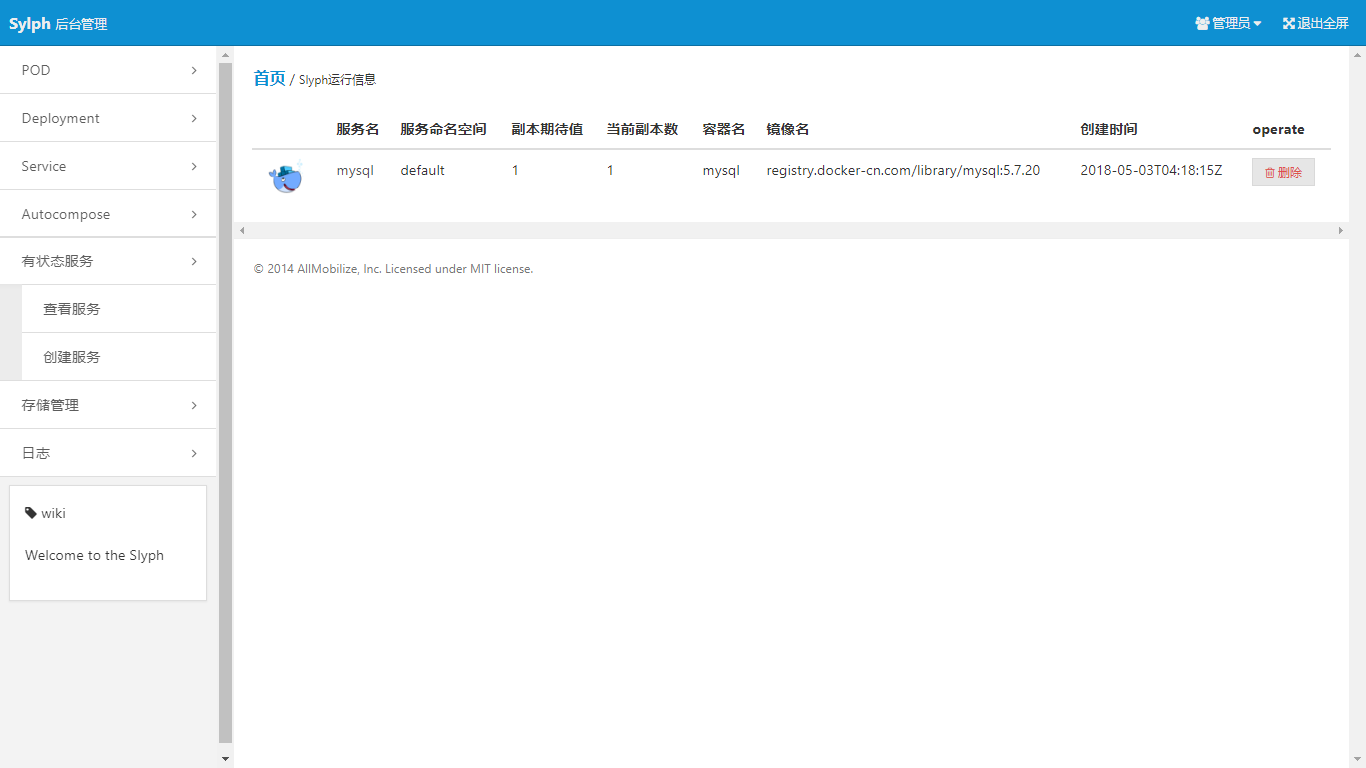


图4-6有状态应用操作界面

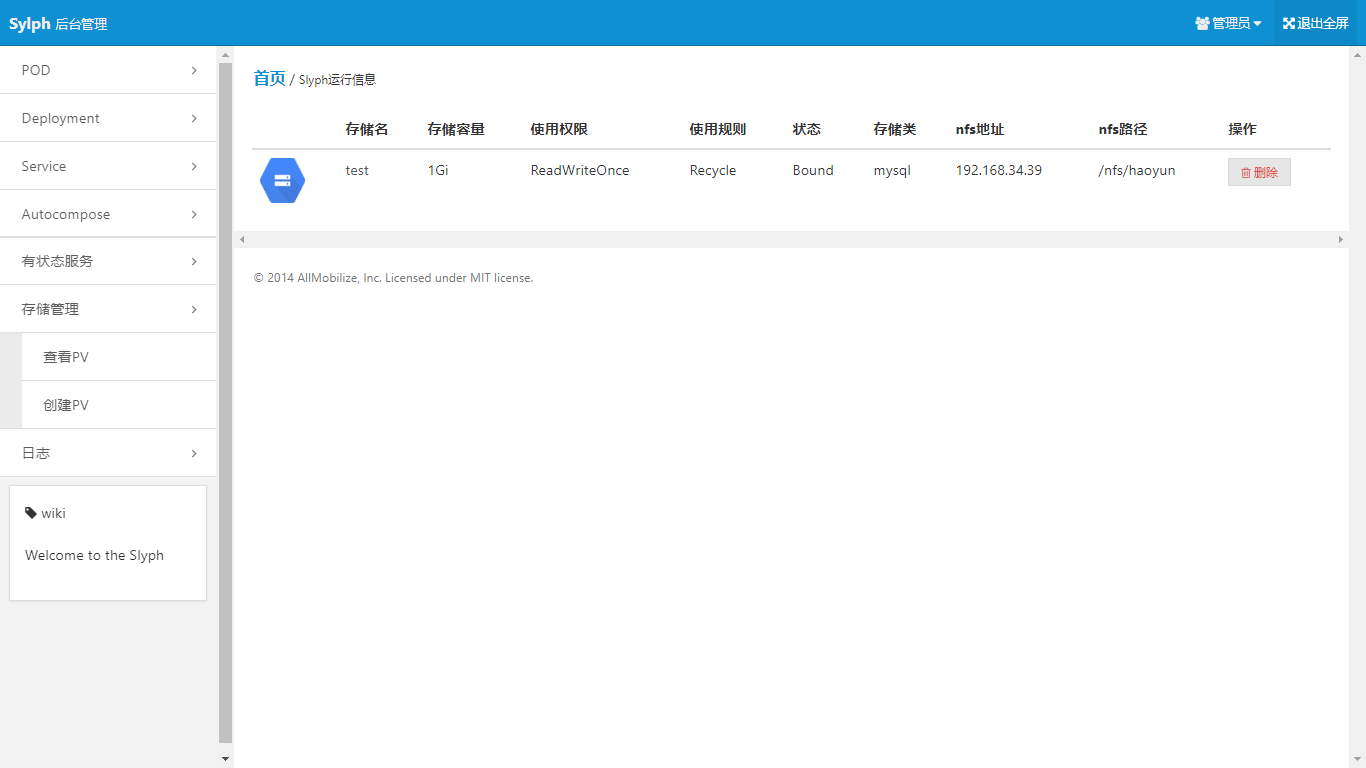


图4-7存储管理界面

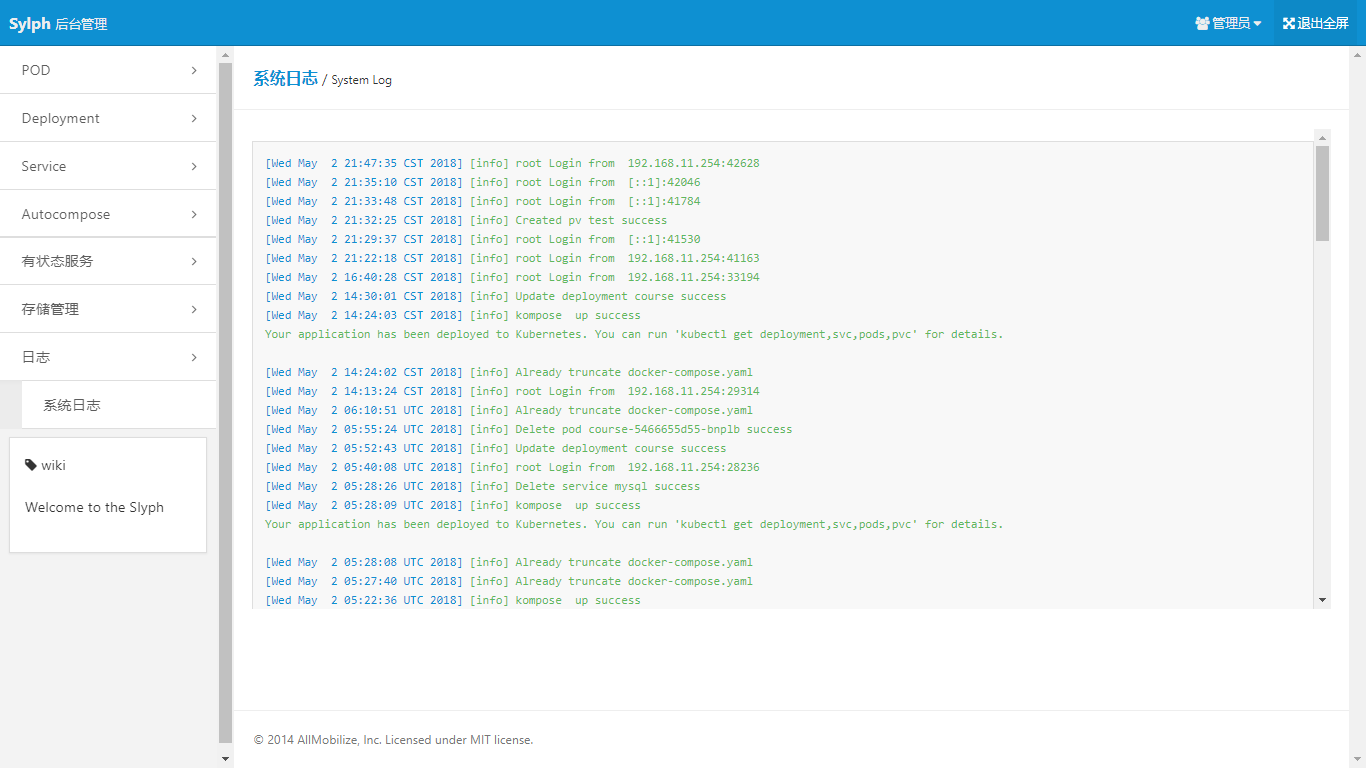


图4-8日志界面

所有前端界面的内容都是通过Ajax异步请求后端接口获取的，资源的查看是调用后端的getjson接口，前端JS代码将获取的JSON数据渲染成对应的列表，资源的其他操作均是通过Ajax将页面表单内容转为JSON数据传送给后端相应接口并回显返回信息。

## 4.3功能模块实现

### 4.3.1 管理员登入

登录流程：

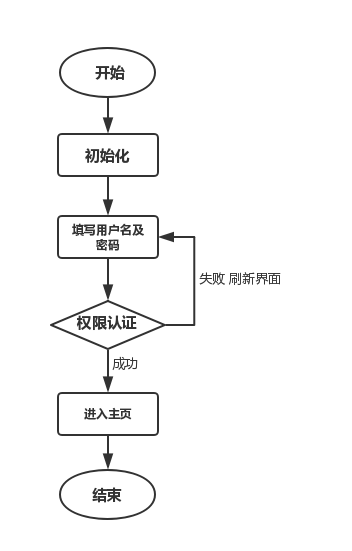
 用户进入登录页面，填写账户名称和密码，点击登录后表单将传输给后端接口，后端将用户名对应的密码和数据库中的密码进行对比，如果密码一致则跳转到主页否则刷新登入界面。登录流程图如图4-9所示。

