申请上海交通大学工程硕士学位论文

基于微服务架构的智慧社区应用服务与资源管理系统

|  |  |
| --- | --- |
| 学校代码： | 10248 |
| 作者姓名： | 李会一 |
| 学 号： | 1130379067 |
| 导 师： | 饶若楠 |
| 学科专业： | 软件工程 |
| 答辩日期： | 2015年 1月 14日 |

上海交通大学软件学院

2014年 12月

A Dissertation Submitted to Shanghai Jiao Tong University

for Master Degree of Engineering

Application Service and Resource Management for Smart Community System based on Micro-service Architecture

|  |  |
| --- | --- |
| University Code： | 10248 |
| Author： | Huiyi Li |
| Student ID： | 1130379067 |
| Mentor： | Ruonan Rao |
| Field： | Software Engineering |
| Date of Oral Defense： | 2015.1.14 |

School of Software

Shanghai Jiaotong University

Dec., 2014

上海交通大学

学位论文原创性声明

本人郑重声明：所呈交的学位论文，是本人在导师的指导下，独立进行研究工作所取得的成果。除文中已经注明引用的内容外，本论文不包含任何其他个人或集体已经发表或撰写过的作品成果。对本文的研究做出重要贡献的个人和集体，均已在文中以明确方式标明。本人完全意识到本声明的法律结果由本人承担。

学位论文作者签名：

日期： 年 月 日

上海交通大学

学位论文版权使用授权书

本学位论文作者完全了解学校有关保留、使用学位论文的规定，同意学校保留并向国家有关部门或机构送交论文的复印件和电子版，允许论文被查阅和借阅。本人授权上海交通大学可以将本学位论文的全部或部分内容编入有关数据库进行检索，可以采用影印、缩印或扫描等复制手段保存和汇编本学位论文。

**保密**□，在 年解密后适用本授权书。

本学位论文属于

**不保密**☑。

（请在以上方框内打“√”）

学位论文作者签名： 指导教师签名：

日期： 年 月 日 日期： 年 月 日

基于微服务架构的智慧社区应用服务与资源管理系统

# 摘 要

智慧社区是指充分利用物联网、云计算、移动互联网等新一代信息技术的集成应用，为居民、居委会、业委会、物业和其他服务机构等社区相关人员提供的一种智慧化的生活环境和新的社区管理形态。目前在同类型系统中，主要存在以下问题：1）系统内社交模型简单，与Nextdoor等新兴邻里社交模式相比，缺少实名制邻里社交机制，从而导致邻里间信任度降低。2）应用服务间耦合严重，随着应用范围以及复杂度的增加，扩容以及更新流程复杂度大大增加；3）系统计算、存储、网络资源利用率低，没有一个很好的资源管理平台进行统一管理。

针对上述问题，本文以实验室承担的智慧邻里社区服务软件系统研发实际项目为背景，在对微服务系统架构、即时通讯系统、统一资源管理平台以及服务治理等相关技术进行深入分析研究的基础上，设计并实现了一个基于微服务架构的应用服务与资源管理系统。该系统在提供基于实名制的邻里社交系统的基础上，利用服务治理系统与资源管理系统两大关键系统的结合解决了应用服务耦合严重以及平台资源利用率低的问题。测试及应用表明，该系统是可行及有效的。

与其他类似系统相比，本文的工作具有以下的特点：

1. 对邻里社交模型进行抽象，设计并实现实名制邻里社交平台，该系统包括即时通讯、用户状态管理、应用消息推送三大服务，并实现提供易于使用的应用接口，简化外部服务的集成复杂度。
2. 智慧社区系统中需要使用统一资源管理系统对于诸如MongoDB，Tomcat Server，Storm等始终运行的长服务进行管理，而Yarn作为目前广泛使用的统一资源管理平台，缺乏对于这类长服务的支持。因此本文在Yarn基础上对其服务注册、日志管理、容器资源动态变更三个方面进行改进，同时提供客户端接口，在支持长服务的同时简化服务的运行时管理难度。
3. 社区应用服务包括生活服务、物业管理、党建服务等居民生或中的方方面面，而微服务架构的低耦合、易变更等特性对于整个系统的开发维护都提供了极大的便利，因此本文结合微服务架构以及服务治理框架Dubbox，实现服务注册、发现、监控、调用统计、容量评估等功能，并通过监控获得的服务状态与资源管理平台进行集成，对服务进行自动水平扩展。从而实现对应用服务的综合管理。

