学校代码：10270　　分类号：　　学号：152300241

上海师范大学

硕士专业学位论文

基于 Docker 的分布式 Web 平台的研究与实现

学　　院：信息与机电工程学院

专业学位类别：工程硕士

专 业 领 域：计算机技术

研 究 生 姓 名：李志盼

指 导 教 师：陆黎明

完 成 日 期：2018-5-6

摘要

Docker 相较于传统的虚拟机，减少了 Guest OS 层的资源消耗，旨在提供更轻量级的基于进程的软件服务，它也为开发人员提供了快速的开发、测试、实施的平台环境。对于分布式 Web 平台的发布，Docker 也提供了集群化的管理发布方案。

目前，在持续集成发布、容器内资源性能日志全面监控调度、容器资源的弹性伸缩和应用的灰度发布方面，Docker 还有很大提升空间。持续集成方面，虽然借助持续集成工具 Jenkins 可以实现，但是相对于集成软件的操作，这对用户提出了较高的技术操作要求。另外，针对于集群内应用性能监控和日志，一直是容器运维厂商的难题，这样使得建立全方位监控和日志的方案变得尤为重要。将容器的弹性伸缩和基于应用的版本升级完美的应用于平台生产环境，实现容器应用生命周期的管理，一直是制约容器推广和应用的难点。如何有效解决这些问题，本文设计和实现的基于 Docker 的分布式 Web 平台将从持续集成、性能监控、灰度发布和日志检索方面进行探索。本文的工作和创新点如下：

首先，本项目设计的分布式 Web 平台中的一键式持续集成有效的解决了分布式 Web 应用的开发、测试、实施的环节环境不一致的问题，弱化了平台的差异性，使开发、测试更专注于业务的实现，简化应用发布周期。一键式持续集成实现了 SVN 代码管理工具与 Jenkins 持续集成工具的有机集合，节约了大量维护成本，同时也降低了持续集成的操作难度。

其次，本文针对使用 Docker 平台后，容器应用的资源性能数据采集、汇总和分析问题，提供了性能监控方案。并利用 Kubernetes 的特性，将容器资源实时的使用率加入到应用弹性伸缩的策略中，实现基于容器的资源监控和动态的自适应调整资源占用，做到了应用的弹性伸缩自适应。该方案能够实时地对应用节点进行弹性的控制，解决了 Docker 容器无法根据性能访问自适应的问题，使应用可以灵活的提供服务，同时在调度触发上引入调度日志的审计，确保其调度的有效性。

然后，对软件项目的版本变更升级探索了灰度发布的机制。通过项目实践中引入灰度发布，减少业务的中断，提高客户的产品满意度。同时，也为开发人员和运维人员节省了大量升级等待时间，降低了版本升级过程中出错率。

最后，对容器内大量性能信息和日志，建立了一体化的日志采集和分析平台，实现了容器内节点、应用和调度界面化的资源管理。该平台一方面提供了丰富的

查询接口，能够适应日志检索的多维度查询要求；另一方面降低了 Docker 容器资源的运维复杂性，节约了运维人力成本。

本文对分布式 Web 平台中的持续集成、性能监控、灰度发布和日志检索功能进行了逐一测试，验证了其有效性。该平台的设计和实现，为 Docker 虚拟化应用以及软件项目的持续集成、性能监控、灰度发布和日志检索提供了完美的解决方案，具有很强的现实意义。

关键词：持续集成；弹性伸缩；灰度发布；日志检索；Docker；Jenkins；Kubernetes

Abstract

Compared to traditional virtual machines, Docker reduces the resource consumption of the Guest OS layer and aims to provide a more lightweight process-based software service. It also provides developers with a rapid development, testing and implementation platform environment. For the distribution of distributed Web platforms, Docker also provides a clustered management publishing solution.

Currently, Docker still has a lot of room for improvement in terms of continuous integration release, comprehensive monitoring and scheduling of resource performance logs in containers, elastic scaling of container resources, and grayscale publishing of applications. In terms of continuous integration, although it can be achieved with the continuous integration tool Jenkins, it has a higher technical operation requirement for the user than the operation of the integrated software. In addition, application performance monitoring and logging for intra-cluster applications has always been a challenge for container operation and maintenance vendors. This makes it all the more important to establish comprehensive monitoring and logging solutions. The application of container elasticity scaling and application-based version upgrades to the platform production environment to achieve the container application life cycle management has always been a difficult point to restrict the promotion and application of the container. How to effectively solve these problems, the Docker-based distributed Web platform designed and implemented in this paper will explore continuous integration, performance monitoring, grayscale publishing, and log retrieval. The work and innovation of this article are as follows:

First, the one-click continuous integration in the distributed Web platform designed by this project effectively solves the inconsistencies in the development, testing, and implementation of the distributed Web application environment, weakening the differences in the platform, and making development and testing more effective. Focus on the realization of the business and simplify the application release

cycle. One-button continuous integration enables an integrated collection of SVN code management tools and Jenkins continuous integration tools, which saves a lot of maintenance costs and also reduces the difficulty of continuous integration operations.

Second, this article provides a performance monitoring solution for collecting, summarizing, and analyzing resource performance data for container applications after using the Docker platform. Utilizing the features of Kubernetes, the real-time usage rate of container resources is added to the elastic scaling policy, which implements container-based resource monitoring and dynamic self-adaptive resource allocation. This ensures that the application's elastic scalability adapts. The solution can flexibly control the application nodes in real time, solve the problem that the Docker container cannot access self-adaptation based on performance, enable the application to provide services flexibly, and at the same time introduce the audit of scheduling logs on the scheduling trigger to ensure the effective scheduling.

Then, we explore the mechanism of grayscale release for upgrading version of software project. Introduce grayscale publishing through project practice to reduce business interruptions and increase customer satisfaction. At the same time, it also saves a lot of upgrade wait time for developers and operation and maintenance personnel, and reduces the error rate during version upgrade.

Finally, an integrated log collection and analysis platform was established for a large number of performance information and logs in the container, and resource management of the nodes, applications, and the interface of the container was implemented. On the one hand, the platform provides a rich query interface, which can adapt to the multi-dimensional query requirements of log retrieval; on the other hand, it reduces the complexity of the operation and maintenance of Docker container resources and saves the operation and maintenance labor costs.

This article tests the continuous integration, performance monitoring, gray-scale publishing, and log retrieval functions in a distributed Web platform and verifies its effectiveness. The design and implementation of the platform provide a perfect solution for the continuous integration, performance monitoring, gray-scale

distribution, and log retrieval of Docker virtualization applications and software projects, and has a strong practical significance.

Key Words: Continuous integration; Elastic scaling; Grayscale publishing; Log retrieval; Docker; Jenkins; Kubernetes

目录

第 1 章 绪论

1.1 研究背景

云计算[1]借助于虚拟化软件，把大量主机 CPU 计算能力，磁盘的存储服务和网络带宽的资源统一起来，为用户提供弹性应用的能力。尽管在底层虚拟化方面，不同的云服务提供商的实现有较大差异，但对整体的资源利用率的提供、资源的按需分配都起到很好的控制作用[2]。

SaaS、PaaS、IaaS 是云计算的三层架构[3]，如图 1-1 所示，三者分别从基础资源、平台中间件、软件应用环境中竭力减少资源的消耗浪费，为用户提供高效和可扩展的云计算能力[4]。

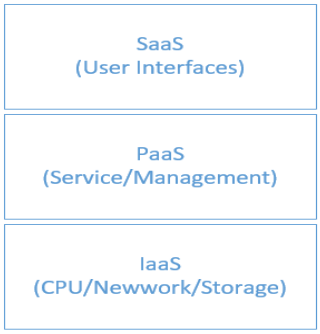

图 1-1 云计算三层架构

SaaS(Software as a Service)是基于软件的服务[5]。通过 SaaS 提供商提供的软件应用环境和运维，为企业提供开箱即用的软件服务。例如人事管理系统就是每个企业所必须的办公管理辅助软件，在 SaaS 软件出现之前，企业必须为人事管理系统购买主机，购买和发布人事管理系统，花费大量人力、财力、精力建立人事管理系统。但是，当采用 SaaS 软件后，企业只需按天或者月付给提供商费用，也不用担心软件的维护问题，直接就可以方便快捷的享受互联网的信息管理服务。

PaaS(Platform as a Service)是基于平台的服务[6]。通过 PaaS 提供商提供的专业的定制化平台，开发厂商可以快速发布软件的应用环境，更加专注于研发应用。PaaS 提供商一般提供开发工具、中间件和数据库等开发应用程序必须的软件服务。例如 Azure 就是一个具体的 PaaS。Azure 服务平台包括了以下主要组件：Windows Azure；Microsoft SQL 数据库服务、Microsoft .Net 服务；用于分享、储存和同步文件的 Live 服务；针对商业的 Microsoft SharePoint 和 Microsoft

Dynamics CRM 服务等。

IaaS（Infrastructure as Service）是基于基础设施的服务[7]。用户可从 IaaS 提供商中获得底层主机的计算、网络、存储使用服务。IaaS 承载了平台和软件的服务。PaaS 为研发人员提供了完整的开发平台，SaaS 则为企业或个人提供了直接使用的应用软件服务。例如国内比较著名的阿里云、腾讯云。

在云计算三层架构下，IaaS 层虚拟化提供的虚拟机，存在着资源利用严重浪费、调度臃肿、软件环境依赖不一致的问题。传统的 PaaS 平台发展依赖于 IaaS 平台，为解决此类问题，PaaS 厂商逐渐意识到利用容器技术可以很大的提高资源利用率的问题[8]，但是在应用部署上、服务环境一致上和资源应用的架构上 PaaS 仍有很大的限制，这也带来了运维投入过大和应用资源难以隔离的问题。

由此看来，IaaS 和 PaaS 均有其适应的场景，且仍有需要完善的地方。同时随着计算机领域技术发展的日新月异，多样性和差异性已成为应用平台扩展的主要障碍。以 Docker 为代表的新一代的 PaaS 平台[9]则实现了全方位的应用生命周期管理，且其良好的开放性、移植性、操作性，加快了应用交付周期。

Docker 是虚拟化的一种轻量级替代技术[10]。Docker 底层基于 Go 语言开发实现。Docker 的应用类似于集装箱。其可以打包应用所需的操作系统、平台环境和依赖应用到镜像，使发布应用时，可以完全隔离外界环境，达到一致化和标准化的应用环境。同时，Docker 采用分层的方式加载镜像，极大的提高了应用启动和发布的速度。

自从 Docker 在 2013 年作为 Public 项目开源以来，国内外越来越多的公司开始加入 Docker 生态圈。RedHat 在 Docker 的基础上开发了云 PaaS 平台， 亚马逊也发布了 EC2 容器服务平台，甚至于传统的 VMware 虚拟化厂商也开始支持 Docker。国内，阿里云、腾讯云、百度云、京东云纷纷都已经构建了基于 Docker 的云服务平台。基于国内网络下载缓慢的问题，DaoCloud 也提供了免费的镜像加速服务，很大程度上促进了容器应用的推广。自从 Docker 诞生依赖，几乎每年镜像的下载次数都呈倍数的增长，这无疑也证明了 Docker 应用技术的火热[11]。

整体而言，一个优秀的云计算平台，不仅要为用户提供基础的基于计算、存储和网络资源的虚拟化，更重要的是结合业务应用，一体化地解决应用平台的部署、调度、全生命周期的管理。但当今的基于 Docker 应用的容器平台，可提供的功能依然不够完善，比如 Docker 应用环境的发布和管理依旧是一个很大的难题。另一方面，应用平台没有完善的自调整机制。这些问题将限制云平台的发展和应用。

1.2 研究意义

Docker 因其部署应用的高效性已得到众多企业认可，也吸引着大量的开发者将自己的应用发布到容器中。但是，容器应用操作的便捷性和灵活性，始终影响着用户的服务体验和平台发展。当用户将应用发布到平台上时，希望可以快速地完成自动化的集成测试，毕竟代码库存越是积压，越是得不到生产验证，代码间交叉感染越大，发布下个版本的风险和复杂度将会提高。同时，代码越早发布，用户反馈就越早，越便于帮助市场人员及时收集和调整市场的策略。还有，面对资源的限制，用户非常希望建立基于资源监控反馈的调度，灵活的解决访问的拥塞，并对已有的极其简单的发布方案进行优化，以适应更大的访问量和数据量，并建立完整的监控和日志事件审计机制。

在持续集成方面，Docker 虽为应用提供充足的资源支撑，但是随着容器个数的增加，应用整体环境维护的成本显著增大，且在生产系统的迭代更新中，人为管理代码版本工作变得尤其重要。再说频繁的发布测试占用了运维的大量时间和精力，况且由于人为的疏忽等因素，极易造成发布的失败。

在容器的管理调度方面，由于容器的使用需要一定的 Linux 操作命令的基础，加大了 Docker 平台使用的难度。另外 Docker 平台也无法对集群的健康度进行检查，妥善调度集群内的资源分配，日常的监控和运维也缺乏一体化的运维方案。当利用容器平台发布应用时，环境排错和系统调优仍耗费相当大的人力，因此迫切需要一个便于操作和管理的容器 Web 平台，来管理容器的资源监控、灰度发布和日志检索功能。

持续集成与容器资源管理有着紧密的联系，容器资源监控为持续集成提供稳妥的调度策略，持续集成给容器资源管理带来便捷和灵活。因此，面对这些问题，本文主要将以 Docker 基础架构平台为例，努力完善容器内持续集成、资源监控、灰度发布和日志检索方面的缺陷。

1.3 研究现状

持续集成与容器内资源管理自从 Docker 出现以来，一直是比较热门的话题。相当多的学者已经开始这些问题的研究。

其中，在容器的持续集成方面，2006 年 Martin Fowler 就提出持续集成会改变整个软件行业的理念[12]。Paul M.Duvall 在他写的《持续集成-软件质量改进和风险减低之道》中系统化的论述持续集成的相关概念[13]。陶镇威[14]在《基于 Jenkins 的持续集成研究与应用》中提出的 Jenkins 持续集成方案丰富了自动化测

试的脚本，但没有解决环境不一致的问题。同样王宁[15]在《基于 Jenkins 的持续集成系统的设计与实现》中定制开发了 Jenkins 的插件和增加了反馈的机制，但也没有解决环境不一致的问题。钟炜达 [16]在《一种基于 Docker 的持续集成平台的设计与实现》中提出了使用 Jenkins 的任务分发调度，来实现华南理工大学能源云平台集成的自动化问题，但是在用户体验和 Jenkins Master 分发方面还有较大提高的空间。张成[17]在《基于 Docker 的持续集成系统的设计与实现》也搭建了一个自动构建平台，但是平台强依赖构建工具 Shell，无疑增大了后期规模扩大后脚本的维护工作。