图4-9登录流程图

登录功能的主要代码如下：

func (main \*LoginController) Post() {

log := models.Log{}

name := main.GetString("username") //从表单中获取用户名

password := main.GetString("password") //从表单中获取密码

var pass string

db, err := models.LinkTomysql()

defer db.Close()

if err != nil {

log.Time = models.Gettime()

log.Etype = "error"

log.Event = "Failed to link mysql:" + err.Error()

go models.Insertlog(&log) //向数据库插入日志

} else {

rows, err := db.Query("select password from user where name=? ", name)

for rows.Next() {

rows.Scan(&pass)

}

defer rows.Close()

if err != nil {

log.Time = models.Gettime()

log.Etype = "error"

log.Event = name + " failed to get password ip:" + main.Ctx.Request.RemoteAddr

go models.Insertlog(&log)

main.Redirect("/v1/user/login", 302)

} else if pass == password {

log.Time = models.Gettime()

log.Etype = "info"

log.Event = name + " Login from " + main.Ctx.Request.RemoteAddr

go models.Insertlog(&log)

main.SetSession("username", name)

main.Redirect("/v1/index/", 302)

} else {

log.Time = models.Gettime()

log.Etype = "error"

log.Event = name + " login with wrong password ip:" + main.Ctx.Request.RemoteAddr

go models.Insertlog(&log)

main.Redirect("/v1/user/login", 302)

}

}

}

用户名和密码都匹配，则登录成功进入主页，可查看和管理资源，若密码或用户名错误，则后端会向数据库插入数据记录下登入的用户名和登入者的IP。

用户登录成功后，后端会将用户名写入session做权限控制，session的过期时间为6分钟，若长时间未操作需要重新登入。

### 4.3.2 POD的显示与管理

POD操作流程：

用户在左侧菜单栏选择POD即可弹出POD操作的二级菜单。用户选择查看POD功能时前端会调用后端的/v1/pod/getjson接口，并将获取到的JSON数据渲染成列表显示在前端。用户选择创建POD功能时，前端会将表单数据传送给/v1/pod/ceate接口创建POD并回显返回数据。用户点击删除对应POD时，前端会调用/v1/pod/delete接口删除POD并弹窗提醒是否删除成功,然后刷新页面。

POD操作流程图如图4-10所示。

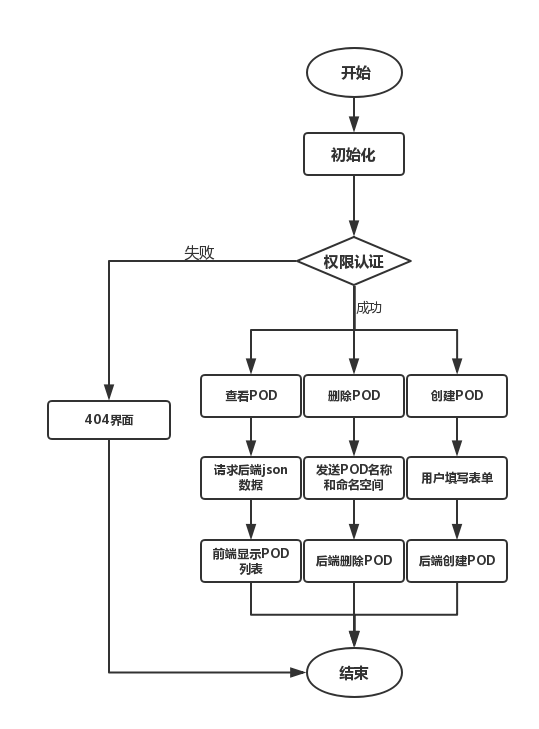


图4-1 POD操作流程图

POD功能核心代码如下所示。

if \_, err := models.GetClientset().CoreV1().Pods(req.Namespace).Create(&pod); err != nil {

log.Time = models.Gettime()

log.Etype = "error"

log.Event = "Failed to create pod " + req.Name + " " + err.Error()

go models.Insertlog(&log)

return http.StatusInternalServerError, err

}

if err := models.GetClientset().CoreV1().Pods(req.Namespace).Delete(req.Name, &metav1.DeleteOptions{}); err != nil {

log.Time = models.Gettime()

log.Etype = "error"

log.Event = "Failed to delete pod " + req.Name + " " + err.Error()

models.Insertlog(log)

return http.StatusInternalServerError, err

}

pods, err := models.GetClientset().CoreV1().Pods("default").List(metav1.ListOptions{})

if err != nil {

log.Time = models.Gettime()

log.Etype = "error"

log.Event = "Get pod error " + err.Error()

models.Insertlog(log)

return nil

}

后端在controller层做表单校验和权限控制，校验完成后会调用model层的实际逻辑代码，代码调用Kubernetets client的API对POD资源进行查询和管理并且将执行的操作或操作中产生的错误记录在数据库当中。