关键词：智慧社区，资源管理，服务治理，邻里社交

APPLICATION SERVICE AND RESOURCE MANAGEMENT FOR SMART COMMUNITY SYSTEM BASED ON MICRO-SERVICE ARCHITECTURE

# ABSTRACT

As a new concept of modern community governance. Smart Community is the representation for integrated applications using internet of things, cloud computing and mobile internet technologies. It aims to provide a safer, more comfortable and convenient modern living environment for community residences. The paper is on the background of smart community governance system of a city, the system aims to provide a unified resource management and service governance platform for all micro-services. So far, there exists some problems among similar systems: 1) Application services have a characteristic of strong coupling, along with applications scope and complexity augmentation, difficulty for scaling and adjusting such services will be substantially increased; 2) Compute, storage and network resources utilization are in low utilization because there is no unified resource management system; 3) Due to missing system model of neighborhood social network, there are barriers to user communication within the system.

Aim at these problems, we first lucubrate micro-service system architecture, unified resource management system, service governance technique and other related technologies. And propose a solution for application service and resource management based on micro-service architecture to solve the service coupling and low resource utilization problem. Test and application indicates that our system is feasible and effective.

Compared with other similar systems, our work has the following features:

1. The paper takes the open source unified resource management system Apache Yarn into a full analysis and improve service registration, log management, container resource adjustment aspects of Yarn. With these improvements, Yarn was able to host long-lived services such as a pool of tomcat servers and Apache Hbase. Meanwhile, we provide client interface for clients or outside systems to interact with those Yarn deployed services.
2. The paper the micro-service architecture and service governance framework Dubbox, designs and implements functionalities such as service registration, discovery, monitor, invoking statistics, capacity evaluation. Other than that, we make use of the monitor information retrieved and integrates with unified resource management system to auto scaling services.
3. Abstract the neighborhood social network model, design and implement the platform which includes instance messaging, user status management, push notification modules. And provides easy for access interface for outside systems or services.

**Keywords:** Smart Community, Resource Management, Service Governance, Neighborhood Social Network

**目 录**

[摘 要 I](#_Toc437092489)

[ABSTRACT III](#_Toc437092490)

[1 绪 论 1](#_Toc437092491)

[1.1 研究背景及意义 1](#_Toc437092492)

[1.2 国内外研究现状 1](#_Toc437092493)

[1.3 研究目标和内容 1](#_Toc437092494)

[1.4 本文章节安排 1](#_Toc437092495)

[2 相关技术分析 2](#_Toc437092496)

[2.1 容器化技术 2](#_Toc437092497)

[2.2 资源管理平台 2](#_Toc437092498)

[2.3 微服务软件系统架构 2](#_Toc437092499)

[2.4 即时通讯协议 2](#_Toc437092500)

[2.5 本章小结 2](#_Toc437092501)

[3 系统需求分析与架构设计 3](#_Toc437092502)

[3.1 业务分析 3](#_Toc437092503)

[3.2 系统需求分析 4](#_Toc437092504)

[3.2.1 功能性需求 4](#_Toc437092505)

[3.2.2 非功能性需求 5](#_Toc437092506)

[3.3 系统架构设计 7](#_Toc437092507)

[3.3.1 技术架构 7](#_Toc437092508)

[3.3.2 逻辑架构 9](#_Toc437092509)

[3.3.3 部署视图 11](#_Toc437092510)

[3.3.4 系统实现环境 11](#_Toc437092511)

[3.4 本章小结 13](#_Toc437092512)