在容器资源管理方面，仇臣[18]在《Docker 容器的性能监控和日志服务的设计与实现》中采用 Agent 的方式对容器集群性能进行采集，但是没有将性能监控应用到弹性场景。刘辉扬[19]在《基于 Docker 的容器监控和调度的研究与实现》也把性能监控加入到了资源调度的策略中，但是仅仅应用于 Web 应用的实例，也没有真正统一开发和测试环境，没有解决各其他服务如数据库的弹性伸缩问题。边俊峰[20]在《基于 Docker 的资源调度及应用容器集群管理系统设计与实现》提出基于 SLA 驱动的资源动态算法来满足系统资源的调度，延长了调度的周期，但是在资源调度触发有效性上缺乏了一定的保证。何思玫[21]在《面向容器云平台的集群资源调度管理器的设计与实现》中把最小费用最大流问题在容器云平台的资源调度上做了验证，但调度过程仍强依赖于人工参与，无法实现容器资源自适应的问题。周佳威[22]在《Kubernetes 跨集群管理的设计与实现》中对于跨集群的服务发现和自动迁移问题，进行了详细的验证，但是其调度和迁移过程中也没有引入容器资源的实时的使用率。

1.4 主要工作

针对云计算平台在持续集成方面和构建的 Docker 平台低效资源利用率、调度应用任务缓慢、容器资源管理混乱等多种问题[23]，本文依据上海电信 CRM 云化 POC 测试的需求，利用 Docker 容器作为应用部署和资源调度的基本单位，使用 Kubernetes 对容器集群进行管理，完成了 Docker 容器平台的管理与调度平台的设计与实现。首先利用 Jenkins 对代码的变动进行持续的版本发布和升级；其次，建立统一的容器资源管理平台，利用 Kubernetes 集群管理策略对 PaaS 平台进行发布、监控和日志采集，以及对应用中的版本上线进行灰度发布。具体工作内容如下：

（1）利用 Docker 以及 Jenkins 构建一个可持续发布代码版本的平台，解决环境发布不一致的问题。

（2）使用 Kubernetes 管理容器集群的实例，并对应用中的容器监控进行研究和设计，建立基于资源动态反馈调节的弹性容器资源管理。

（3）对 Kubernetes 管理容器中的灰度发布策略进行研究和设计，解决容器应用发布问题，实现业务不间断升级。

（4）对 Kubernetes 管理容器中的操作日志进行采集，对容器平台的事件和日志建立审计制度。

对比现有架构和技术方案，本文的创新点主要如下：

（1）持续集成：实现 SVN 代码管理工具与 Jenkins 持续集成工具的有机集合，提供界面化的一键式持续集成操作方式，节约维护成本和难度。

（2）资源监控：将容器资源实时的使用率加入到应用弹性伸缩的策略中，实现基于容器的资源监控和动态的自适应调整资源占用，做到应用的弹性伸缩自适应，并在调度触发上引入调度日志的审计，确保调度的有效性。

（3）容器管理：提供容器内主机、应用、业务的日志监控策略，并支持多条件查询。对容器内性能信息和日志进行采集和管理，提供全方面的和全方位运维支撑；提供容器内节点、应用和调度界面化的资源管理。

（4）平台开发：设计的分布式 Web 平台集成了持续集成、资源监控、灰度发布和日志检索的功能，实现了对 Docker 容器资源的较为全面的管理，为 Docker 虚拟化应用以及软件项目的持续集成、性能监控、灰度发布和日志检索提供了完美的解决方案。

1.5 组织结构

第一章：绪论。介绍云计算的发展历程，分析国内外的云计算与容器技术的现状和特点，并指出现有 Docker 平台在持续集成和资源管理方面的不足，以此作为本文的主要研究方向和内容。

第二章：理论基础以及相关的技术研究。主要对项目中所涉及到的技术和知识进行详细介绍，有 Docker、Jenkins、Kubernetes 集群容器管理和 EFK 等等。

第三章：总体设计。对用户在 Docker 中持续发布代码的需求进行分析，以 Jenkins 为基础，对发布中的调度问题进行分析。接着介绍了以 Kubernetes 为基础部署分布式应用的架构，并基于此架构设计了整个平台的资源监控，灰度发布和日志检索方案。

第四章：系统实现。详尽说明项目的核心功能的原理和解决方案。主要是持续集成功能、资源监控功能、灰度发布功能和日志检索功能的设计与实现。

第五章：测试与结论。汇总平台测试的主机环境，通过对应的测试用例，完

成相应功能的测试和验证。

第六章：总结与展望。全面总结了本文的工作，说明了所做研究和工作的意义，指出了存在的不足以及将来可以改进的方面。

1.6 本章小结

本章对本项目的研究背景，研究意义和研究现状进行了简要说明。首先对当前云计算虚拟化的背景、现状和发展进行了详细的阐述，并指出在 IaaS 层虚拟化的不足，接着对基于 Docker 的半虚拟产品进行论述，也指出现有 Docker 平台管理的不完善之处。随后，提出为 Docker 平台持续集成、资源监控、灰度发布和日志检索的需求价值，并说明了在持续集成、资源监控、灰度发布和日志检索的主要实现内容，并提出本文的主要研究目标，也对本文的主题结构进行了描述。

第 2 章 平台相关技术研究

2.1 Docker

Docker[24]是 PaaS 提供商 DotCloud 开源的基于 Linux 内核虚拟化的容器引擎，基于 Go 语言开发并且项目已开源，源码托管在 GitHub[25]上，开发者凭借 Docker 容器可以非常容易的实现打包应用程序以及所依赖的操作系统环境，从而发布应用到其它环境中，极大地简化了平台之间的差异性，具有较强的移植性[26]。另外，由于其只打包了应用和其依赖的环境，相较于传统的虚拟化技术，减少了操作系统层的资源加载，应用运行时又采用了分层技术，底层已构建的镜像会设置成只读模式，当需要写入时会写入其上的读写层，其他应用也可以复用底层镜像，极大的提高资源的利用率。

2.1.1 Docker 概述

Docker 项目是一个轻量级的虚拟化解决方案[27]，其在 Linux 内核虚拟化的基础上进行了进一步的封装，与传统的虚拟化借助于硬件层面的实现方式不同，Docker 主要在操作系统层面进行虚拟化，极大地复用了主机操作系统的资源。Docker 与传统的虚拟化实现层面的对比如图 2-1 所示。

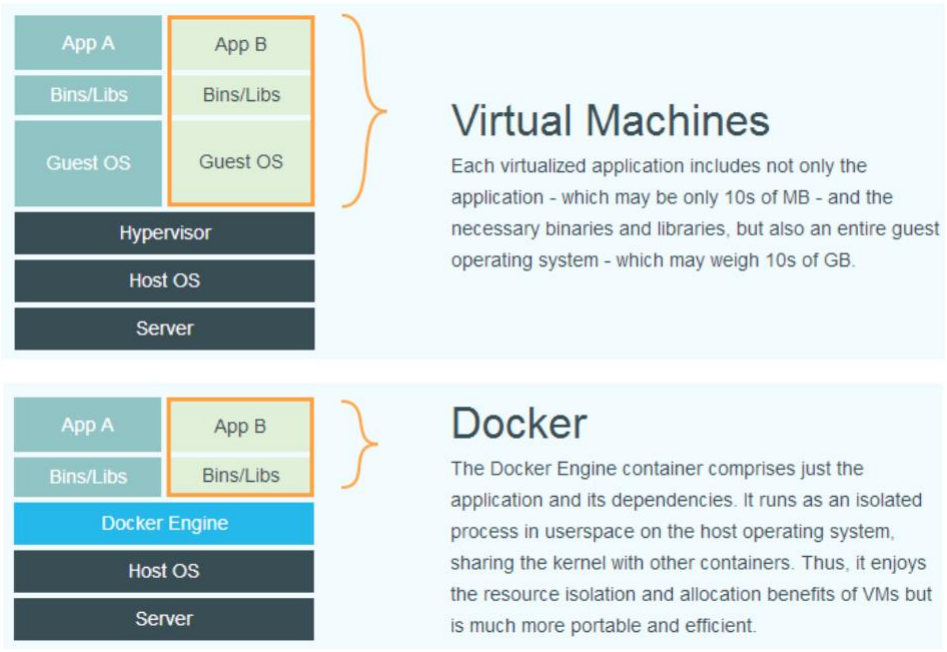

图 2-1 Docker 与传统虚拟化实现对比

2.1.2 Docker 体系结构

Docker 是一个基于 C/S 的架构模式，通过客户端 Docker Client 或者 REST API[28]向 Docker Daemon 守护进程发送请求，Docker Daemon 守护进程处理完成后返回给客户端。Docker 的主要组成如下。

- 镜像： Docker 镜像是容器运行的静态模板，每个镜像都开始于一个基础镜像，每个镜像都包含一些列的层，Docker 使用 UnionFS（联合文件系统）将这些层组成镜像。
- 容器：Docker 容器是由 Docker 镜像创建而来，主要包含了操作系统、用户文件以及元数据。容器运行时依赖于 Namespaces(命名空间)和 UnionFS 实现了命名空间和存储的隔离，同时借助于 Linux 底层的 CGroup（控制组）实现资源的隔离。
- 仓库： Docker 仓库用于保存 Docker 镜像，一般分为共有和私有。

Docker 虽然火热，但并非完全是个新技术，因为其主要是实现了 Linux 内核技术的封装。Docker 使用到的技术如下。

- Namespaces 是 Linux 提供的一种内核级别环境隔离的方法，其提供了类似于 chroot 命令（修改根目录到指定目录下）的方式，实现对 UTS（主机名）、IPC（进程间通信）、NS(挂载点)、PID（进程 ID）、NET(网络访问)、User（虚拟用户映射）等的隔离机制[29]，使容器中运行进程而且不能影响容器外的其它进程。
- Control Groups 是 LXC 的重要组成部分，主要对资源进行配额和度量。其通过配置限制 CPU、Memory、Blkio(块读写 IO)、Devices（设备）的方式，实现资源核算和限制功能。
- UnionFS 是一种分层的轻量级的并且高性能的文件系统，它支持对文件系统的修改以层的方式叠加，同时挂载到同一个虚拟文件系统下。

2.1.3 Docker 部署应用

Docker 的发行版本分为 CE(community edition)社区版和 EE(enterprise edition)企业版，CE 由社区维护和提供技术支持，为免费版本；EE 版本为收费版本，由售后团队和技术团队支持技术支持。它不仅支持主流的 Linux 操作系统，也支持 Mac OS(苹果操作系统)和 Microsoft Windows 10。简单的部署和应用主要如下。

（1）部署

以在 CentOS7 下部署 CE 版本的 Docker 为例，由于其已内置 Docker 源，输入下面命令直接安装即可（需确保主机联网）。

```
yum install -y docker
```

安装完成后假如 Docker 服务未启动成功，需要通过 systemctl 启动 Docker 的服务。

```
systemctl start docker
```

（2）应用

Docker 启动完成后，可以通过 Docker version 查看已安装 Docker 的版本，创建简单的容器应用 hello-world 命令如下

```
docker run hello-world
```

2.2 Jenkins

Jenkins[30]是一个基于 Java 语言开发的开源的持续集成工具，用于监控持续集成的重复性的工作。通过 Jenkins 的自动化的集成，能减化开发人员的版本发布工作，从而降低人工发布操作失误的风险，使开发人员更加专注于业务代码的实现和优化。

2.2.1 Jenkins 概述

Jenkins 起源于 Hudson（Hudson 是商用的）[31]，主要用于持续、自动的构建/测试软件项目、监控外部任务的运行。Jenkins 提供了一个平台化的能力，把各个环节的组件接入进来，实现端到端的交付。常用的版本控制工具有 SVN、GIT，构建工具有 Maven、Ant、Gradle。

2.2.2 Jenkins 体系结构

Jenkins 对于软件的开发，有着非常重要的作用。Jenkins 采用 Master/Slave 架构，Master 提供 Web 界面的方式管理 Job 和 Slave，Job 可以运行在 Master 节点上或者被分配到 Slave 上，Slave 上主要是执行 Job 相应的任务，此时通过双向字节流的连接与 Master 进行通信。

Jenkins 持续集成具有的特性：易于安装和配置；具有详细的日记记录和跟踪；丰富的第三方插件的支持。

2.2.3 Jenkins 部署应用

Jenkins 的部署主要有四种方式，可以通过在线 yum 命令或者下载 Jenkins.war 应用包放至 Tomcat 中$CATALINA_BASE/webapps 下，还有就是通过 java -jar

命令方式启用，此处安装采用了官网推荐的 Docker 镜像方式安装。

（1）部署

以在 CentOS7 下部署最新版本的 Jenkins 为例

```
docker run -p 8080:8080 -p 50000:50000 -v /home/Jenkins_home:/var/Jenkins_home jenkins
```

此处使用-p 是将容器内 8080 和 50000 端口映射到主机端口上，使用-v 参数是将容器内/var/Jenkins_home 映射到主机/home/Jenkins_home 上。

（2）应用

Jenkins 安装完成后，可以通过浏览器 http://127.0.0.1:8080 访问，第一次使用的话，需要配置相应的插件，如 JDK、Maven 等。

2.3 Kubernetes

Kubernetes(简写为 K8s）[32]是 Google 公司开源的容器集群管理系统。其构建于 Docker 之上，提供了一整套的应用部署、维护、扩展机制，通过 Kubernetes 可以实现方便地管理跨机器运行容器化的应用。

2.3.1 Kubernetes 概述

Kubernetes 是一个全新的基于容器技术的分布式架构领先方案。自从开源后，其成为一个开放的容器管理平台，引来越来越多的开发者加入。随着企业实践的增加，Kubernetes 被逐渐证明是一个完备的分布式系统支撑的集群容器管理平台，连 RedHat 公司的集群管理产品 OpenShift[33]底层也是使用 Kubernetes 的集群管理技术。

Kubernetes 作为容器资源管理的首选，可以轻松实现系统线上和线下的升级，另外其提供了强大的横向扩容能力，非常适合微服务场景的开发。

2.3.2 Kubernetes 体系结构

Kubernetes 是一个高度自动化的资源控制系统，通过 Kubernetes 提供的 Kubectl 工具或者 API 调用实现对资源对象 Master、Node、Replication Controller、Service 等的操作，并将资源的状态保存在 Etcd(配置共享和服务发现的键值存储仓库)[34]中。Kubernetes 的资源对象有以下几种[35]。

1．Master

Master 是 Kubernetes 中的集群控制节点，负责整个集群的管理和控制。基本上所有的控制命令都是由 Master 调度的，假如 Master 节点不可用，所有的控制

命令即将会失效。

Master 节点运行着一系列相关进程：Kubernetes API Server(kube-apiserver)以 HTTP Rest 方式提供了资源操作的关键服务进程；Kubernetes Controller Manager(kube-controller-manager)是所有资源的自动化控制中心进程；Kubernetes Scheduler(kube-scheduler)是资源调度的进程。