### 4.3.3 Deployment的显示与管理

Deployment操作流程：

用户在左侧菜单栏选择Deployment即可弹出Deployment操作的二级菜单。用户可通过选择不同的功能对Deployment进行查看、创建和滚动更新。当用户选择查看Deploy ment功能时前端会调用后端的/v1/deployment/getjson接口，并将获取到的JSON数据渲染成列表显示在前端。用户选择创建Deployment功能时，前端会将表单数据传送给/v1/deployment/ceate接口创建Deployment并回显返回数据。用户点击删除对应Dep loyment时，前端会调用/v1/deployment/delete接口删除Deployment并弹窗提醒是否删除成功,然后刷新页面。

Deployment操作流程图如图4-11所示。

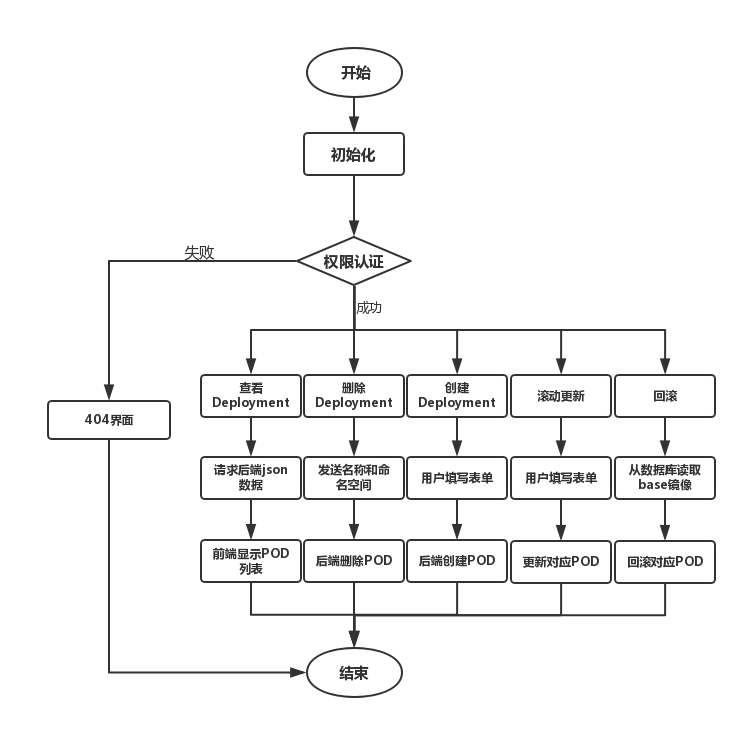


图4-11Deployment操作流程图

Deployment功能核心代码如下所示。

deploymentsClient:=models.GetClientset().AppsV1beta1().Deployments(req.Namespace)

result, err := deploymentsClient.Create(deployment)

if err != nil {

log.Time = models.Gettime()

log.Etype = "error"

log.Event = "Failed to create deployment " + result.GetObjectMeta().GetName() + " " + err.Error()

go models.Insertlog(&log)

return http.StatusInternalServerError, err

}

models.Insertbaseimage(req.Name, req.Containers[0].Image, "base")

go models.Insertbaseimage(req.Name, req.Containers[0].Image, "new")

log.Time = models.Gettime()

log.Etype = "info"

log.Event = "Created deployment " + result.GetObjectMeta().GetName() + " success"

go models.Insertlog(&log)

后端在controller层做表单校验和权限控制，校验完成后会调用model层的实际逻辑代码，代码调用Kubernetets client的API对Deployment资源进行查询和管理并且将执行的操作或操作中产生的错误记录在数据库当中。

### 4.3.4 Service的显示与管理

Service操作流程：

用户在左侧菜单栏选择Service即可弹出Service操作的二级菜单。用户选择查看Service功能时前端会调用后端的/v1/service/getjson接口，并将获取到的JSON数据渲染成列表显示在前端。用户选择创建Service功能时，前端会将表单数据传送给/v1/service/ceate接口创建Service并回显返回数据。用户点击删除对应Service时，前端会调用/v1/service/delete接口删除Service并弹窗提醒是否删除成功,然后刷新页面。

Service操作流程图如图4-12所示。

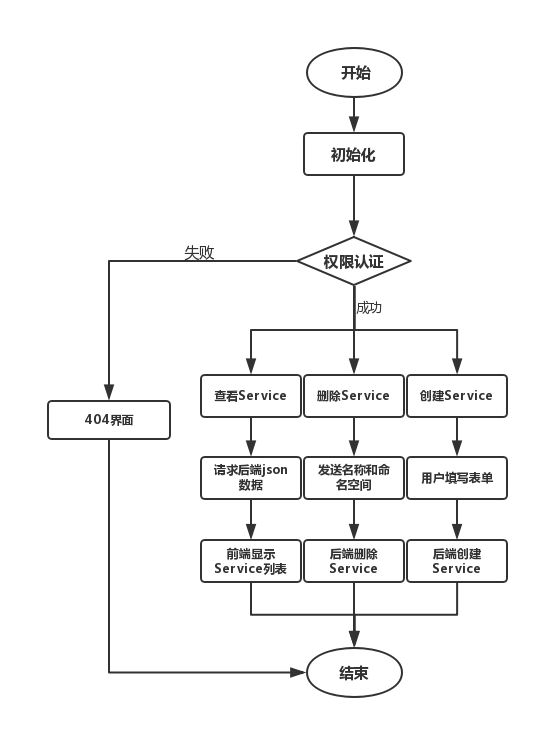


图4-12 Service操作流程图

Service功能核心代码如下所示。

if\_,err :=models.GetClientset().CoreV1().Services(service.Namespace).Create(&service); err != nil {

log.Time = models.Gettime()

log.Etype = "error"

log.Event = "Created service " + req.Name + " error"

go models.Insertlog(&log)

return http.StatusInternalServerError, err

}

if serviceimp.Name != " " {if err := models.GetClientset().CoreV1().Services(req.Namespace).Delete(req.Name, &metav1.DeleteOptions{}); err != nil {

log.Time = models.Gettime()

log.Etype = "error"

log.Event = "Failed to delete service " + req.Name + " " + err.Error()

go models.Insertlog(&log)

return http.StatusInternalServerError, err

}

}

services,err:=models.GetClientset().CoreV1().Services("default").List(metav1.ListOptions{})