[4 系统核心功能模块及关键技术研究与实现 14](#_Toc437092513)

[4.1 资源管理服务子系统 14](#_Toc437092514)

[4.1.1 服务发现与注册 15](#_Toc437092515)

[4.1.2 日志管理 19](#_Toc437092516)

[4.1.3 容器资源可伸缩性 20](#_Toc437092517)

[4.2 邻里社交子系统 23](#_Toc437092518)

[4.2.1 邻里社交模型 23](#_Toc437092519)

[4.2.2 用户在线状态管理 23](#_Toc437092520)

[4.2.3 社交消息缓存、推送机制 23](#_Toc437092521)

[4.3 基于微服务的社区管理子系统 23](#_Toc437092522)

[4.3.1 服务划分及容器化 23](#_Toc437092523)

[4.3.2 服务注册与发现 23](#_Toc437092524)

[4.3.3 服务治理 23](#_Toc437092525)

[4.4 本章小结 23](#_Toc437092526)

[5 系统测试及应用 24](#_Toc437092527)

[5.1 系统测试 24](#_Toc437092528)

[5.1.1 功能测试 24](#_Toc437092529)

[5.1.2 性能测试 24](#_Toc437092530)

[5.2 系统应用 24](#_Toc437092531)

[5.2.1 运行实例 24](#_Toc437092532)

[5.2.2 应用效果及分析 24](#_Toc437092533)

[5.3 本章小结 24](#_Toc437092534)

[6 总结与展望 25](#_Toc437092535)

[6.1 本文工作小结 25](#_Toc437092536)

[6.2 展望 25](#_Toc437092537)

[攻读学位期间发表的学术论文 36](#_Toc437092538)

# 绪 论

## 研究背景及意义

智慧社区是指充分利用物联网、云计算、移动互联网等新一代信息技术的集成应用，为居民、居委会、业委会、物业和其他服务机构等社区相关人员提供的一种智慧化的生活环境和新的社区管理形态。随着智慧社区应用在社会进步与发展过程中作用的不断提升，其信息化运营与管理成本也在社会经济中产生了重要的影响。面对智慧社区中海量应用所造成的巨大的系统集成、运维成本，建立统一、高效的应用、资源管理方案是降低运营成本，提高服务质量的迫切需要。

本文以实验室承担的智慧邻里社区服务软件系统研发实际项目为背景，该系统建立了统一智慧社区应用平台，将广泛的社区应用，如居民生活服务、居委会工作服务、物业服务、党建服务、便民便利服务等进行整合，形成完整的社区服务闭环。同时为社区治理中的生活群众、政府管理群众、服务群众提供有价值的服务。例如，社区内的居民可以通过系统内邻里社交、物业、生活（外卖、洗衣、超市、家政等）、二手市场、社区公告等服务中获得生活便利；社区中政府管理群众可通过系统内党建、人事管理、社区公告、活动、邮件等服务中增加与居民沟通渠道，提升社区管理服务质量；社区内服务群众可通过系统内便民便利服务平台发布自身服务，增加自身营收。

然而， 在构建智慧社区治理系统过程中，遇到了以下的难题：1）应用服务间耦合严重，随着应用范围以及复杂度的增加，扩容以及更新流程复杂度大大增加；2）系统计算、存储、网络资源利用率低，没有一个很好的资源管理平台进行统一管理；3）缺乏统一的邻里社交模型，系统内用户交流存在壁垒。因此，如何建立统一、高校的应用、资源管理方案，成为一大问题。

## 研究目标和内容

本文以实验室承担的智慧邻里社区服务软件系统研发实际项目为背景，在对微服务系统架构、即时通讯系统、统一资源管理平台以及服务治理等相关技术进行深入分析研究的基础上，设计并实现了一个基于微服务架构的应用服务与资源管理系统。该系统在提供基于实名制的邻里社交系统的基础上，利用基于Dubbox的服务治理系统以及基于Apache Yarn的资源管理系统两大关键系统的结合解决了应用服务耦合严重以及平台资源利用率低的问题。并通过测试及试运行应用，以验证该系统的可行性及有效性。