2．Node

Node 是 Kubernetes 集群中相对于 Master 而言的工作节点（之前也称为 Minion），由 Master 分配工作负载。在 Node 上运行的关键服务：kubelet 负责 Pod 应用的创建启停操作以及配合 Master 节点的集群资源管理；kube-proxy 实现 Kubernetes Service 的通信和负载机制；Docker Engine 负责本机的容器创建和管理。

Node 节点可以在运行期间动态增加到 Kubernetes 集群中。默认情况下本节点的 kubelet 会向 Master 注册，注册成功后，kubelet 进程会定时向 Master 节点汇报节点的信息，包括操作系统、Docker 版本、CPU、内存以及 Pod 的运行情况。

3．Pod

Pod 是 Kubernetes 的最基本操作单元，包含一个或多个紧密相关的容器，类似于豌豆荚的概念。Kubernetes 为每个 Pod 都分配了唯一的 IP 地址，一个 Pod 里的多个容器共享 Pod IP 地址。一个 Pod 里的容器与另外主机上的 Pod 容器通过虚拟二层网络技术（如 Flannel、Openvswitch）实现直接通信。

Pod 分为两种类型：静态 Pod 存储在 Node 上的具体文件中，只能在此节点上运行；普通的 Pod 会被持久化到 etcd 中，随后被 Master 调度到 Node 上进行绑定，接着被该 Node 上的 kubelet 进程实例化成容器并启动。

每个 Pod 都可以对其能使用的服务器计算资源设置限额，当前设置限额的计算资源有 CPU 和 Memory。千分之一的 CPU 配额为最小单位，用 m 来表示，由于 CPU 配额为绝对值，保证了 Pod 在不同机器上占用资源的一致性。Memory 配额也是绝对值，单位是内存字节数。

4．Label

Label 是一个 key=value 的键值对，key 和 value 有用户自己定义。资源对象通过定义 Label 可以实现多维度的资源分组管理。

5．Replication Controller

Replication Controller（简称 RC）是 Kubernetes 系统中的核心概念，用于定义 Pod 副本的数量。在 Master 内，Controller Manager 进程通过 RC 的定义来完成 Pod 的创建、监控、启停等操作。

根据 Replication Controller 的定义，Kubernetes 能够确保在任意时刻都能运行用于指定的 Pod“副本”（Replica）数量。如果有过多的 Pod 副本在运行，系统就会停掉一些 Pod；如果运行的 Pod 副本数量太少，系统就会再启动一些 Pod，总之，通过 RC 的定义，Kubernetes 总是保证集群中运行着用户期望的副本数量。同时，Kubernetes 会对全部运行的 Pod 进行监控和管理。

通过对 Replication Controller 的使用，Kubernetes 实现了应用集群的高可用性。通常，Kubernetes 集群中不止一个 Node，假设一个集群有 3 个 Node，根据 RC 的定义，系统将可能在其中的两个 Node 上创建 Pod。

6.Deployment

Deployment 引入是为了解决 Pod 的编排问题。Deployment 相对于 RC 展示了当前 Pod 部署的进度，由于 Pod 的创建、调度和启动需要一定的时间，连续变化的部署过程会体现在 Deployment 中。

7.Horizontal Pod Autoscaler

Horizontal Pod Autoscaler (简写为 HPA)实现了 Pod 的扩容和缩容，与之前的 RC、Deployment 一样，都属于 Kubernetes 的资源对象。HPA 主要是通过追踪分析 RC 控制的所有目标 Pod 的负载变化情况，来确定是否需要针对性的调整目标 Pod 的副本数。

8.Service

Service 是 Kubernetes 的核心对象之一，定义了一个服务的访问入口地址。Service 最早使用环境变量实现服务发现，但由于不够直观，引入了 DNS 系统，把服务名当作 DNS 域名。

9.Volume

Volume 是 Pod 中能够被多个容器访问的共享目录，与 Docker 中的 Volume 类似，但 Kubernetes 的 Volume 主要对应的是 Pod。

10.Persistent Volume

Persistent Volume(简写为 PV)相当于 Kubernetes 集群中的某个网络存储中对应的一块存储。

11.Namespace

Namespace 实现了多租户的资源隔离。默认会创建一个名为“default“的 Namespace。结合 Kubernetes 的资源配额管理，限定不同租户的占用的资源，例如 CPU 使用量、内存使用量。

12.Annotation

Annotation 与 Label 类似采用了 key/value 键值对进行定义，但 Label 定义为 Kubernetes 对象的元数据，而 Annotation 则是用户任意定义的附加信息。

2.3.3 Kubernetes 部署应用

Kubernetes 的安装依赖于 Etcd 和 Docker 环境，故安装 Kubernetes 之前需要先安装 Docker 环境和 Etcd，Docker 的安装参考 2.1.3 节，Etcd 可通过命令 yum install -y etcd 安装。

（1）部署

以在 CentOS7 下部署 Kubernetes 为例。

```
yum install kubernetes
```

安装完成后需要配置/etc/kubernetes 下对应的 apiserver、kubelet 和 config 文件，启动服务的命令如下。

```
systemctl start kube-apiserver.service
systemctl start kube-controller-manager.service
systemctl start kube-scheduler.service
systemctl start kubelet.service
systemctl start kube-proxy.service
```

（2）应用

Kubernetes 启动完成后，可以通过 kubectl version 查看已安装的版本，查看 default 命明空间下的 Pod 命令如下

```
kubectl get pods
```

2.4 EFK

随着互联网技术的发展，原来的单机发展到多机再到大规模集群。众所周知，一个系统由大量的服务构成，其中每个应用与服务的日志分析管理也变得越来越重要[36]。本文主要介绍使用 Fluentd+Elasticsearch+Kibana 搭建日志收集系统。

2.4.1 Fluentd 简介

Fluentd[37]是完全开源的日志采集工具，实现了日志收集、处理和存储，从而不需要编写特殊的处理脚本。Fluentd 的性能已经在各领域得到了证明：从 5000+服务器上收集日志，每天 5TB 的数据量，Fluentd 在高峰时间可以达到 50,000 条信息每秒。Fluentd 主要采用分布式形式部署。

Fluentd 是一个专为处理数据流设计的开源数据收集器（有点像 syslogd），但是它使用 JSON 作为数据格式。Fluentd 集成插件式的架构决定了其高可扩展性和高可用性。

Fluentd 是以一个守护进程的方式运行。实际生产中，各种不同来源的信息首先发送给 Fluentd，接着 Fluentd 根据配置通过不同的插件把信息转发到不同的地方（比如文件、SaaS Platform、数据库），甚至可以转发到另一个 Fluentd。

使用 Fluentd 接触比较多的主要是 td-agent.conf 文件，其中配置主要有以下 6 类，分别为：

- source： 定义输入。
- match：定义输出的目标，如写入对应的消息队列，或者发送到指定地点。
- filter：过滤功能，对指定的信息进行筛选。
- system：系统级别的设置。
- label：定义组操作，实现复用和路由。
- @include：引入其他文件，与 Java、Python 中的 import 类似。

2.4.2 Elasticsearch 简介

Elasticsearch[38]是一个使用 Java 开发的基于 Apache Lucene(TM)的开源搜索引擎。Lucene 一直被认为是最先进、性能最好的、功能最全的搜索引擎库，但是其使用具有一定的复杂。Elasticsearch 使用 Lucene 作为其技术核心来实现所有索引和搜索的功能，它的目的是通过简单的 RESTful API 来简化 Lucene 的高复杂性。Elasticsearch 的特性主要为：

- 采用分布式实时的文件存储并且支持每个字段的索引查询。
- 分布式的实时分析搜索引擎。
- 高扩展性并支持 PB 级结构化或非结构化数据。

丰富的功能集成到服务里面，应用可以通过简单的 RESTful API 或者简单的命令行与之交互。

Elasticsearch 为 Java 用户提供了两种内置客户端：

（1）节点客户端(node client)：节点客户端起查询请求控制作用。其本身不存储数据，但是其可以提供数据在集群中的具体位置，并且能够将请求直接转发到对应的节点上。

（2）传输客户端(transport client)：能够发送请求到远程集群。其本身不加入集群，只是将请求转发给集群中的节点。

两个 Java 客户端都通过 9300 端口与集群交互，使用 Elasticsearch 传输协议(Elasticsearch Transport Protocol)。9300 端口也是集群中的节点之间也通信端口。

Elasticsearch 的 RESTful API 是基于 HTTP 协议以 JSON 为数据交互格式请求，向 Elasticsearch 发出的请求的组成部分与其它普通的 HTTP 请求是一样的：

curl -X<VERB> '<PROTOCOL>://<HOST>:<PORT>/<PATH>?

<QUERY_STRING>' -d '<BODY>'

- VERB：HTTP 方法如 GET, POST, PUT, HEAD, DELETE。
- PROTOCOL：http 或者 https 协议（代理时使用 https）。
- HOST：Elasticsearch 集群中的节点的主机名。
- PORT：Elasticsearch HTTP 服务所在的端口，默认为 9200。
- PATH：API 路径（例如_count 将返回集群中文档的数量），PATH 可以包含多个组件，例如_cluster/stats 或者_nodes/stats/jvm。
- QUERY_STRING：一些可选的查询请求参数，例如?pretty 参数将使请求返回更加美观易读的 JSON 数据。
- BODY：一个 JSON 格式的请求主体。

2.4.3 Kibana 简介

Kibana[39]是一个开源的分析与可视化平台，主要配合 Elasticsearch 使用。通过 Kibana 可以搜索、查看存放在 Elasticsearch 索引里的数据，另外 Kibana 提供各种不同的图表、表格、地图便于高级数据分析与统计。

Kibana 让我们理解大量数据变得很容易。它使用简单，不需要写任何代码，没有其他基础软件依赖，基于浏览器的接口能快速创建和分享，实时展现 Elasticsearch 查询变化的动态仪表盘。

Kibana 是一个 Web 应用，可以通过 5601 端口访问。例如：127.0.0.1:5601。当访问 Kibana 时，会默认加载 Discover 页面，使用默认的索引格式。时间过滤器设置为近 15 分钟，搜索查询设置为 match-all(*)，即可展示当前日志。

2.5 本章小结

本章介绍了本项目是使用的理论基础和相关技术。主要介绍了 Docker、Jenkins、Kubernetes 和 EFK 的定义，以及体系结构和简单的部署应用。Docker 主要为虚拟化平台，性能优势突出，简化再部署，降低风险系数，缩短环境交付和部署周期。另外，Docker 所占空间较小，删除后所包含的文件也会被删除，充分提高物理空间利用率，所用的迭代策略简单快捷。综合上述优势简析，结合 Docker 的分布式 Web 平台的构建有较强的应用优势。

第 3 章 总体设计

3.1 系统需求分析

需求是一个系统设计的指南，明确需求，有针对性地进行系统设计，才能更为有效的服务实际应用。本文提出的分布式 Web 平台需求主要来源于上海电信 CRM 云化 POC 测试项目，其测试目标为以互联网化的“平台+应用”架构为目标，以 CRM 订单业务场景为 POC 应用，综合验证 PaaS 云平台的能力。测试的主要方向为 CRM 订单业务和平台运维。由于本人主要参与了该 POC 测试项目的平台运维方面的功能，故本文主要针对平台运维方面的测试用例。其中关于平台运维方面的主要有以下测试要点。

- 可靠性：当开发者代码提交后，能够快速的进入测试和版本发布，保证生产环境、测试环境和开发环境运行的一致可靠性。
- 高性能：当机器资源一定时，最大限度地利用平台的 CPU、内存、磁盘和网络资源，提供高效运转的平台支撑。
- 弹性伸缩、多租户：各应用在高低峰情况下进行扩容或缩减，同时需保证系统正常运行。另外，各租户应用平台资源要进行隔离，互不影响。
- 业务连续性：应用系统在长时间运行情况下，各节点的状态的稳定性。
- 业务可监控：针对应用、平台、硬件的资源的可监控和运维能力。

由上述需求可知，分布式 Web 平台建设的核心要点是：①提供保证各环境一致的可靠的发布调度平台，即持续集成，稳定的发布应用平台版本；②提供保证平台资源使用率均衡和全面监控的策略，即有效资源监控，有效及时的掌握全局应用运行状态；③保证应用各平台资源调度的高效、灵活、稳定、隔离、不间断，即稳定灰度发布，实现业务连续和在线弹性伸缩机制；④并对各节点的日志及时获取，即高效的日志检索功能，提供稳定的审计和日志服务。

为满足测试点高性能的需求，最大限度地利用平台资源。首先考虑云计算虚拟化技术，但传统的 IaaS 层虚拟化，还存在实施复杂和调度缓慢的问题。相比之下，Docker 作为新兴的半虚拟化产品，入门相对简单且对资源的利用更高，国内很多公司对 Docker 开始了产品化定制开发，Docker 技术研究也逐渐成熟，故本次测试主体架构采用 Docker 虚拟化。

针对可靠性、弹性伸缩多租户、业务连续和业务监控的测试点，使用 Docker 可以达到事半功倍的效果。根据测试核心要点的功能分析，本分布式 Web 平台

主要实现的功能如下。

- 持续集成：实现代码的自动化发布。
- 资源监控：实现平台性能监控和基于性能监控反馈的弹性伸缩。
- 灰度发布：实现版本的无感知的升级。
- 日志检索：针对平台日志事件，提供检索。

3.2 持续集成概要设计

持续集成（Continuous Integration，简称 CI）是软件开发环节保证项目稳定开发和快速版本迭代的首要环节。若要软件开发团队成员高效协同工作并确保软件开发的质量，持续集成已成为开发过程中不可回避的问题[40]。

由于本次项目的时间周期较短，且中间版本是否升级发布是依靠人工识别，也避免无效的中间版本的发布，故本项目采用一键式自动化触发项目的构建，即当开发组长确认版本可以升级后，只需要对指定的 SVN 代码位置，选择合适的 Docker 容器的镜像，就可实现版本的编译和发布环节。由于每次升级均是启用新的容器应用，这样也摒弃了开发环境、测试环境、生产环境的不一致的元素，保证环境发布高度可靠性。另外，持续发布环境与 Docker 容器结合，Jenkins 将是最优秀的持续集成平台，其可以充分利用容器的镜像技术，实现应用的自动化构建和快速启动。

要实现持续集成，主要需要解决的问题有持续集成环境的搭建和持续集成中与 Docker 的集成流程。环境搭建主要是指搭建一个可以从 SVN 代码服务器上拉取代码，自动的进行代码合并，并结合本地 Maven 环境进行编译、构建、自动化测试的集成平台；Docker 的集成流程主要是把最新的代码版打包，利用容器的镜像快速构建应用的环节。

本次项目持续集成，集合了 Jenkins、Maven 和 SVN 功能，主要达到的目的如下。

- 减轻软件发布时压力，保证团队代码提交质量。
- 集成和测试环节实现自动化，避免人工干预，防止过程重复发生，缩短时间、降低费用和减少工作量。
- 代码尽早发布，实现高效快速迭代。并可对用户早反馈问题，尽可能有计划适应。
- 积压的代码库存影响生产检验，积压量越大，交叉感染的概率增加，下个版本的复杂度、管理成本和风险越高。