后端在controller层做表单校验和权限控制，校验完成后会调用model层的实际逻辑代码，代码调用Kubernetets client的API对Service资源进行查询和管理并且将执行的操作或操作中产生的错误记录在数据库当中。

### 4.3.5有状态应用的显示与管理

有状态应用操作流程：

用户在左侧菜单栏选择有状态服务即可弹出有状态应用操作的二级菜单。用户选择查看有状态服务功能时前端会调用后端的/v1/st/getjson接口，并将获取到的JSON数据渲染成列表显示在前端。用户选择创建有状态服务功能时，系统会首先检测是否已创建好存储类，若没有则会跳转到创建存储界面，若有则会在存储表单填写好已有的存储类供用户选择。前端会将表单数据传送给/v1/st/ceate接口创建有状态服务并回显返回数据。用户点击删除对应有状态服务时，前端会调用/v1/st/delete接口删除对应服务并弹窗提醒是否删除成功,然后刷新页面。

有状态应用操作流程图如图4-13所示。

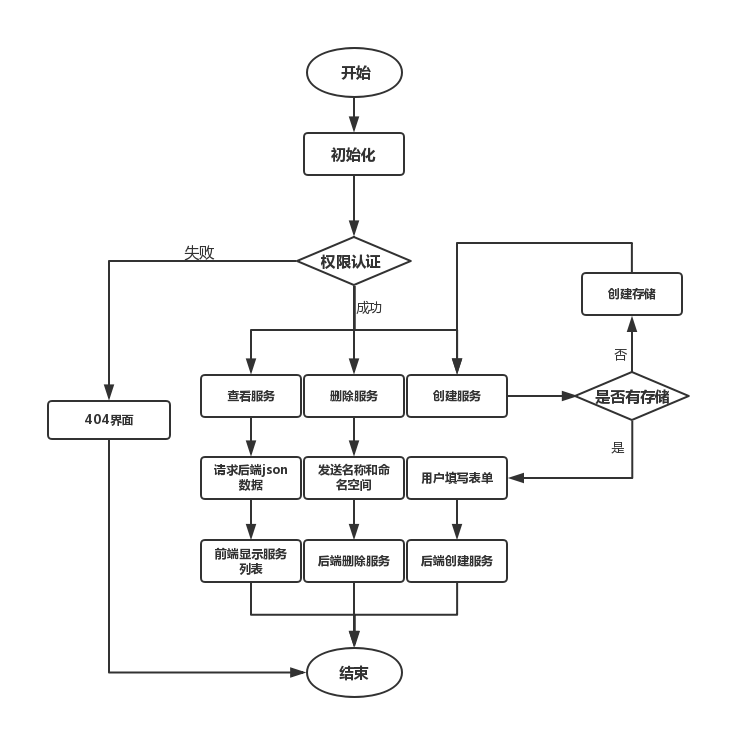


图4-13 有状态应用操作流程图

有状态服务功能核心代码如下所示。

if \_, err := models.GetClientset().AppsV1beta2().StatefulSets(req.Namespace).Create(&st); err != nil {

log.Time = models.Gettime()

log.Etype = "error"

log.Event = "Created StatefulSets " + req.Name + " error"

go models.Insertlog(&log)

return http.StatusInternalServerError, err

}

if stimp.Name != " " {

if err := models.GetClientset().AppsV1beta2().StatefulSets(req.Namespace).Delete(req.Name, &metav1.DeleteOptions{}); err != nil {

log.Time = models.Gettime()

log.Etype = "error"

log.Event = "Failed to delete Statefulset " + req.Name + " " + err.Error()

go models.Insertlog(&log)

return http.StatusInternalServerError, err

}

}

sts, err := models.GetClientset().AppsV1beta2().StatefulSets("default").List(metav1.ListOptions{})

后端在controller层做表单校验和权限控制，校验完成后会调用model层的实际逻辑代码，代码调用Kubernetets client的API对Statefulset资源进行查询和管理并且将执行的操作或操作中产生的错误记录在数据库当中，在创建有状态服务的时候会创建对应存储PV的PVC，在删除有状态服务的时候会自动将PVC删除。

### 4.3.6存储的显示与管理

存储管理的操作流程：

用户在左侧菜单栏选择存储管理即可弹出存储管理操作的二级菜单。用户选择查看存储功能时前端会调用后端的/v1/sto/getjson接口，并将获取到的JSON数据渲染成列表显示在前端。用户选择创建存储功能时，前端会将表单数据传送给/v1/sto/ceate接口创建存储并回显返回数据。用户点击删除对应存储时，前端会调用/v1/sto/delete接口删除对应存储并弹窗提醒是否删除成功,然后刷新页面。