围绕上述的研究目标，本文的研究内容主要有：

1. 实名制邻里社交系统：
   1. 分析并建立基于实名制的邻里社交模型，对社区中的管理、生活、服务群众的社交模式进行抽象；
   2. 建立即时通讯系统，支持点对点、群组等社交方式；
   3. 实现用户状态管理服务，支持对用户或外接设备在线状态的管理；
   4. 实现应用消息推送服务，支持聊天消息、应用广告等信息推送；
2. 实现基于Apache Yarn的支持“长服务”的统一资源管理系统：
   1. 改进Yarn自身基于jobId的服务管理机制，通过服务发现注册机制扩展服务管理的功能及灵活性。解决由于节点迁移后导致的整体服务不可用的问题，并提供更多的协议支持；
   2. 改进Yarn对于服务日志的管理机制，通过重写日志管理器及客户端API，支持在服务运行过程中对日志的管理，并支持日志管理策略的配置；
   3. 实现Yarn中Container资源动态变更功能，使Yarn能够在不重启Container的情况下对Container资源进行变更，从而增加服务的可用性；
3. 基于微服务架构的社区管理系统：
   1. 对智慧社区应用进行抽象划分，微服务化；
   2. 基于服务治理框架Dubbox，实现服务发现、注册、监控调用统计、容量评估等功能，对服务进行管理。
   3. 与统一资源管理系统集成，以服务监控信息为依据，按照一定策略对服务进行自动水平扩展。
4. 系统验证及测试：通过系统测试及试运行应用，验证该系统的可行性及有效性。

## 本文章节安排

本文一共分为6个章节，各章节内容安排如下：

第一章，绪论。本章阐述了本文的研究背景及意义，目的与意义，明确了本文的研究目标及内容，对本文将要讨论的内容进行总揽。

第二章，国内外研究现状及分析。本章从即时通讯服务协议、统一资源管理平台、微服务软件系统架构、服务治理等方面总结国内外研究现状，并对其优缺点进行分析与研究。

第三章，系统需求分析与架构设计。本章首先对系统进行了需求分析，包括业务分析、功能性需求分析以及非功能性需求分析。然后从系统的技术架构、逻辑架构、部署视图以及系统实现环境等四个方面进行了系统架构设计。

第四章，系统核心功能模块及关键技术研究与实现。本章重点介绍了系统核心功能模块的设计与实现，包括邻里社交系统，基于微服务的服务治理系统以及支持长服务的统一资源管理系统。

第五章，系统测试及应用。本章首先对系统进行了功能测试和性能测试，然后设计系统运行实例，并对系统应用效果进行分析。

第六章，总结与展望。本章对本文内容进行总结，并对后续工作提出展望。

# 国内外研究现状及分析

## 资源管理平台

在目前的复杂的企业解决方案中，所有的服务器按照业务逻辑被拆封成若干个集群，这些集群各自运行着相对独立的业务，并由专门的运维人员管理。

然而，由于集群中应用特点的不同，它们的资源利用率可能不一致。例如，同一时间段内有的集群业务繁忙，资源利用率高，甚至可能出现资源排队等待的情况，而另外一些集群则处于业务空闲，资源利用率极低的状态。此外，多个集群也可能导致数据冗余度增加，例如一个集群中的业务若需要另一集群中的数据，可能需要跨集群拷贝数据，进而占用更多的磁盘空间。因此，为了解决多集群所带来的问题，如Borg（Google）， Yarn（Apache，Hadoop项目下一个分支，开源），Mesos（Twitter，开源），Corona（Facebook，开源）这一类统一资源管理系统相继出现。概括起来，这类系统主要解决以下两类问题：