3.2.1 持续集成环境分析

搭建持续集成环境，是发布代码的重要环节。搭建持续集成环境与开发人员的熟悉程度和项目需求的难易程度关系紧密。

优秀的开发人员往往善于总结和归纳，并在实践中提升。从 2.2 节中了解到 Jenkins 是一个端到端交付的优秀的持续集成软件，并熟悉了其简单的部署，对于 CRM 云化 POC 测试项目的测试点可靠性，具有较高的吻合性。Docker 与 Jenkins 的完美搭配可以简化应用的部署方式、隔离操作系统与应用之间的强耦合和标准化应用的启停操作方式，有效地解决编译和运行时依赖、环境不一致和泛滥的管理部署问题。

结合本次 CRM 云化 POC 的持续集成的测试需求，还有一些隐藏的需求需要实现，比如要求现场进行演示，使用局方的内部网络，底层的 Docker 虚拟化环境搭建、Jenkins 环境的搭建、Jenkins 与 Docker 的结合配置、版本管理工具 SVN 集成和编译的工具 Maven 集成等等，都是需要考虑的问题。按照问题的类型可以分为安装类和集成类的问题。

对于安装类的问题，主要需要解决的是内网上网的限制问题。由于 Linux 基本上都内置了 Yum 的软件包管理工具，当安装 Jenkins 时会自动的下载安装相关的依赖，极大的方便了网络上软件包的安装，但由于此次测试的网络限制，需要在内网环境下安装。此次，本项目采用了 Yum 下缓存 Rpm 安装包的方式，获得所需的版本包依赖，然后通过上传 Rpm 安装包的方式解决 SVN、JAVA、Maven、Docker、Kubernetes 的安装问题。

对于集成类的问题，需要谨慎的逐个进行测试。首先是 Docker 和 Kubernetes 的版本兼容问题，本次项目安装采用了 Docker1.12.6 版本，至少需要安装 Kubernetes1.6 版本。其次是 Docker 和 Jenkins 的集成问题，由于本次基础环境采用 Docker 搭建，安装 Jenkins 也采用了 Docker 镜像的方式，为便于 Docker 镜像的快速复用，也需要搭建 Docker 私有仓库，并将需要的 Jenkins 镜像和应用服务器的基础 Tomcat 镜像打包上传至 Docker 私有仓库。然后是 Jenkins 与插件的集成，由于采用的是镜像部署方式，待应用部署成功后，需要安装其推荐的插件软件，并关联对应的 JDK、Maven、SVN。最后需要处理的是分布式 Web 平台与 Jenkins、SVN 和 Docker 镜像的有机结合，通过 JAVA 后台组装的 POST 请求调用 Jenkins 的 API 接口，完成任务 Job 的创建。其中 POST 请求的组装需要设置 SVN 代码源路径和编译完成后执行的 Docker 运行命令，最后以 XML 文件的方式提交到 Jenkins 的 API 中。

持续集成环境的搭建需要有准确的方向指引和在发布环境上的不断调试，中

间需要多次查询官网的安装文档和查看调试安装过程中软件或主机的日志。持续集成环境搭建是一次性投入终身受益的工作，在后期代码版本的发布时，会深切的感受到自动化带来的便利，从而节省出宝贵的时间投身于业务代码的创造当中。

3.2.2 持续集成流程分析

持续集成流程主要是将 Jenkins 和 Docker 的功能串连起来，达到一键式平台发布的作用。将 Jenkins 的特性和 Docker 结合，主要的持续集成流程如下：

（1）开发者提交代码

（2）分布式 Web 平台调用 Jenkins 的 API 触发一键式编译

（3）借助于 Dockerfile 完成应用镜像的构建运行

（4）使用 docker tag 标记应用镜像为本地镜像

（5）使用 docker push 上传应用镜像到本地私有仓库

通过以上流程可以发现，除第 1 个受限于代码版本确认和质量检测的人为因素，其它均可以在流程中实现代码版本的自动化一键式持续集成。一键式持续集成需要解决 SVN、Dockerfile 配置文件的参数收集和 Jenkins 拉取代码自动编译以及编译完成后应用镜像创建操作。然而 Docker 镜像的构建、上传和下载将是耗时较为严重的环节，为解决这个问题，采用建立 Docker 私有镜像仓库。接下来主要分析对应的参数收集、自动编译和应用镜像创建的需求。

（1）参数收集：良好的交互界面不仅能提供给客户优越的交互体验，还能带来生产力的提升。传统的持续集成参数收集界面交互复杂，用户的关键元素识别率低且配置繁琐，故本项目提出一键式持续集成方案。参数输入需要包括：镜像的名称、镜像的简介、代码仓库地址、项目名称、Dockerfile 位置。

（2）自动编译：自动编译需要实现从 SVN 拉取源代码，在 JDK 和 Maven 环境下进行编译。传统的持续集成主要是在 Jenkins 上创建任务 Job，然后输入对应的 SVN 参数等等，接着点击“立即构建”，由 Jenkins 负责整体的编译工作。本次项目的自动编译环节可以复用 Jenkins 的自动编译请求，但请求的方式更换成了 API 调用方式，可以极大地提升工作效率，避免系统之间的切换，使分布式 Web 平台功能更全面和智能化。

（3）应用镜像创建：应用镜像的创建必须在自动编译之后，待自动编译成功后，需要使用 Dockerfile 和镜像创建出应用的最终可运行的镜像版本，并将版本的镜像上传至本地私有仓库，方便做应用版本的副本和重建操作。

依靠一键式持续集成可以有效解决 SVN、Docker 和 Jenkins 的集成问题，并将最终创建的应用镜像资源上传至私有的镜像仓库，方便以后基于此镜像的应用

轻松的发布于测试和生产环境。

本项目分布式 Web 平台构建的一键式持续集成提供了人性化的 Web 管理界面，简洁的持续集成选项和功能完备的 API 接口，可以清晰的展示文件应用结构，让开发者专注于开发本身。

3.3 资源监控概要设计

资源监控模块使运维人员能够监视和控制系统中每个节点和应用程序容器中当前资源的使用情况。运维人员从节点列表中选择待监控节点，并获取该节点下的应用容器列表以及节点 CPU 和内存资源利用率图表信息。从应用程序容器列表中，选择需要监视的应用程序容器，并获取应用程序容器 CPU、内存资源使用图表信息。

资源监控是软件开发项目的保障性环节。通过建立全容器平台资源性能的监控，可以及时了解到平台的资源利用和健康性检查。当监控平台建立之后，可以通过性能的规则检查，比如 CPU 利用率超过 95%，增加应用节点，减少项目访问的压力，达到根据实时的访问情况，自动做到应用弹性伸缩的功能。

针对本次项目资源监控的测试点，主要是对容器性能监控和容器的弹性伸缩环节。容器性能监控主要实现对容器应用资源的 CPU、MEM 资源利用率的实时采集和分析展示，弹性伸缩主要实现对容器资源伸缩的规则的建立和达到触发条件后应用的弹性扩展和伸缩。

3.3.1 容器性能监控分析

针对本次项目容器的性能监控模块主要有性能采集和性能分析展示功能。采集顾名思义就是对应用的监控指标的实时采集，分析展示主要是对容器性能的指标进行实时的图形化展示，以及对大量应用性能信息的分类汇总分析等。

（1）性能采集

通过 docker stats 采集命令，周期性的对容器性能进行检测，并将信息进行入库的方式实现对性能信息的采集。容器类似于迷你主机，需要对 CPU、MEM 进行长期检测。检测主要通过 docker stats 命令，然后通过周期性的扫描执行，将 CPU、MEM 利用率指标进行入库。以查看应用 nginx 为例，其检测性能执行结果如下：

```
$ docker stats nginx
CONTAINER    CPU%    MEM USAGE / LIMIT    MEM%
nginx    0.12%    4.2MB/1.04GB    0.41%
```

其中，第一列为容器别名，第二列为 CPU 使用率，第三列为已使用内存和已分配的内容，第四列为内存的利用率。

1）docker stats 计算 CPU Percent 的算法如下：

```
cpu_delta = cpu_total_usage - pre_cpu_total_usage;
system_delta = system_usage - pre_system_usage;
CPU % = ((cpu_delta / system_delta) * length(per_cpu_usage_array) )
* 100.0
```

docker daemon 会记录这次读取/sys/fs/cgroup/cpuacct/docker/[containerId]/cpuacct.usage 的值，作为 cpu_total_usage；并记录了上一次读取的该值为 pre_cpu_total_usage；读取/proc/stat 中 cpu field value，并进行累加，得到 system_usage;并记录上一次的值为 pre_system_usage；读取/sys/fs/cgroup/cpuacct/docker/[containerId]/cpuacct.usage_percpu 中记录，组成数 per_cpu_usage_array。

2）docker stats 计算 Memory Percent 的算法如下：

```
MEM USAGE = mem_usage
MEM LIMIT = mem_limit
MEM % = (mem_usage / mem_limit) * 100.0
```

读取/sys/fs/cgroup/memory/docker/[containerId]/memory.usage_in_bytes 的值，作为 mem_usage；如果容器限制了内存，则读取 memory.limit_in_bytes 作为 mem_limit，否则 mem_limit = machine_mem；

（2）性能分析和展示

性能分析和展示主要是对已采集的数据进行汇总分库分表，便于后面图形化展示其小时、天和月表的最值。

由于需要分析汇总展示 CPU 在一定时间内的最大值、最小值，需要每小时、每天、每月对已采集的信息，进行抽离出最大值、最小值。分析汇总可以通过定时任务实现，主要触发时间为小时、天、月度之后的 5 分钟，通过对之前的数据查询，获取到当前应用的最值，记录到以小时、天、月命令的表中，待页面展示汇总数据时，可以查询对应的时间库汇总表。

对于性能的数据展示，借助于前端展示框架 Highcharts 实现对数据的封装图表化。展示主要是将已获取的性能数据，按照应用，在横坐标为时间，纵坐标为利用率的数值，以曲线的形式方便直观的展示。

3.3.2 容器弹性伸缩

容器弹性伸缩功能主要有建立伸缩规则和实现弹性伸缩功能。由于本项目的测试点主要针对应用高低峰时弹性，而判断应用的高低峰需要借助于平台的实时的性能数据，反过来调整资源的分配。

建立伸缩规则主要是对应用预先设置阈值。由于本项目主要使用的 CRM 应用主要的消耗在计算能力上，故相对于建立规则的监控指标主要是 CPU 和内存的使用率。当 CPU 和内存达到最高值时，弹性减少应用个数，同样性能监控指标达到最低值时，弹性扩展应用的个数。当把规则持久化到数据库中后，通过周期性的性能采集入库时，与已建立的规则进行匹配。达到限制条件时，才可触发弹性控制。

实现弹性伸缩主要是对已经建立的 CPU 和内存使用率匹配规则后的处理。比如当监控指标达到最大阈值时，主要采取 Kubernetes 调整 Replication Controller 的配置来增加 Pod 的副本数，从而实时的分担已有应用的访问压力。另外，针对于调度中的事件，需要建立日志记录功能，方便对弹性伸缩事件的检索和日常运维的审计功能。

3.4 灰度发布概要设计

在实际生产经营过程中，生产环境的 7X24 不间断的服务是众多客户期待的需求，毕竟对于业务量密集的企业，1 秒钟的停滞将带来巨大的经济损失。灰度发布一直是软件开发项目版本上线的亮点环节。本项目设计的灰度发布主要是在版本 V1 可用的情况下，同时部署一个版本 V2，在保障整体系统稳定情况下，从版本 V1 整体平滑切换升级到版本 V2。

本项目设计的灰度发布采用了逐步替换策略。在保证系统稳定提供服务的前提下，从版本 V1 升级到版本 V2，可实现 Pod 和 Replication Controller 的完全替换，灰度升级执行的命令如下：

```
kubectl rolling-update web-v1 -f web-rc.yaml --update-period=10s --rollback
```

对于大型的产品化的应用灰度发布，还需要考虑的因素有发布用户的筛选策略，而本次项目测试周期短且紧急度较高，测试人员主要关注的还是无感知的版本升级，故此次的调整重点还是保证切换过程中，避免出现异常的可能性。对于灰度发布的功能，主要需要实现升级版本的确认和版本升级操作。

3.4.1 版本确认

对版本确认是保证应用最终过渡到指定版本的保障环节。版本确认功能包含对要升级版本选择和升级过程中日志审计。

由于本次环节主要是版本确认，即可以认为已存在 V1 版本，否则将会发挥持续集成的功能，发布应用。当开发人员对 V1 版本优化后生成 V2 版本，客户急需对新版本进行验证，需要对其进行灰度升级。但是当版本迭代更新相当频繁时，准确的对指定版本升级变得尤为重要。对版本确认是保证应用最终过渡到指定版本的保障环节。

针对应用的升级版本，本项目主要针对 CRM 页面，版本变更依赖的基础环境变化不大，所以页面设计只需增加选择对应的镜像版本的选项。明确本次升级到的镜像的版本。当选择版本后，点击确定，即可对应用镜像的版本进行灰度升级操作。后台需要对应用的版本升级情况进行维护，前台界面也需要实时展示版本的升级过程。

3.4.2 版本升级

版本升级是继版本确认后对应用进行实际操作的环节。主要有对应用版本的实际操作升级和持久化版本信息到数据库中。

实际操作升级需要维护 Kubernetes 中的 Deploy 发布策略。由于本项目底层语言采用 JAVA 开发，且 Kubernetes 提供了丰富的 REST API 接口，故主要采用基于 HTTP 的请求访问协议，摒弃 Kubernetes 易出错的命令行方式。查阅 API 接口文档，可知请求 Kubernetes 集群的/namespaces/{namespace}/deployments /{name}接口，且对请求参数以 json 的格式进行接收，主要是简化传输过程中的流量消耗。

持久化版本信息到数据库主要是对应用版本信息的维护。首先需要等实际操作升级完成后，对现有版本的挂载镜像版本进行维护。由于版本的升级是逐渐的过程，需要有定时任务去扫描当前应用的实际存活状态；并对当前应用启动和关闭的事件进行详细的记录。通过版本的切换变更，可以观察到升级均是等待新版本运行起来，老版本逐渐退出的过程。

通过对版本的升级，可以完成新版本的切换。任何的版本升级过程，都不可能是完完整整的无感知，应用的启动和停止时间，主要取决于镜像。新旧版本的流量切换，会有短暂的切换效果，但是都是秒级的，基本刷新下页面就可以查看的新版的功能。

传统的灰度发布策略主要是针对 HTTP 入口应用进行灰度，在应用前端配置

Nginx 负载，在负载的过程中根据相应的策略进行灰度选择和请求转发。Http 请求的入口都在 Nginx 上，Nginx 会根据 location 的配置进行 uri 的选择。此时请求数据会判断当前应用是否已经开启灰度，再次判断是应用级别的灰度还是服务级别的灰度，然后根据管控平台配置的灰度策略进行灰度，一般支持白名单、小流量、正则表达式。