存储管理操作流程图如图4-14所示。

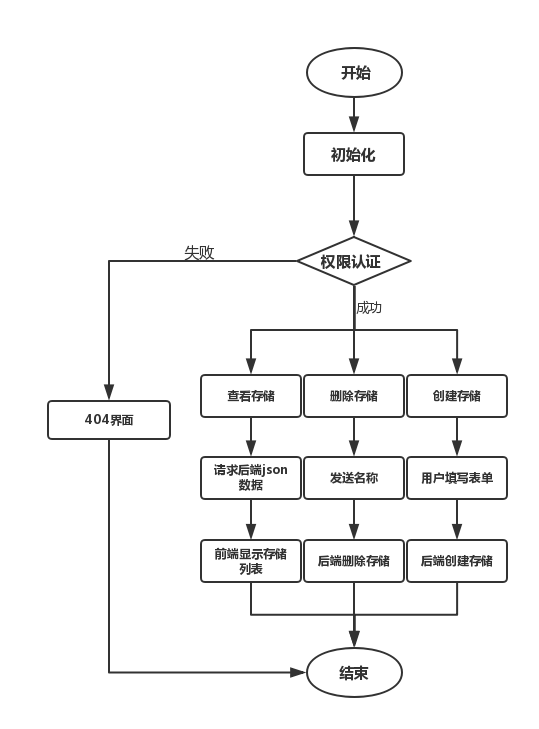


图4-14 存储管理操作流程图

目前系统存储支持的类型为NFS，用户在使用存储功能之前需要搭建好相应的NFS服务器，并手动创建存储文件夹如/nfs/mysql，然后在创建存储PV的时候填写上对应的NFS服务地址和路径。

核心代码如下所示。

if \_, err = client.CoreV1().PersistentVolumeClaims(req.Namespace).Create(pvc); err != nil {

log.Time = Gettime()

log.Etype = "error"

log.Event = "Failed to create PVC " + req.Name + " " + err.Error()

go Insertlog(&log)

return http.StatusInternalServerError, err

}

if err := client.CoreV1().PersistentVolumeClaims(req.Namespace).Delete(req.Name, &metav1.DeleteOptions{}); err != nil {

log.Time = Gettime()

log.Etype = "error"

log.Event = "Failed to Delete for PVC " + req.Name + " " + err.Error()

go Insertlog(&log)

return http.StatusInternalServerError, err

}

if pvcs, err = client.CoreV1().PersistentVolumeClaims("default").List(metav1.ListOptions{}); err != nil {

log.Time = Gettime()

log.Etype = "error"

log.Event = "Failed to Get list of PVC " + " " + err.Error()

go Insertlog(&log)

return nil

}

后端在删除存储后并不会清理对应的NFS文件，这样做是为了数据安全，如果用户想恢复数据是需要创建相同NFS路径的PV即可，若用户需要清理数据需要手动清理NFS对应文件夹。

### 4.3.7一键部署

一键部署功能是本系统的一个重要组成，在应用开发的时候，自主研发的应用经常需要依赖于各种已有组件，例如：开发一个web项目需要依赖于类似于mysql等的数据库；微服务开发过程中，为服务应用需要依赖于SpringCloud的组件如Eureka、Hystrix和ConfigServer。一键部署功能让你只需要关注依赖服务配置信息，在前对输入对应的yaml文件系统就会帮用户编排对应的服务。

一键部署的操作流程：

用户在左侧菜单栏选择Autocompose即可弹出一键部署操作的二级菜单。用户选择UP&DOWN后在右侧输入框填入对应的yaml文件然后再点击下方的UP按钮即可一键部署应用，点击Down按钮会删除对应服务，用户选择Watch菜单页面将显示已部署的服务。

一键部署功能核心代码如下所示。

cmd := exec.Command("kompose", "up")

cmd := exec.Command("kompose", "down")

f, err1 = os.Create(filename)

\_, err1 = io.WriteString(f, string(requestbody))

后端会接收用户输入的yaml文件内容并在宿主机上创建出名为docker-compose的yaml文件，之后使用kompose up命令或kompose down命令实现应用的一键部署和删除，在系统运行的Docker中需要声明环境变量KUBERNETES\_MASTER的值为Kubernetes集群apisever的地址。

一键部署流程图如图4-15所示。

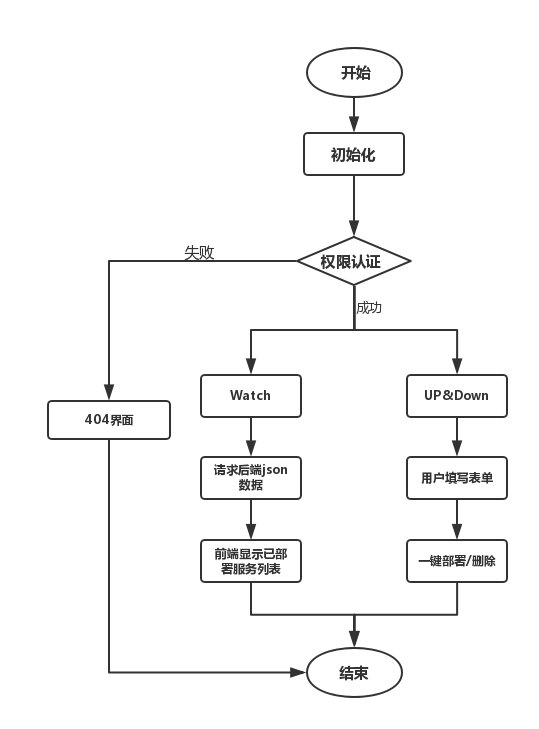


图4-15 一键部署操作流程图

### 4.3.8日志

日志功能记录用户操作的返回值和操作执行时间到数据库中。

日志功能的操作流程：

用户通过选择系统日志功能触发前端调用后端的/v1/getlog/log接口，前端会将获取到的JSON数据渲染成列表显示在系统日志显示框内。

核心代码如下所示。

func Readlog() []Log {

var logs []Log

o := Getorg()

num, err := o.Raw("SELECT time, etype, event FROM log ORDER BY time desc LIMIT 100").QueryRows(&logs)

if err != nil {

fmt.Println(num, err)