1. 提高集群资源利用率

在大数据时代，为了存储和处理海量数据，需要规模较大的服务器集群或者数据中心，一般说来，这些集群上运行着数量众多类型纷杂的应用程序和服务，比如离线作业，流式作业，迭代式作业，crawler server，web server等，传统的做法是，每种类型的作业或者服务对应一个单独的集群，以避免相互干扰。这样，集群被分割成数量众多的小集群，一部分集群运行Hadoop，一部分运行Storm，一部分运行Spark，一部分运行web server，然而，由于不同类型的作业/服务需要的资源量不同，因此，这些小集群的利用率通常很不均衡，有的集群满负荷、资源紧张，而另外一些则长时间闲置、资源利用率极低，为了提高资源整体利用率，一种解决方案是将这些小集群合并成一个大集群，让它们共享这个大集群的资源，并由一个资源统一调度系统进行资源管理和分配，这就诞生了Borg，YARN，Mesos，Torca，Corona。从集群共享角度看，这类系统实际上将所有硬件资源抽象成一个台大型计算机，供所有用户使用。

1. 服务自动化部署

一旦将所有计算资源抽象成一个“大型计算机”后，就会产生一个问题：各种不同的服务如何进行部署？同样，Borg/YARN/Mesos/Torca/Corona一类系统需要具备服务自动化部署的功能，需要系统能够支持从简单的web server到诸如Spark，Storm等复杂的分布式计算框架的自动部署，管理。因此，从服务部署的角度看，这类系统实际上是服务统一管理系统，这类系统提供服务资源申请，服务自动化部署，服务容错等动能。

1. 动态调整切分资源，增强系统扩展性

由于各类服务特征的不同，例如离线服务大都在夜间启动运行，对资源需求高，而传统应用服务则在日间活跃，导致系统对资源的规划很难一次性准确或稳定不变。因此，需要此类系统能够支持对已部署服务资源的动态调整与切分。增强系统的扩展性。

### Apache Yarn

Yarn是开源项目Apache Hadoop的一个资源管理系统，最初设计是为了解决Hadoop中MapReduce计算框架的资源管理问题，但经过不断演化，它已经是一个更加通用的资源管理系统，可将MapReduce计算框架作为一个应用程序运行在Yarn系统之上，通过Yarn来管理资源。如果你的应用程序也需要借助Yarn的资源管理功能，你也可以实现Yarn提供的编程API，将你的应用程序运行于Yarn之上，将资源的分配与回收统一交给Yarn去管理，从而大大简化资源管理功能的开发。目前，已有诸如Storm，Spark，Tez，HBase分布式计算框架或存储系统成功运行与Yarn之上。Yarn生态圈如图2-1所示：

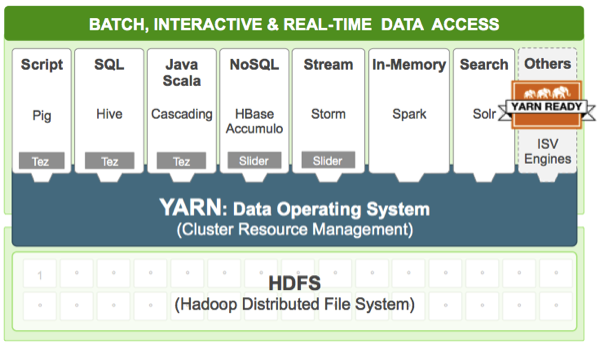


图 2‑1 Yarn生态圈环境

Figure ‑ Ecosystem for Yarn related systems

Yarn是基于Master/Slave模式的分布式架构，具体架构图如图2-2所示：

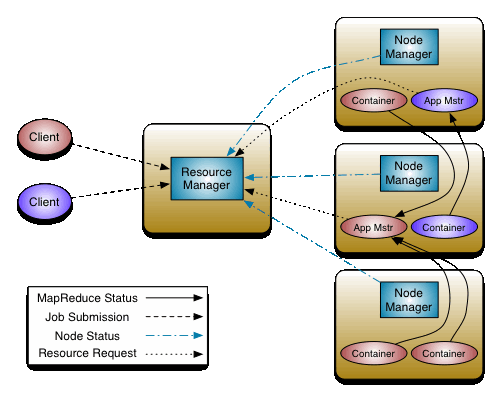


图 2‑2 Yarn架构图

Figure ‑ Architecture of Yarn

上图，从逻辑上定义了Yarn系统的核心组件和主要交互流程，各个组件说明如下：

1. Yarn Client：

Yarn Client提交Application给RM，它会首先创建一个Application上下文对象，并设置AM必须的资源请求信息，然后提交给RM。Yarn Client可以与RM通信，从而获得一个已经提交并运行的Application的状态信息。