但是由于本次入围测试项目周期短和针对性强的性质，主要对容器内资源进行灰度，直接应用内部的灰度策略即可实现。

3.5 日志检索概要设计

日志检索也是软件开发项目的保障性环节。其主要实现了对容器主机日志、应用日志和业务日志的采集、汇总、分析和展示功能。

对于 Docker 内日志而言，其分散性是导致运维难度大的主要原因。常见的采集主要有通过命令 docker logs ,但这种方式不能实时获取日志，存在一定延迟。对于采集，也有通过直采容器内的日志目录方式，例如/var/lib/docker/containers/${ContainerId}/${ContainerId}-json.log。但这种方式随着应用节点的增多维护将变得异常困难。况且还有一个问题，Docker 运行日志是在/var/lib/docker 下的，按照常规主机操作系统文件的分配，/var 一般只会分配 10G 左右的空间，对于大量主机、应用、业务日志来讲肯定是不够用，所以需要对 Docker 运行时的根目录进行设置，避免日志量过大，导致的系统崩溃。

本次项目主要是在采集容器日志目录的基础上，建立一个集中式的采集服务，通过汇总日志，独立的提供日志采集汇总平台。对容器的日志采集和搭建日志平台存储和展示日志成为本环节较为重要的功能。

3.5.1 日志采集

日志采集就是数据获取的过程。对于 Web 平台来说，采集主机的运行日志、应用日志和业务日志非常重要，有助于我们分析当前应用主机的负载情况和应用的健康度进行分析。有的日志可以单独进行分析，而有的日志则需要结合主机上的资源日志进行分析。比如应用出错后，有可能涉及到负载的主机资源调度的连锁影响。

日志采集功能是指对指定的日志目录的文件进行采集记录的功能。主机和应用的日志采集规则相当简单，而对业务日志的采集需要应对匹配的关键字，需要对业务的指定关键字行进行匹配分析，比如包含订单处理过程。另外，日志的来源也需要做详细的记录，比如主机 IP、日志文件的路径。日志采集模块由单独

的 Docker 容器应用复制统一采集和传送至后端存储。

3.5.2 日志分析

采集到的日志数据或者经过安全分析后数据需要进行存储，为方便找到这些数据，需要进行索引的建立，来实现查询和检索功能。

本次项目中日志存储主要采用 Elasticsearch 中的分布式实时文档存储。在存储的基础上增加了类似于传统数据库的索引分层结构，建立索引后，新接收的日志文件写入到对应的索引下。由于对于日志的数据量而言，建立相应的索引，对于页面的检索效率还是有很大影响的。本次项目的建立的索引为关键字+日期，因为增加了日期后，更有助于定位当天的测试订单业务执行情况。

日志的界面展示也是检查日志功能的重要环节。本次项目主要采用了开源的 Kibana 的界面显示，满足常见的界面查询交互。

3.6 本章小结

本章介绍了本文的主题需求和总体的设计技术。首先对当前持续发布中的难点需求进行分析，以及如何架构以 Jenkins 的发布流程进行说明。随后，对使用 Docker 中容器资源管理中资源监控、灰度发布、日志检索模块的实现进行了简单的原理性设计说明。

第 4 章 系统实现

4.1 持续集成

有了总体设计，这个平台的大体框架已经构建成功。现在所要做的工作就是补充每个环节的细节，以及每个细节中应注意的点。本项目中的持续集成主要通过使用 Jenkins 和 Docker 实现，搭建环境的步骤和持续集成流程如下。

4.1.1 持续集成环境搭建

通过自动化技术，可以简化项目的构建和部署工作。而借助于 Jenkins 的持续集成，可以实现应用的快速打包和自动部署。

本项目采用的 SVN 代码管理工具、Jenkins 持续集成工具和 Docker 私有仓库皆部署在局域网中，为解决内网的网络限制问题，需要先获取 Rpm 安装包，获取的主要过程如下：

(1) 缓存 Rpm 安装包

在外网 Linux 主机上设置/etc/yum.conf 下的 keepcache=1，打开下载后保存选项，即可保存安装过程中的依赖 Rpm 安装包。

(2) 从缓存安装对应软件

复制外网 Linux 上/var/cache/yum 下文件到内网对应的主机目录上，Yum 使用-C 参数从缓存中安装对应的软件，如安装 Docker 的命令如下：

```
yum -C install docker
```

获得软件安装包之后，需要对持续集成环境中的 Docker 私有仓库、Jenkins 和 Jenkins 依赖插件进行安装测试，主要的操作步骤如下：

(1) 搭建 Docker 私有仓库

从 Docker 的镜像仓库中下载 registry 镜像，然后应用该镜像启动容器，且指定镜像挂载目录~/registry。执行的命令主要如下：

```
docker run -d -p 5000:5000 -v ~/registry:/tmp/registry registry
```

（2）搭建 Jenkins

使用 Docker 下载 Jenkins 镜像，然后启动容器，且容器启动过程中指定镜像挂载项目代码的目录/var/jenkins_home。执行的命令主要如下：

```
26  docker run -p 8080:8080 -p 50000:50000 -v /home/jenkins_home:/var/
    jenkins_home jenkins
```

8080 端口是 Jenkins 的端口，50000 端口是 master 和 slave 通信端口。之所以使用-v 参数，可以对已构建的项目进行持久化，使项目不会因为容器的消失而导致构建应用的失败。

（3）安装插件

启动项目后，下载所需插件（主要是 JDK 和 Maven Intergration Plugin 插件）。如果缺少 Maven Integration Plugin 插件，在项目创建任务时，将不会有 Maven 选项。

在 Global Tool Configuration 中配置 Maven 和 JDK。配置 JAVA_HOME 和 MAVEN_HOME，分别指向/usr/local/jdk, /usr/local/maven 的安装目录。

（4）配置项目

输入项目名称和选择 Maven project 即可新建项目。当定义项目完成后，点开项目，点击 build now 则立马开始一次构建，通过点击构建信息，可查看构建的明细。

4.1.2 持续集成流程实现

上述的持续集成环境在局域网内搭建成功，基础的运行环境已准备好，这样才能够进行下一步，展开持续集成的流程。

持续集成流程，主要是将指定的 SVN 地址中的项目内容，从 SVN 代码服务器上拉取下来，调用 Jenkins 的自动编译的功能，结合 Docker 镜像发布到容器应用。使应用代码可以达到快速的进行版本升级。

鉴于 4.1.1 节中已经对 Docker 私有镜像仓库和 Jenkins 的基础环境搭建进行了论述，本小节主要论述应用发布的具体实现。整理流程可以分为两步。第一步，开发组长确认要发布的 SVN 项目和镜像，进行一键式发布；第二步为触发集成构建后，通过 Jenkins 自动编译和 Docekr 镜像实现应用版本的升级。

开发组长一键式发布，主要确认的界面元素有镜像名称、代码仓库地址、项目的名称、Dockerfile 等。本次项目开发设计的一键式请求界面如图 4-1 所示。

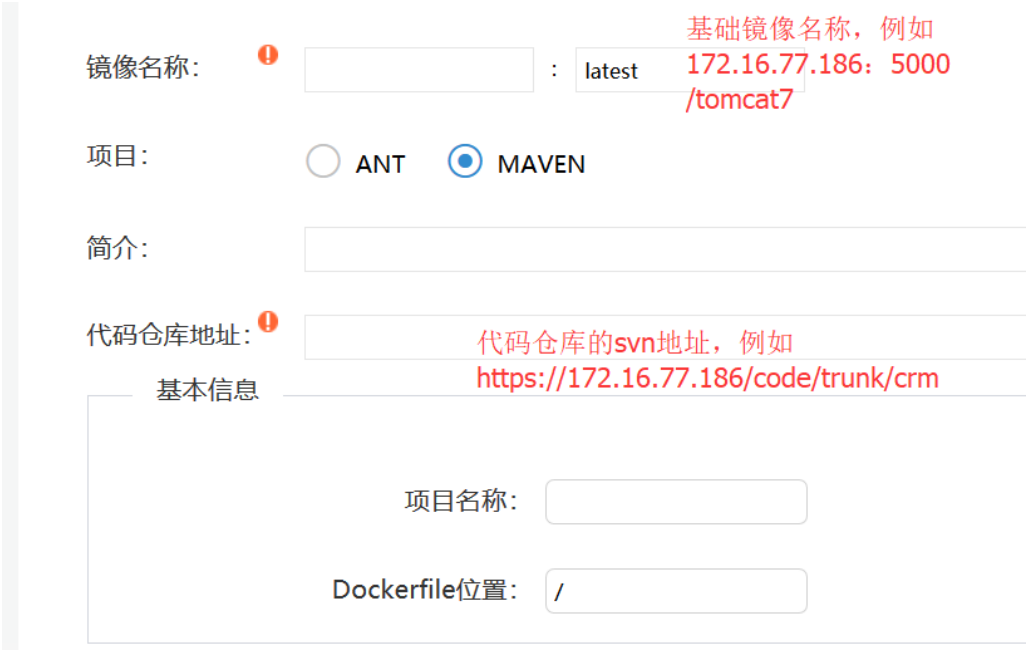

图 4-1 持续集成一键式请求界面

集成构建触发后，需要结合 Jenkins 和 Docker 镜像实现代码服务的发布。在 Jenkins 中创建项目任务，使用 Maven 完成项目的构建和打包。在 Docker 中使用 Docker build 完成项目打包、Docker tag 标记项目镜像包别名和 Docker push 将项目镜像上传至私有的镜像服务器。

根据 Jenkins 与 Docker 结合的流程，实现集成模块需要在 Controller 控制层中添加 CiControler,用于对持续集成的请求控制，Service 业务层需要添加 ICiApp 接口和其接口实现类 CiAppImpl，对于创建持续集成任务，会请求 CiControler 中的 createCi()方法,然后在方法中调用 CiAppImpl 中的 saveCi()方法实现创建过程，还有一些更新 updateCi()、删除 deleteCi()的请求方法，其类图如图 4-2 所示。

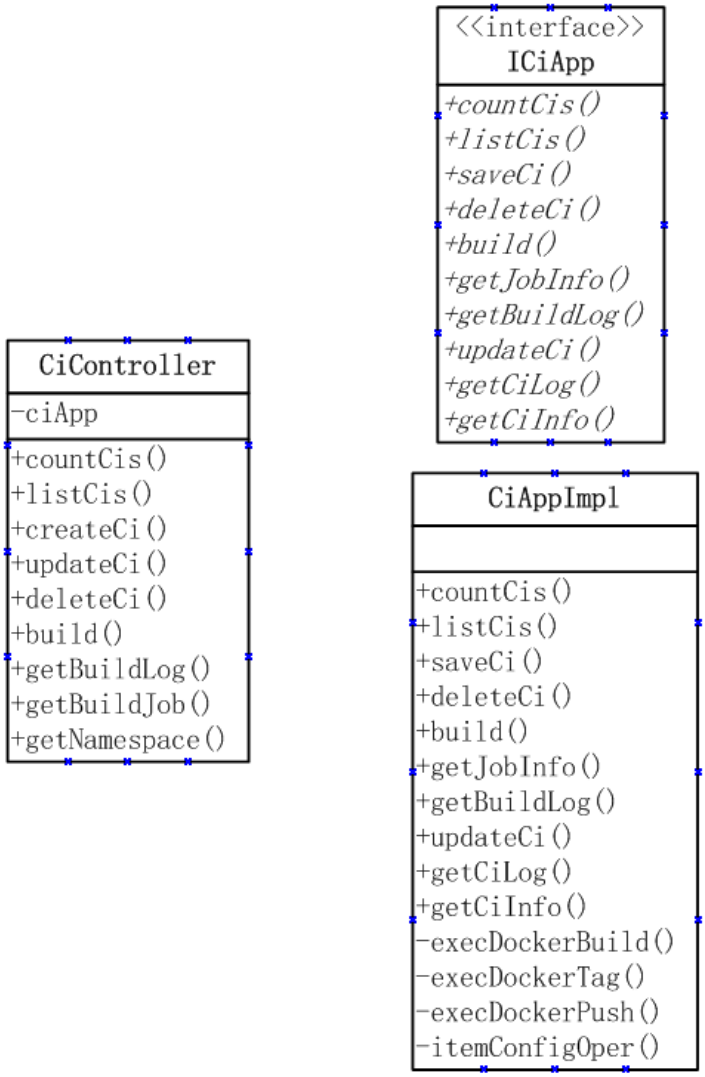

图 4-2 持续集成类图

对项目进程持续集成请求和项目构建，首先需要做镜像 Image 和仓库 SVN 的校验，然后依次组装项目发布的 Docker build、Docker tag 和 Docker push 命令，接着调用 Jenkins 的接口，实现项目的自动集成。实现核心代码如下：

```
// 处理持续集成主流程
public ResponseMessage saveCi(String namespace, CiVO ciVO) {
    // 1.定义返回的结果集
    ResponseMessage mes = new ResponseMessage();
    // 2.镜像名称和代码仓库地址参数校验
    if(StringUtils.isEmpty(ciVO.getImage())||StringUtils.isEmpty(ciVO.getCodeSource()))
        return mes;
    // 3.生成 docker build 命令
    String buildExec = execDockerBuild(namespace ,ciVO);
    // 4.生成 docker tag 命令
```

```
String tagExec = execDockerTag(namespace ,ciVO);
// 5.生成 docker push
String pushExec = execDockerPush(namespace ,ciVO);
// 6.调用 Jenkins 的 createItem 接口，创建项目构建
Response resp = itemConfigOper(ciVO , buildExec ,tagExec, pushExec ,"create");
// 7.保存构建结果
if (response.getStatus() == 200) {
        // Jenkins 创建 job 成功后，即可在系统后台保存该任务
        RmDockerCi rdc = new RmDockerCi();
        rdc.setId(UUIDGenerator.create());
        rdc.setStatus("Create");
        rdc.setCodeSource(ciVO.getCodeSource());
        rdc.setLastBuildTime(new Date());
        rdc.setImage (ciVO.getImage());
        rmDockerCiDao.save(rdc);
        // 返回状态
        mes.setStatus(ResponseMessageStatus.SUCCESS);
    } else {
        mes.setStatus(ResponseMessageStatus.ERROR);
    }
    return mes;
}
```

4.2 资源监控

4.2.1 容器性能监控

由于 3.3.1 节已对 Docker 的采集命令进行了说明，故本节只对采集时的具体的策略进行研究实现。容器的性能监控主要针对容器层面的监控服务，该服务能够为用户提供平台的适用情况和性能数据的实时监控，然后对其进行加工，以图表的形式直观地展现给用户，这些数据包含 CPU、网络、内存和磁盘等指标的使用情况。监控服务的实现需要关注两点，一是架构设计，从全方位的角度设计系统的每个环节，保证数据的采集、传输、处理和存储；二是注重用户体验，将用户的方便使用需求纳入考虑，实现系统的人性化设计。