}

return logs

}

func Insertlog(log \*Log) {

o := Getorg()

id, err := o.Insert(log)

if err != nil {

fmt.Println(id, err)

}

}

日志查询功能流程图如图4-16所示。

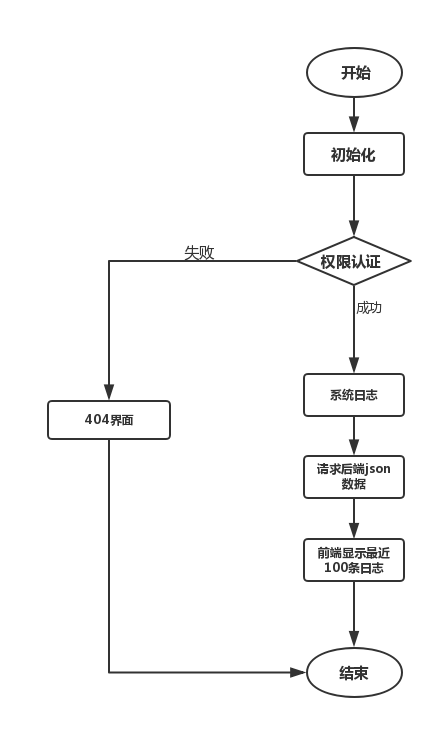


图4-16 日志查询功能流程图

# 第五章 Slyph数据库设计

## 5.1数据库的对比与选择

Redis，Mysql，HBASE，Oracle数据库简而言之就是电子化的文件仓库，用户可以对文件中的数据进行增加、删除、查询、更新等操作。比较普遍使用的数据库有Mysql、Sqlite、Hbase和Redis，本论文将对以上四种数据库进行对比并选择出最适合本次设计的数据库，四种数据库对比如下所示。

（1）MySQL：免费并且广泛应用在中小型网站中。同时Mysql对Docker的支持也十分良好。（吴坤安 等，2017）

（2）Sqlite: Sqlite基于内存并且十分便于迁移，但是其并不支持多客户端同时写入。

（3）Hbase：Hbase的表结构是树形的，其中的每一个值需要表名、列族名、行和变量名对应。但是Hdfs不支持条件查询，对系统资源也有较高要求，使用时需要根据需求声明大量配置信息。

（4）Redis：Redis基于内存的特点使其在读写方面的执行效率非常高但是基于内存的数据库管理系统存储的数据量与服务器本身的内存大小相关。

这几款数据库中Mysql对Golang的语言支持良好而且已经实现Docker化，并且Mysql操作简便已经可以满足本项目的需求，所以在本项目中将使用Mysql作为数据库管理系统。

## 5.2 部署Mysql容器

服务器部署mysql的步骤:

第一步：使用命令 docker pull mysql:5.7.20拉取Mysql镜像；

第二步：使用命令 docker run –it –p 3306:3306 –e MYSQL\_ROOT\_PASSWORD password -d mysql:5.7.20启动Mysql；

## 5.3 数据库表结构设计

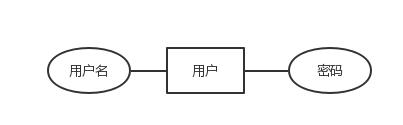
本系统中数据库的作用主要是存储管理员相关信息、部署应用的镜像信息和管理员对平台操作时所产生的日志信息因此设计三张表去存储相关信息。一个良好的数据库设计能够减少数据冗余和操作异常等问题，对节约存储空间和增强并发性能有很大帮助。本系统所需数据库实体的E-R图如图5-3、5-4、5-5所示。

图5-3用户实体E-R图

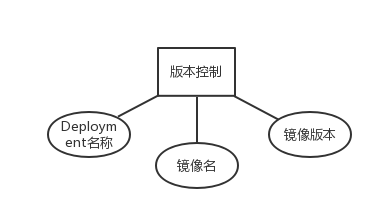


图5-4版本控制实体E-R图

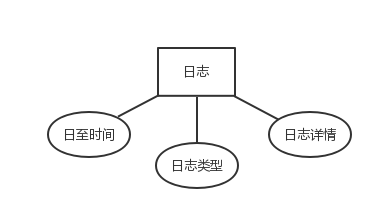


图5-5日志实体E-R图

表创建代码如下：

CREATE TABLE `user` (

`id` int(5) NOT NULL AUTO\_INCREMENT,

`name` varchar(20) NOT NULL,

`password` varchar(20) NOT NULL,

PRIMARY KEY (`id`)

) ENGINE=InnoDB AUTO\_INCREMENT=2 DEFAULT CHARSET=UTF-8;

CREATE TABLE `dep` (

`id` int(5) NOT NULL AUTO\_INCREMENT,

`dep` varchar(50) NOT NULL,

`image` varchar(50) NOT NULL,

`ver` varchar(10) NOT NULL,

PRIMARY KEY (`id`)

) ENGINE=InnoDB AUTO\_INCREMENT=2 DEFAULT CHARSET=UTF-8;

CREATE TABLE `logs`(

`time` varchar(20) NOT NULL,

`etype` varchar(10) NOT NULL,

`event` varchar(50) NOT NULL,

PRIMARY KEY (`time`)

) ENGINE=InnoDB AUTO\_INCREMENT=2 DEFAULT CHARSET=UTF-8;

## 5.4 Mysql的连接配置

Mysql连接配置核心代码如下：

db,err:=sql.Open("mysql","root:password@tcp(host)/k8s?charset=utf8")

if err != nil {

glog.Infoln("Fail to connect")