1. Resource Manager（RM）：

RM是Yarn集群的Master角色，负责管理整个集群的资源，并对Application进行资源分配。RM作为集群资源的管理和调度角色，如果出现单点故障的情况，则整个集群的资源都无法使用。（这一问题已由2.4.0版本中新增的RM HA方案解决）

1. Node Manager（NM）：

NM是Yarn集群中的Slave角色，是集群中拥有实际资源的工作节点。我们提交Application之后，会将组成Application的多个Component调度到对应的NM上执行。

1. Container

Container是Yarn集群中资源的抽象，将NM上的资源进行量化，根据需要组装成一个个Container，然后服务于已授权资源的计算任务。计算任务在完成计算后，系统会回收资源，以供后续计算任务申请使用。目前，Container包含两种资源：内存和CPU，后续版本中会陆续增加硬盘、网络等更多的资源选项。

1. Application Master（AM）

AM主要管理和监控部署在Yarn集群上的Application，以MapReduce为例，MapReduce Application是一个用来处理MapReduce计算的服务框架程序，为用户编写的MapReduce程序提供运行时支持。通常我们在编写一个MapReduce程序中可能包含多个Map Task和Reduce Task，而各个Task的运行管理与监控都是由这个MapReduce Application来负责，比如运行Task的资源申请，由AM向RM申请；启动/停止NM上某Task的对应的Container，都有AM向NM请求来完成。

### Apache Mesos

Mesos是Apache下的开源分布式资源管理框架，它被称为是分布式系统的内核。Mesos最初是由加州大学伯克利分校的AMPLab开发的，后在Twitter、Airbnb、Netflix等公司得到广泛使用。