针对性能数据的实时采集，是通过增加定时任务的实现。由于本次项目主要使用 Spring Quartz 开发，设置其触发器表达式为 0/30 * * * * ?(即每 30 秒)对现有的容器应用执行 Docker stats 命令，收集其 CPU 和内存利用率。

针对性能数据的汇总，从时间上分为三个维度，主要有小时维度、天维度和月维度，也是通过建立定时任务实现，设置其触发器分别为 0 5 * * * ?(即每小时

05 分钟时)，0 5 0 * * ?（每天凌晨 05 分时）和 0 5 0 1 * ?（每月初凌晨 05 分时），主要是针对之前一小时、一天和一月内，各个容器中的性能数据汇总出其最大值、最小值和平均值，然后持久化到相应的性能监控数据表中。

针对性能数据的展示，主要使用 Highcharts 实现。由于 Highcharts 组件已内置了图标展示的常用规则，只需要将性能数据封装到对应的横坐标 xAxis 和纵坐标 series 中即可。项目中展示性能主界面如图 4-3 所示。

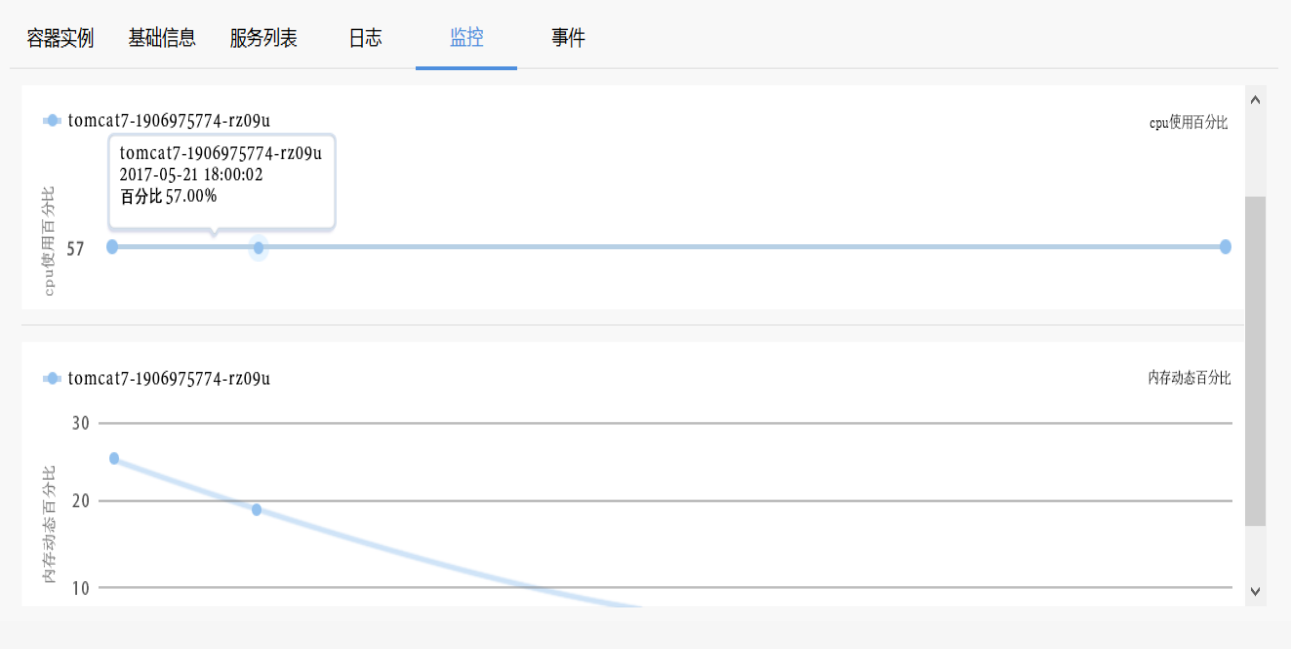

图 4-3 性能界面

图 4-3 主要是 tomcat7 应用的监控界面。横坐标为时间轴，纵坐标分别为 CPU 利用率和内存利用率，界面上显示了某一时刻 CPU 利用率达到 57%。

4.2.2 容器弹性伸缩

容器弹性伸缩主要需要完成规则的建立和规则触发后的弹性伸缩事件。其中规则的建立环节需要对已有的性能数据汇总和分析，以便于对容器整体资源进行调整。

弹性伸缩主要是利用 Kubernetes 的特性，将容器的实时的性能监控 CPU 使用率和内存使用率加入到应用弹性伸缩的策略中，当满足特定的规则时，实现容器的资源动态的自适应调整，做到应用的弹性伸缩自适应。弹性伸缩规则的建立，是基于现有的监控数据。对容器的伸缩设置有三个环节。

第一，建立相应的伸缩组，主要是对伸缩组的名称、资源（需要进行伸缩设置的应用）、最大值（伸缩的上限）和最小值（伸缩的下限）。设置伸缩组的界面如图 4-4 所示。

图 4-4 设置伸缩组

第二，建立策略，主要是对规则触发后单次执行增长或缩减的个数和伸缩一次后的冷却时间（单位为秒）进行设置。设置策略的界面如图 4-5 所示。

图 4-5 设置策略组

第三，建立规则，主要是对具体的性能监控指标设置阈值。设置规则的界面如图 4-6 所示，支持增长或缩减规则设置，加入规则计算的指标有 CPU 和内存，统计方式可以为最大值、最小值、平均值，比较方式可以为大于、大于等于、等于、小于和小于等于，阈值为用户自定义的阈值，周期为统计 CPU 和内存使用率的统计周期，单位为分钟。通过对规则的建立，实现当 CPU 使用率或内存使用率达到阈值时应用个数的自动增长或缩减，达到容器内的资源自适应。

图 4-6 设置规则

规则触发后的弹性伸缩事件，需要对其本身的执行过程有详细的事件记录。规则触发主要在对性能数据采集之后，当判断性能监控指标达到最大值或者最小值后（即规则触发后），通过调用 Kubernetes 中的 Replication Controller 实现容器节点的扩展或伸缩。规格的检测主要在定时任务 AutoScalingStrategy 中触发处理。设计的弹性伸缩的类图如图 4-7 所示，前台界面主要由 ScalingController 控制请求，调用 ScalingAppImpl 中的接口实现，AutoScalingStrategy 是定时任务中调度的实现，例如 doSchedule()是定时伸缩任务的入口。

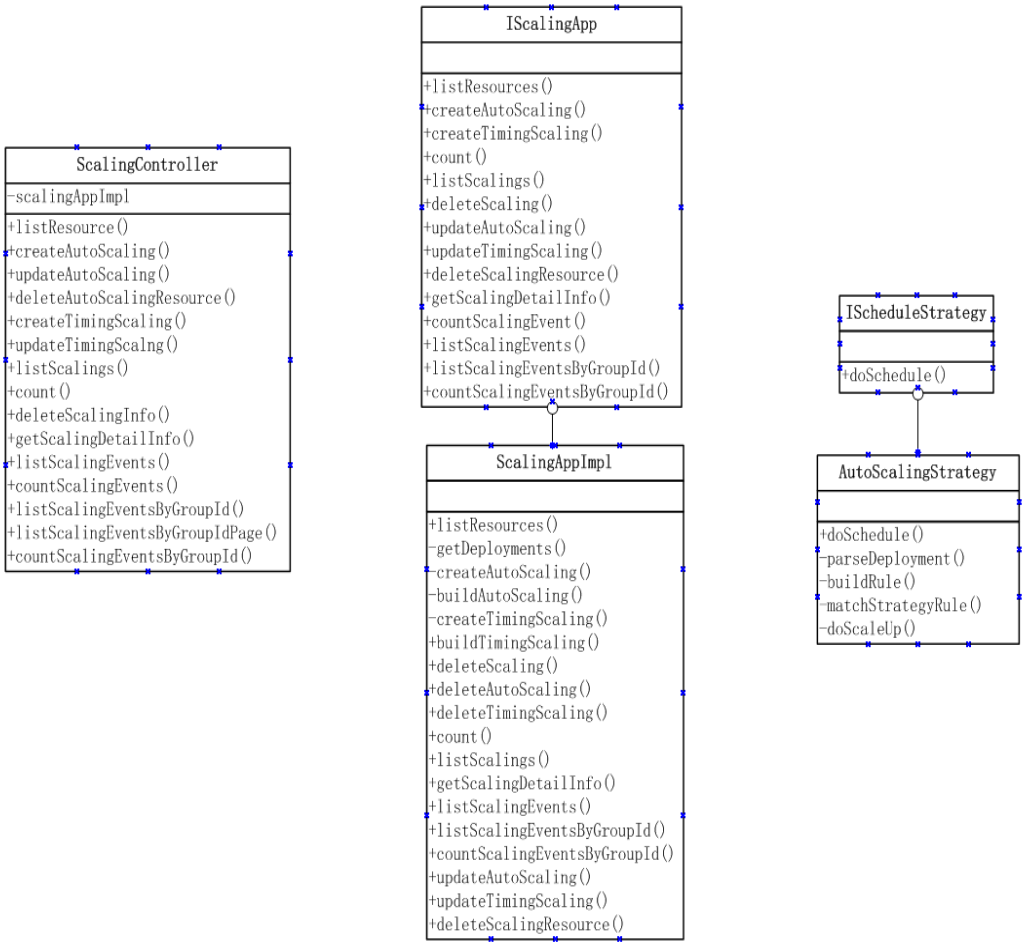

图 4-7 弹性伸缩类图

基于流程开发的弹性伸缩功能，其主要实现代码如下。

```
// 自动伸缩任务
public void doSchedule(AutoScaling autoScaling) {
    // 1.参数异常处理
        if (Optional.ofNullable(autoScaling.getSpec().getResources()).isPresent()) {
            // 2.循环当前设置的规则
            autoScaling.getSpec().getResources().stream().parallel().forEach(r -> {
            try {
              // 2.1 设置 Deployment
              MonitorResourceInfo res = parseDeployment (autoScaling, r);
              // 2.2 绑定规则
              List<StrategyRuleMonitorDataInfo> strate=buildRule(autoScaling, res);
                // 2.2.1  循环判断规则是否满足
                    strategy.stream().parallel().forEach(s->{
                        try {
                            // 2.2.1.1 获取性能数据匹配
                            ScalingInfo scalingInfo = matchStrategyRule(s);
                            // 2.2.1.2 执行伸缩操作
                            doScaleUp( scalingInfo);
```

```
                                } catch (StrategyRuleException e) {
                                    LOGGER.error(e.getMessage());
                                }
                            });
                        } catch (ResourceParseException e) {
                            LOGGER.error(e.getMessage());
                        } catch (MonitorPointException e) {
                            LOGGER.error(e.getMessage());
                        }
                    });
                }

            }
```

与常见的容器弹性伸缩方案相比，本次弹性伸缩引入了实时的性能监控检测，可以结合实际的容器环境，实现资源的自适应调整，而不用简单的依靠历史经验来决定伸缩规则，从而使弹性伸缩可靠并且有依据性。其次，通过配合持续集成，可以实现多环境下同一项目的自适应，实现开发、测试和生产环境的强一致。然后，通过私有镜像仓库和持续集成的环境保证，该项目设计的弹性伸缩同样适用于数据库和缓存的服务实例，而不仅仅是 Web 应用实例。最后，在容器弹性伸缩设计时保留了定时设置弹性伸缩的策略，为周期性的有规律的调整应用预留了接口，并且会对其调度过程中的调度事件进行记录，方便对应用的调度事件的跟踪。

4.3 灰度发布

灰度发布是指在黑与白之间，能够平滑过渡的一种发布方式。传统软件的发布需要经过需求分析，再投入研发。在研发结束后，进行内部测试，测试通过，交由 Beta 进行整合，然后 Beta 版本发布通过测试后投入到正式环境。而这一发布方式存在局限性，容易忽略用户体验，在用户中进行推广时，遇到不能短期修复的问题，则需要软件版本退出流通。而灰度发布通过小规模的用户进行试用能有效避免这一问题，它注重用户体验，可以根据用户体验进行版本修复，至多延长发布期，而不是导致版本的回退。

ABtest 就是一种灰度发布方式，让一部用户继续用 A，一部分用户开始用 B，如果用户对 B 没有什么反对意见，那么逐步扩大范围，把所有用户都迁移到 B 上面来。灰度发布可以整体上保证系统的稳定性，在初始灰度的时候就可以发现、调整问题，以保证其问题的影响度最小。

4.3.1 版本确认

本次项目主要是针对已有应用节点。当对该应用进行灰度版本发布时，需要选择对应的项目版本。灰度升级项目选项如图 4-8 所示。

图 4-8 灰度升级项目选项

当点击确定，后台即开始对应用的版本进行灰度升级。版本升级前，后台会对于当前的版本进行校验，当前应用是否存在，当前升级后的版本是否存在。当确认无误后，才会去执行相应的操作，毕竟对当前环境的自检测和提供版本升级的成功率也是保障系统稳定的重要环节。

4.3.2 版本升级

灰度升级的后台主体实现，依靠 Kubenetes 进行集中管理，主要是通过设置 Deploy 发布的策略对应用版本进行升级和调整。reCreateDeployment 函数主要是在容器副本不存在的情况下，增加应用的副本，否则，调用 update Specified Deployment 函数对现有的容器副本进行调整。

根据版本确认与版本升级流程，实现灰度升级模块需要在 Controller 控制层中添加 DeploymentController,用于对灰度的请求控制，Service 业务层需要添加 IDeploymentApp 接口和其接口实现类 CiAppImpl，灰度升级的主要实现访问为 updateDeployment()更新操作，还有一些删除 deleteDeployment()、分页展示 listDeployments()和 countDeployments()方法，类图如图 4-9 所示。