}

defer db.Close()

连接代码中的host为运行Mysql镜像的服务器地址，端口默认为3306，password为运行Mysql时的环境变量MYSQL\_ROOT\_PASSWORD的值，为了防止中文乱码，所以设置了连接编码为UTF-8，每次函数执行完毕后都会调用db.Close()。

# 第六章 系统测试

为检测系统是否对应用的负载均衡做到优化，编写了一个选课系统的测试demo其功能包含用户访问、用户登入、用户注册、学生选课、学生退课、老师开课和老师删除课程的功能，数据库采用Mysql。

利用Slyph的一键部署功能将选课系统demo部署到Slyph中并通过浏览器访问demo的负载均衡进行测试。测试使用WebBench工具模拟1k-14k客户端同时访问demo。

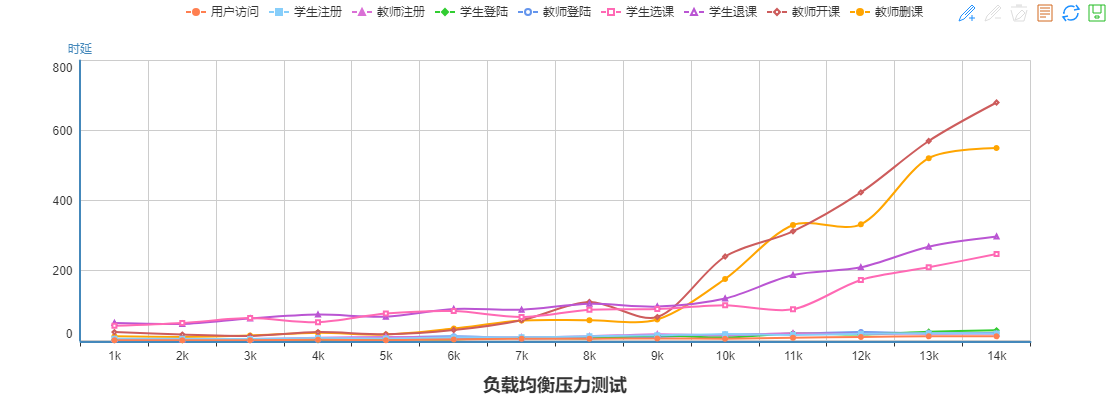
测试结果如图6-1所示：

图6-1测试结果

# 第七章 总结

本设计使用GO语言开发，使用Kubernetes的解决方案，实现了管理员Web端操作Docker化应用的编排、部署、弹性伸缩、下线、滚动更新和回滚的功能，同时应用的负载均衡也得到了显著的提升，管理员可以根据应用的压力预期值在不影响用户体验的情况下动态的调整应用的副本数量以适应不同的负载压力。本设计对服务器资源的利用也达到了最大化，在云计算时代下，对应用的高负载和非离线式更新的需求越来越明显。设计的当前版本的单管理员的管理方式不适应企业级开发，仍需要根据企业需求实现多用户和权限认证功能。

通过对Docker的学习让我发现了应用的开发、交付和运维可以变得如此便捷，通过对Kubernetes源码的学习和对其组件的调用让我更加深刻地理解了容器编排的魅力，在设计过程中，对自己不熟悉的前端知识的学习扩展了我的知识面，对MVC架构和Go project都有了实践性的宝贵经验。

# 致谢

通过不断地努力，本次的毕业设计终于顺利完成，在设计过程中遇到了很多的问题，除了通过自查资料以外，还要感谢我的毕业设计指导老师，覃章建老师。期间许多的问题都由覃老师帮我指出，并告诉我解决的方法，尤其是在论文撰写的时候，由于缺乏经验，很多地方自己考虑的都并不周到，覃老师经常督促我们，时常让我们将论文发给他，帮我们检查并提出修改建议。同时，社区的许多前辈也给予了我很多的帮助，让我对Kubernetes有了更深入的理解。最后感谢所有教育过我的老师，在你们所教授的知识的帮助下我完成了这次的设计。

# 参考文献

参考文献：

杜军. 基于Kubernetes的云端资源调度器改进[D].浙江大学,2016.

唐瑞. 基于Kubernetes的容器云平台资源调度策略研究[D].电子科技大学,2017.

王舜. 基于最小迁移代价的云计算资源负载均衡调度算法研究[D].电子科技大学,2013.

周佳威. Kubernetes跨集群管理的设计与实现[D].浙江大学,2017.

仇臣. Docker容器的性能监控和日志服务的设计与实现[D].浙江大学,2016.

何松林. 基于Docker的资源预调度策略构建弹性集群的研究[D].浙江理工大学,2017.

张迟第. Kubernetes将统治云端[N]. 人民邮电,2017-12-14(007).

张建,谢天钧. 基于Docker的平台即服务架构研究[J]. 信息技术与信息化,2014(10):131-134.

吴坤安,黄文思,韩泽华,黄屏发,李芸,王荣烨,陈伟仑. 基于Docker的数据库微服务系统设计与实现[J]. 国外电子测量技术,2017,36(12):57-62.

黄永慧,陈程凯. HTML5在移动应用开发上的应用前景[J]. 计算机技术与发展,2013,23(07):207-210.

Brandon Butler. Kubernetes - the platform for running containers - is getting more enterprisey[J]. Network World (Online),2016.

Matt Asay. Why Kubernetes is winning the container war[J]. InfoWorld.com,2016.

Serdar Yegulalp. Kubernetes 1.4 makes container orchestration bigger -- and simpler[J]. InfoWorld.com,2016.