**Mesos架构**：

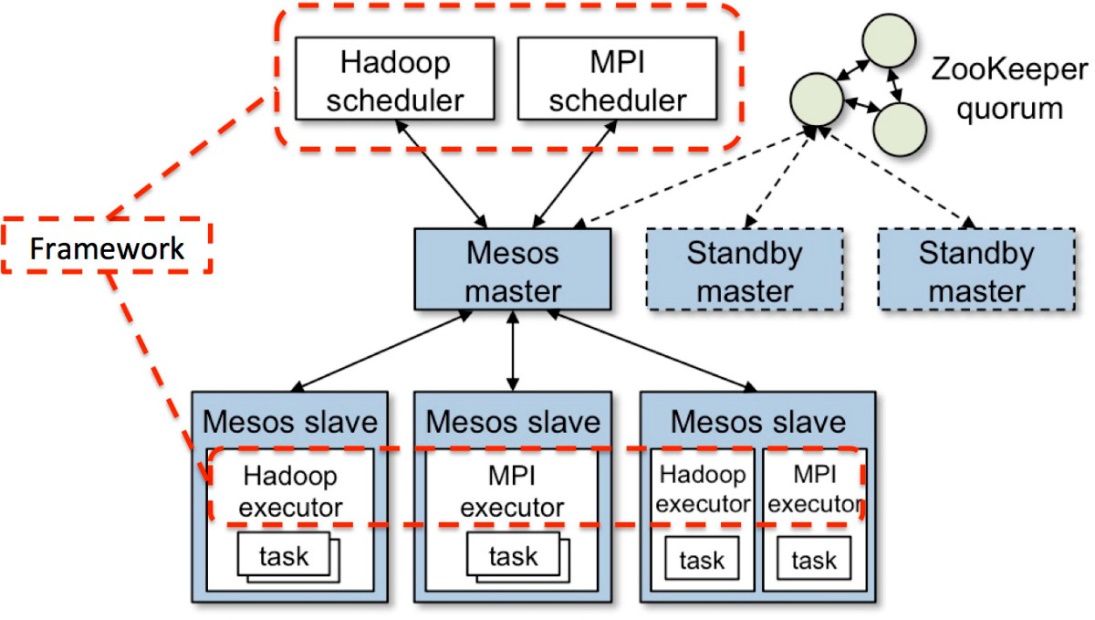


图 2‑3 Mesos架构图

Figure ‑ Architecture of Apache Mesos

如图2-3所示，Mesos实现了两级调度架构，它可以管理多种类型的应用程序，第一级调度是Master的守护进程，管理Mesos集群中所有节点上运行的Slave守护进程。集群由物理服务器或虚拟服务器组成，用于运行应用程序的任务，比如Hadoop和MPI作业。第二级调度由被称作Framework的“组件”组成。Framework包括调度器（Scheduler）和执行器（Executor）进程，其中每个节点上都会运行执行器。Mesos能和不同类型的Framework通信，每种Framework由相应的应用集群管理。上图仅展示了Hadoop和MPI两种Framework。

Mesos Master协调全部的Slave，并确定每个节点的可用资源，聚合计算跨节点的所有可用资源的报告，然后向注册到Master的Framework（作为Master的客户端）发出资源邀约。Framework可以根据应用程序的需求，选择接受或拒绝来自master的资源邀约。一旦接受邀约，Master即协调Framework和Slave，调度参与节点上任务，并在容器中执行，以使多种类型的任务，比如Hadoop和Cassandra，可以在同一个节点上同时运行。

**Mesos资源分配**：

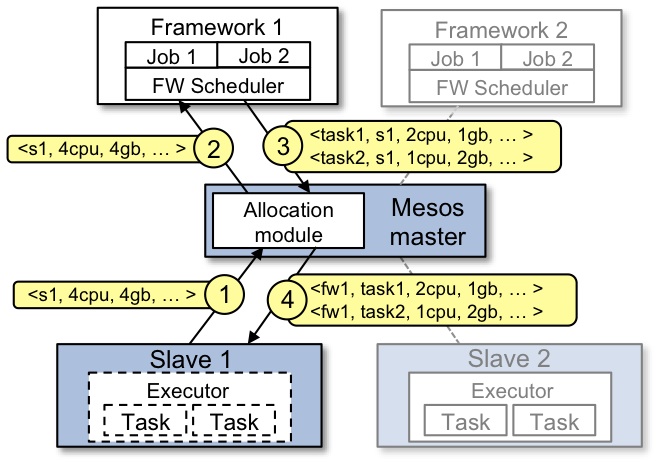


图 2‑4 Mesos资源分配流程

Figure ‑ Resource allocation process of Mesos

如图2-4所示，为了实现在同一组Slave节点集合上运行多任务这一目标，Mesos使用了隔离模块，该模块使用了一些应用和进程隔离机制来运行这些任务。虽然可以使用虚拟机隔离实现隔离模块，但是Mesos当前模块支持的是容器隔离。Mesos早在2009年就用上了Linux的容器技术，如cgroups和Solaris Zone，时至今日这些仍然是默认的。然而，Mesos社区增加了Docker作为运行任务的隔离机制。不管使用哪种隔离模块，为运行特定应用程序的任务，都需要将执行器全部打包，并在已经为该任务分配资源的Slave服务器上启动。当任务执行完毕后，容器会被“销毁”，资源会被释放，以便可以执行其他任务。