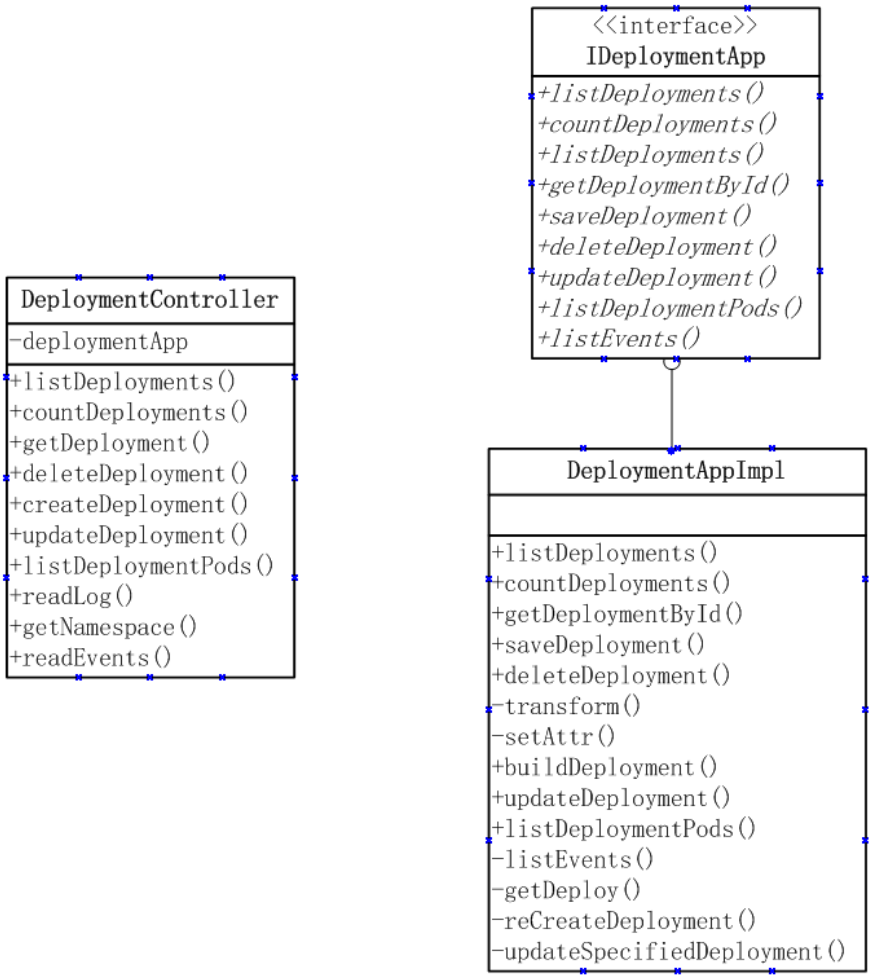

图 4-9 灰度升级类图

对版本升级，需要验证目标版本的有效性，然后构造 Deployment。后台实现主要代码如下：

```
// 灰度发布
public ResponseMessage updateDeployment(String namespace, String deploytId,
    DeploymentSpec spec) {
        // 1.获取原版本
        RmDockerDeployment rmDockerDeployment = getDeploy (deploymentId);
        String deployName = rmDockerDeployment.getName();
        Deployment deploy = new Deployment();
        // 2.获取待升级版本
        String image = null;
        try {
            // 3.根据待升级脚本构造 Deployment
            image = spec.getTemplate().getSpec().getContainers().get(0).getImage();
            DeploymentVO info = listDeploymentPods(namespace, deployId);
            String conName = info.getPods().get(0).getContainers().get(0).getName();
```

```
                spec.getTemplate().getSpec().getContainers().get(0).setName(conName);
            } catch (Exception e) {
                LOGGER.info("未得到应用的版本信息，不对应用版本进行更新");
            }
            // 4.根据是否设置镜像处理
            // 4.1 已设置镜像等参数处理
            if (StringUtils.isNotEmpty(image)) {
              spec.getTemplate().getSpec().getContainers().get(0).setImage(imageUrl + image);
              deploy.setSpec(spec);
            // 4.1.1 灰度升级
              ResponseMessage resm = rollingUpdateDeploy(deploy,namespace, deployName);
            // 4.1.2 结果集返回
              return resm;
            }
            // 4.2 未设置镜像等参数处理
            deploy.setSpec(spec);
            // 4.3 未设置过副本
            if (spec.getReplicas() == -1) {
               // 4.3.1 设置 Deploymen 策略
              ResponseMessage resm = reCreateDeployment(deploy,namespace, deployName);
               // 4.3.2 结果集返回
                return resm;
            }
             // 4.4 更新 Deploy 策略
            ResponseMessage resm = updateSpecifiedDeployment(deploy, namespace,
                                        deployName);
            // 4.5 结果集返回
            return resm;
        }
```

灰度升级过程中，主要是调用 Kubernetes 的 REST API 与 deployments 相关的接口。此处没有采用命令行方式，主要采用了标准的开源软件接口对接的 REST API 接口。针对开源的 Docker 和 Kubernetes，其代码的版本演进相当快，命令行接口的差异性越来越大。一些标准化的开源软件便主张以 REST API 的方式对外暴露接口，以此来屏蔽版本之间的差异性。

当应用进行灰度升级时，后台会根据原有的 Deploy 和用户的输入参数，构造出新的 Deploy 发布策略。假如此应用版本直接是根据镜像升级，则需要在相应的镜像上直接升级到对应的镜像，否则需要查看是否有创建过副本。若没有创建则需要创建新的副本，存在副本的话，需要对现有的副本进行调整。

4.4 日志检索

本次项目关于日志检索主要采用建立一个集中式的采集服务，通过汇总容器挂载在主机指定目录中的日志，并将持续产生的日志持久化存储到 ES 中，通过 Kibana 前端直观的展示。

4.4.1 日志采集

基于 Docker 部署 EFK 非常方便，有各种现成的 image 可用，比如 fluentd-es，实现了 Fluentd 和 Elasticsearch 的集成。其部署的配置文件主要为：

```
apiVersion: extensions/v1beta1
kind: DaemonSet
metadata:
  name: fluentd-es
  namespace: kube-system
  labels:
    k8s-app: fluentd-es
spec:
  template:
    metadata:
      labels:
        k8s-app: fluentd-es
    spec:
      containers:
      - name: fluentd-es
        image: 172.21.3.106:5000/fluentd-es
        resources:
          limits:
            memory: 200Mi
          requests:
            cpu: 250m
            memory: 200Mi
        volumeMounts:
        - name: varlog
          mountPath: /var/log
        - name: varlibdockercontainers
          mountPath: /var/lib/docker/containers
          readOnly: true
        env:
        - name: KUBERNETES_SERVICE_HOST
          value: http://172.21.1.200
        - name: KUBERNETES_SERVICE_PORT
```

```
        value: "8080"
    terminationGracePeriodSeconds: 30
    volumes:
    - name: varlog
      hostPath:
        path: /var/log
    - name: varlibdockercontainers
      hostPath:
        path: /var/lib/docker/containers
```

对于日志的采集 Fluentd 需要配置日志源端和匹配规则，主要是对 source 和 match 的调整。其中 source 主要指向主机上对应日志，而 match 主要是对已匹配的规则传输到 Elasticsearch 中。

4.4.2 日志分析

当数据持久化到容器中，需要借助拥有强大图形展示的 Kibana 把日志直观的展示。从官网下载 Kibana 的 zip 包，解压，运行 bin 下的 kibana.bat，即可启动 Kibana。打开浏览器，键入 http://localhost:5601 即访问 Kibana 内。其中启动 kibana 前需要指定 Elasticsearch 地址。其日志查询建立索引如图 4-10 所示，通过建立索引，即可对当前日志进行展示。

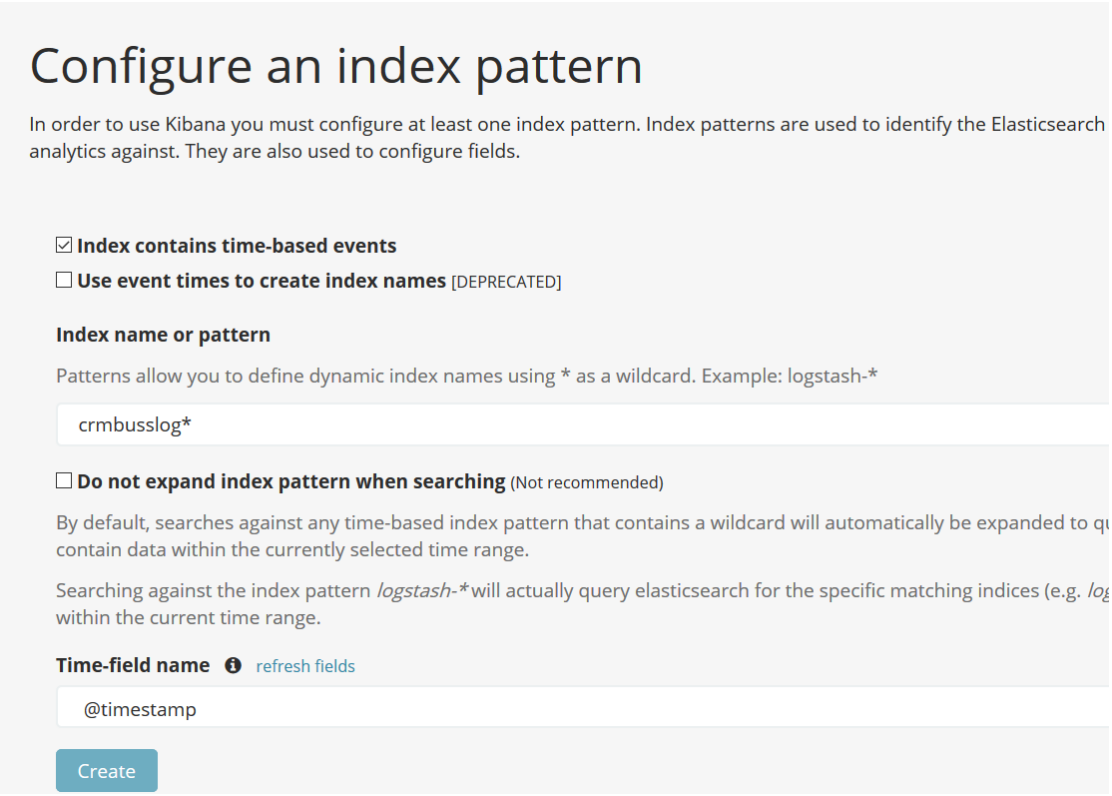

图 4-10 日志查询建立索引

其中由于本次项目主要针对电信用户，故后期对 Kibana 的主题查询界面进行了一定的翻译工作，在测试过程中界面存在相应的调整。相对于日志索引的建立，也引入了日期字段。

日志模块需要在 Controller 控制层中添加 LogController,用于对日志的查询请求控制，类图如图 4-11 所示。

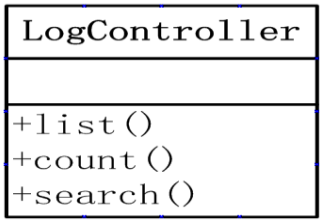

图 4-11 日志模块类图

4.5 本章小结

本章介绍了本文的主要技术的设计和实现。首先搭建了持续发布的平台，对其中的安装部署实施步骤和与 Docker 结合的流程进行了说明。然后，对应用 Docker 中的资源监控，包括性能监控和弹性伸缩模块进行了详细描述，接着对于灰度发布环节的版本确认界面原型和版本升级的原理进行了阐述，最后简述了日志检索模块对于日志采集和存储的方案。

第 5 章 测试与结论

5.1 测试环境

通过第 4 章节的详细设计，对 Docker 的分布式 Web 平台已经有了简单的了解。但是能否能真正应用于实际操作当中，我们仍旧需要对这个平台进行测试，通过一个实验来验证设计的合理性。系统测试是确保系统稳定运行的关键环节。这是系统需求、系统设计和实现的验证。通过使用适当的测试用例，验证系统的功能是否正常实施并成功实施，以避免系统正式投入使用后的质量问题。测试最基本的一个要求就是运行环境，运行环境能够带起来这个平台才是关键，如果运营环境不能达到要求，再好的设计、再完美的软件也可能得不到理想的结论。

本次项目环境测试中,主要有 K8S 的 Docker 底层资源池环境、监控性能定时采集程序和分布式 Web 平台组成，其依赖的主机环境和主机配置如表 5-1 所示。

表 5-1 主机配置

主机名	应用用途	型号、配置描述	操作系统
Web04	EFK/云平台/分布式 Web	8C 32G 100G	Centos7.2
Web05	K8S/后台	2*8c 256G 3*600 G	Centos7.2
Web06	K8S/后台		
Web07	K8S/后台		
Web14	监控采集	8C 32G 100G	Centos7.2
Web15	监控采集		

5.2 测试过程

本系统提供全流程标准化的应用服务，因此，需要对系统进行严格的功能测试，保证系统的良好运行。针对功能的测试是本次项目的成果性检验的环节。根据测试的要求，对实现的持续集成、资源监控、灰度发布、日志检索模块的主要测试过程用例和测试过程如下。

5.2.1 持续集成

持续集成主要是实现对代码环境的一键部署，将指定的 SVN 地址中的项目

内容，从 SVN 代码服务器上拉取下来，调用 Jenkins 的自动编译的功能，结合 Docker 镜像发布到容器应用，使应用代码可以达到快速的进行版本升级。持续集成能够通过代码的迭代尽快发现问题所在，持续集成虽然不能防止代码 bug 的出现，但是它能够将 bug 修复工作变得相对于无持续集成环境要简单和轻松许多。传统软件开发只是在项目结束后进行验收测试，在最后结束阶段发现问题，对于此前的工作则可能更多的是否定，此前的工作将变为无用功。通过日常的代码迭代，能够有效避免最后进行的集中性迭代的繁重工作，也能及早发现问题。持续集成测试用例的要求如表 5-2 所示。

表 5-2 持续集成测试用例

用例编号	1	**用例名称**	持续集成测试用例
测试目的	验证 DID 应用开发和生产环境的一键部署		
预置条件	准备应用发布工具（或平台）、负载均衡环境及相关组件，以及已实施的应用		
例外情况	无	**优先级**	高
测试步骤	1 对指定代码服务器的项目进行构建，并展示构建过程中的日志信息		
期望结果	持续集成满足生产实施要求		

针对持续集成，选择对应的 SVN 代码服务器和匹配的镜像，点击立即构建，后台即可以通过 Jenkins 的接口服务，对指定 SVN 版本上应用获取代码、编译，并借助于 Docker 镜像，完成应用的快速发布。执行成功后，通过查询其构建日志，可以看到整个构建过程中的事件。

持续集成构建界面如图 5-1 所示，项目点击构建后的界面，当构建操作完成后，通过点击该项目的超链接，查看到当前项目的构建结果，如图 5-2 所示，表示项目构建成功，实现了从 SVN 代码服务器上拉取代码，通过 Maven 编译和 Jenkins 持续集成项目资源的部署。通过上述操作，可以证明持续集成功能的有效性。

图 5-1 持续集成操作

图 5-2 持续集成结果

5.2.2 资源监控

资源监控属于可配置模块，资源监控的测试点主要为对容器性能监控和容器的弹性伸缩环节。容器性能监控主要实现对容器应用资源的 CPU、MEM 资源利用率的实时采集和分析展示，弹性伸缩主要实现对容器资源伸缩的规则的建立和达到触发条件后应用的弹性扩展和伸缩。资源监控测试用例的要求如表 5-3 所示。

表 5-3 资源监控测试用例

用例编号	2	**用例名称**	资源监控测试用例
测试目的	验证 DID 应用弹性扩容是否满足生产实施运维要求		
预置条件	准备应用发布工具（或平台）、负载均衡环境及相关组件，以及已实施的应用		
例外情况	无	**优先级**	高
测试步骤	1 对容器的资源性能进行采集 2.对性能信息建立触发规则，达到触发条件时，实现对容器的在线扩缩容		
期望结果	资源监控满足生产实施要求		

针对资源监控，主要是通过对内存的最小值 2，最大值 50 建立规则，当性能监控达到触发条件时，对应用的容器的副本进行调整。执行扩缩容事件的截图如图 5-3 所示。在某一时刻，当主机监控达到内存最大值阈值后，触发系统的扩容规则，应用自动弹性增加一个 1 个节点的调度过程。通过上述操作，可以证明资源监控功能的有效性。

图 5-3 弹性伸缩事件

5.2.3 灰度发布

本次项目主要是针对已有应用节点，当对该应用进行灰度版本发布时，需要选择对应的项目版本。灰度发布主要是实现对容器在线升级。灰度发布测试用例的要求如表 5-4 所示。

表 5-4 灰度发布测试用例

用例编号	3	**用例名称**	灰度发布测试用例
测试目的	验证 DID 应用灰度发布是否满足生产实施运维要求		
预置条件	准备应用发布工具（或平台）、负载均衡环境及相关组件，以及已实施的应用		
例外情况	无	**优先级**	高
测试步骤	1 对应用版本实现灰度升级		
期望结果	DID 应用灰度发布满足生产实施要求		

针对灰度发布，主要是指应用版本的无感知升级。如图 5-4 所示灰度发布，tomcat7-1906975774-rz09u 版本升级到 tomcat7-2830902060-hbyeh 版本，新版本启动运行后，老版本自动停止。通过上述操作，可以证明灰度发布功能的有效性。

图 5-4 灰度发布

5.2.4 日志检索

日志检索主要采用建立一个集中式的采集服务，通过汇总容器挂载在主机指定目录中的日志，并将持续产生的日志持久化存储到 ES 中，通过 Kibana 前端直观的展示。日志检索主要是针对容器的主机、应用和业务日志的检索。日志检索用例的要求如表 5-5 所示。

表 5-5 日志检索测试用例

用例编号	4	**用例名称**	日志检索测试用例
测试目的	验证 DID 应用日志监控是否满足生产实施运维要求		
预置条件	准备好已实施的 DID 应用及日志平台相关组件和页面		
例外情况	无	**优先级**	高
测试步骤	1. 准备客户数据 2. 通过工具持续为客户办理 DID 新建业务 3. 通过重启消息中间件服务、缓存服务、应用服务、构造数据异常等方式，使部分业务处理流程出现异常，部分业务处理成功 4. 登录运维管理平台，进入日志管理界面，通过如下方式查看日志信息： 1) 指定 IP 或服务实例查询日志； 2) 指定时间段查询日志； 3) 指定级别或类型查询日志； 4) 指定业务关键字查询日志； 5) 指定异常关键字查询日志；		
期望结果	日志平台根据条件查询 DID 应用日志实施达到测试目的要求		

针对日志检索，主要是日志的查询。如图 5-5 所示，展示了通过关键字“创建订单”查询指定日期中的日志记录，显示的列项分别有 Time(时间)、_index(索引)、host(主机名)、path(主机上文件目录)、message(消息)，另外也可对展示列项做自定义的设置。通过上述操作，可以证明日志检索功能的有效性。

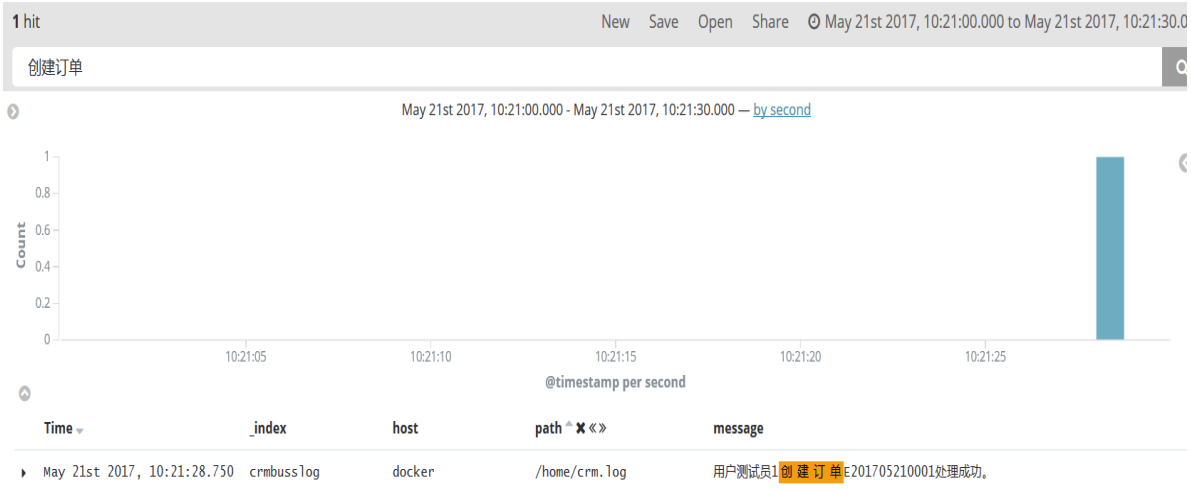

图 5-5 日志检索

5.3 本章小结

本章主要说明通过实践来测试系统功能的可用性以及性能测试，验证本文中既定的持续发布、资源监控、灰度发布和日志检索的需求。每一项需求的测试明确了该测试所要达到的目的、进行测试的预置条件、步骤以及期望得到的结果，并展示了每项需求实际运行结果，每项需求的结果均符合预期。有努力的方向才能得到想要的成果；有了测试环境运行的成果经验，才有可能投入正式的运行环境。

第 6 章 总结与展望

6.1 总结

本文依据上海电信 CRM 云化 POC 测试的需求，针对现在开发、测试和部署过于依赖开发人员，设计了一种基于 Docker 和 Jenkins 的持续集成解决方案。使开发人员不需要参与项目的构建、测试的运行和项目的部署等流程，从而使开发人员更加专注于软件开发环节，间接提高了软件的开发效率，也从而屏蔽了软件开发过程中环境不一致的兼容性问题。构建的基于 Docker 分布式 Web 平台，对容器的资源监控，主要是性能监控和弹性升级的模块进行了实现，另外针对日常应用基于人工发布调度的环节引入了灰度机制，完成平稳的升级。最后对于容器内运行时的主机、应用和业务日志信息进行了采集和存储显示的设计，不仅降低了软件部署成本，还提高了系统资源使用率和降低运维的复杂性。本文主要完成以下工作：

（1）查阅相关文献资料，了解我国当前关于 Docker 分布式 Web 平台现状，然后拟定本文的写作框架。

（2）介绍当前建设分布式平 Web 台需要的技术基础，针对其原理和架设方法进行说明。

（3）结合系统需求展开分析，设计分布式 Web 平台的总体框架。

（4）整合 Jenkins 和 Docker，解决 Jenkins 与 Docker 的内网安装问题，使得 Jenkins 调度任务时能顺利启动容器，实现 SVN 代码管理工具与 Jenkins 持续集成工具的有机集合，提供界面化的一键式持续集成操作方式，节约维护成本和难度。

（5）对容器的 CPU 利用率和内存利用率性能指标建立周期性的采集策略，并通过 Highcharts 直观友好地展示。将容器的资源监控和弹性伸缩灵活的结合，实现基于资源监控的应用自适应调节机制，为容器中的资源监控和弹性伸缩提供了良好的运维方案。

（6）对项目版本进行管理，并在项目实践中引入灰度发布机制，实现升级时业务的不间断。针对容器中的持续升级，提供了友好的界面支撑。

（7）通过 EFK 建立一体化的日志采集和分析平台，它既实现了容器内主机、应用、业务日志的全方位采集和分析，又实现了容器内节点、应用和调度界面化

的资源管理。该平台一方面提供了丰富的查询接口，能够适应日志检索的多维度查询要求；另一方面降低了 Docker 容器资源的运维复杂性，节约了运维人力成本。

（8）完成分布式 Web 平台关于持续集成、资源监控、灰度发布和日志检索功能的编码实现，并针对功能进行逐一测试，验证其有效性，完成基于 Docker 的分布式 Web 平台持续集成、资源监控、灰度发布和日志检索功能的验收。

6.2 展望

本文提出的基于 Docker 的分布式 Web 平台还处于实验测试阶段，功能还需继续完善。

就当前的研究状况，本人认为下一阶段工作应该包括以下几方面：

（1）完善交互界面。当前 Kibana 界面设计不够友好，需要完善交互界面，以及海量日志查询效率优化。

（2）提升应用的通用性。由于当前项目周期短，针对性较强，主要应对于 POC 的入围测试，对建立完全规范的标准的持续集成架构存在一定的差距。

（3）增加弹性伸缩的灵活性。当前弹性伸缩主要考虑了 CPU 和内存的监控指标的影响，针对于大型项目，最好引入更多的监控指标，比如网络、磁盘 IO 和存储，来完善弹性伸缩的策略，增加其调度灵活性。

参考文献

[1] Mell P,Grance T.The NIST definition of cloud computing[J].Communications of the ACM,2011,53(6):50-50.
[2] 刘森.云计算技术的价值创造及作用机理研究[D].浙江大学,2014.
[3] 林利,石文昌.构建云计算平台的开源软件综述[J].计算机科学,2012,39(11):2-6.
[4] 吴朱华.云计算核心技术剖析[M].人民邮电出版社.2011.16-21.
[5] 杨宝龙.基于云计算的软件服务模式的研究[D].北京交通大学,2012.
[6] 林琳,滕腾,李伟彬.Paas 的范畴及架构标准化研究[J].信息技术与标准化,2012(10):27-29.
[7] Fittkau F.Infrastructure-as-a-Service(IaaS) with Amazon EC2/Eucalyptus[J], 2011.
[8] 浙江大学 SEL 实验室.Docker-容器与容器云[M].人民邮电，2016.
[9] 王璞.解析 Docker 如何催生新一代 Paas[J]. 软件和集成电路, 2016(7):74-76.
[10] 卫彪,刘成龙,郭旭.深入浅出 Docker 轻量级虚拟化[J].电子技术与软件工程,2016(10):252.
[11] 佚名.容器技术日渐火热[J]. 软件和信息服务, 2015(9):54-59.
[12] Fowler M,Foemmel M.Continuous integration,2016[EB/OL].http://www. martinfowler.com/articles/continuousIntegration.html,2012.
[13] Duvall P M,Matyas S,Glover A.Continuous integration:improving software quality and reducing risk[M].Pearson Education,2007.
[14] 陶镇威.基于 Jenkins 的持续集成研究与应用[D]. 华南理工大学, 2012.
[15] 王宁.基于 Jenkins 的持续集成系统的设计与实现[D]. 北京邮电大学, 2014.
[16] 钟炜达.一种基于 Docker 的持续集成平台的设计与实现[D].华南理工大学,2016.
[17] 张成.基于 Docker 的持续集成系统的设计与实现[D].苏州大学,2016.
[18] 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.
[19] 刘辉扬.基于 Docker 的容器监控和调度的研究与实现[D]. 华南理工大学, 2016.
[20] 边俊峰.基于 Docker 的资源调度及应用容器集群管理系统设计与实现[D].山东大学,2017.
[21] 何思玫.面向容器云平台的集群资源调度管理器的设计与实现[D].浙江大学,2017.
[22] 周佳威.Kubernetes 跨集群管理的设计与实现[D].浙江大学,2017.
[23] 徐江生.容器云平台的设计与实现[D]. 北京邮电大学, 2017.
[24] Docker[EB/OL].https://docs.Docker.com/.
[25] GitHub[EB/OL].https://github.com/moby/moby.
[26] Boettiger C.An introduction to Docker for reproducible research[J].Acm Sigops Operating Systems Review,2015,49(1):71-79.
[27] TurnbullJ. The Docker Book: Containerization is the new virtuaIization[M]. James Turnbull, 2014.
[28] Docker Remote API[EB/OL].https://docs.Docker.com/engine/reference/api/ Docker_remote_api/.
[29] 汪恺，张功萱，周秀敏.基于容器虚拟化技术研究[J]. 计算机技术与发展，2015(8):138-141.
[30] Jenkins[EB/OL]. https://Jenkins.io/.

[31] 陈婧欣. 基于 Hudson 的持续集成方案的研究与实践[D]. 东北师范大学, 2011.
[32] Kubernetes[EB/OL]. https://Kubernetes.io/.
[33] Red Hat.Openshift-2.0-User_Guide-en-US[EB/OL].https://access.redhat.com/site/documentation.2013.
[34] Documents of etcd[EB/OL].https://github.com/coreos/etcd.
[35] 龚正,吴治辉,王伟,崔秀龙,闫健勇.Kubernetes 权威指南[M].电子工业出版社,2016.
[36] 赵鸣.日志管理和分析系统的设计与实现[D]. 大连理工大学, 2006.
[37] Fluentd[EB/OL]. https://www.fluentd.org/architecture.
[38] Elasticsearch[EB/OL]. https://www.elastic.co/products/elasticsearch.
[39] Kibana[EB/OL].https://www.elastic.co/cn/products/kibana.
[40] 张力文.基于 Jenkins 的项目持续集成方案研究与实现[D].西南交通大学,2017.

致谢

人们常感叹时光的无情，可我却对时光有另外一番感悟“道是无情却有情”。在上海三年的研究生学习和生活让我领略到知识的无限和生活的美好。

我的导师陆黎明教授勤奋工作的态度、严谨治学的品格、细致待人的作风深深地影响着我，引领着我。当我要完成研究生生涯中最后的一个环节撰写毕业论文时，我曾经迷茫过烦躁过，真正能够让我沉寂下来伏案埋头思考系列问题的还是陆老师给我的启迪，每当我在论文写作过程中感到纠结和无奈的关键时候，老师的指导总能让我恍然顿悟。在陆老师的帮助下我确定了论文题目，明确了研究方向，确定了研究内容。

“师者，所以传道授业解惑也”，三年的时光学院的各位老师给予了我知识的营养、生活的关照。子曰：“三人行必有我师焉”，同一师门三年，师门兄弟姐们亲如一家，相互之间互通有无、投桃报李。一个班级的同学们，也是亲密无间、不分彼此、大爱相拥，这份情谊我将终生受用。

“感谢”一词人们经常挂在嘴边，在这里我要将它更改为“感恩”二字，感恩老师、感恩朋友、感恩同学、感恩父母！

2018 年 5 月

论文独创性声明

本论文是我个人在导师指导下进行的研究工作及取得的研究成果。论文中除了特别加以标注和致谢的地方外，不包含其他人或机构已经发表或撰写过的研究成果。其他同志对本研究的启发和所做的贡献均已在论文中做了明确的声明并表示了谢意。

作者签名：李志盼　日期：2018.6.1

论文使用授权声明

本人完全了解上海师范大学有关保留、使用学位论文的规定，即：学校有权保留送交论文的复印件，允许论文被查阅和借阅；学校可以公布论文的全部或部分内容，可以采用影印、缩印或其它手段保存论文。保密的论文在解密后遵守此规定。

作者签名：李志盼　导师签名：[illegible]　日期：2018.6.1