Mesos资源邀约和分配策略，对管理跨多个Framework和应用的资源，是不可或缺的。 资源邀约：即由Master向注册其上的Framework发送资源邀约。每次资源邀约包含一份Slave节点上可用的CPU、RAM等资源的列表。Master提供这些资源给它的Framework，是基于分配策略的。分配策略对所有的Framework普遍适用，同时适用于特定的Framework。 Framework可以拒绝资源邀约，如果它不满足要求，若此，资源邀约随即可以发给其他Framework。由Mesos管理的应用程序通常运行短周期的任务，因此这样可以快速释放资源，缓解Framework的资源饥饿； Slave定期向Master报告其可用资源，以便Master能够不断产生新的资源邀约。 另外，还可以使用诸如此类的技术， 每个Framework过滤不满足要求的资源邀约、Master主动废除给定周期内一直没有被接受的邀约。

### Google Omega

Google的第一代/第二代集群（资源）管理系统被称为Borg，而于2013年，Google公布了它的下一代集群管理系统Google Omega的设计细节。Omega提出Google经历了赛带资源管理系统，如图2-5所示，这三代系统分别如下：

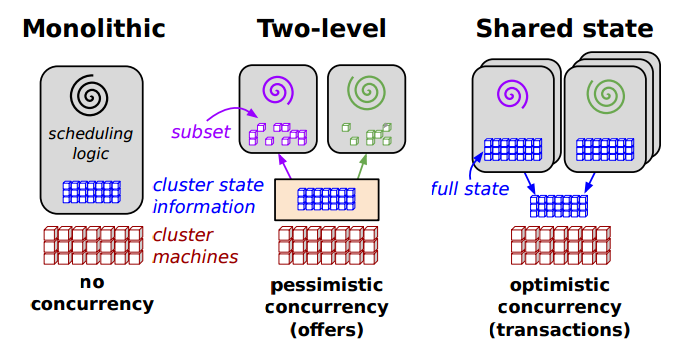


图 2‑5 三代资源调度系统概述

Figure ‑ Overview of resource management system

1. 中央调度器（Monolithic scheduler）

中央式调度器的特点是，资源的调度和作业的管理功能全部放到一个进程中完成，开源界典型的代表是Hadoop JobTracker的实现。这种设计方式的缺点很明显，扩展性差：首先，集群规模受限，其次，新的调度策略难以融入现有代码中，比如之前仅支持MapReduce作业，现在要支持流式作业，而将流式作业的调度策略嵌入到中央式调度器中是一项很难的工作。

Omega论文中提到了一种对中央式调度器的优化方案：将每种调度策略放到单独一个路径（模块）中，不同的作业由不同的调度策略进行调度。这种方案在作业量和集群规模比较小时，能大大缩短作业相应时间，但由于所有调度策略仍在一个集中式的组件中，整个系统扩展性没有变得更好。

1. 双层调度器（Two-level scheduler）

为了解决中央式调度器的不足，双层调度器是一种很容易想到的解决之道（实际上是分而治之策略或者是策略下放机制）。双层调度器仍保留一个经简化的中央式调度器，但调度策略下放到各个应用程序调度器完成。这种调度器的典型代表是Apache Mesos和Hadoop YARN。

双层调度器的特点是，各个框架调度器并不知道整个集群资源使用情况，只是被动的接收资源；Mesos Master仅将可用的资源推送给各个框架，而框架自己选择使用还是拒绝这些资源；一旦框架（比如Hadoop JobTracker）接收到新资源后，再进一步将资源分配给其内部的各个应用程序（各个MapReduce作业），进而实现双层调度。

双层调度器的缺点是：

A）各个框架无法知道整个集群的实时资源使用情况：

很多框架不需要知道整个集群的实时资源使用情况就可以运行的很顺畅，但是对于其他一些应用，为之提供实时资源使用情况可以为之提供潜在的优化空间，比如，当集群非常繁忙时，一个服务失败了，是选择换一个节点重新运行它呢，还是继续在这个节点上运行？通常而言，换一个节点可能会更有利，但是，如果此时集群非常繁忙，所有节点只剩下小于5GB的内存，而这个服务需要10GB内存，那么换一个节点可能意味着长时间等待资源释放，而这个等待时间是无法确定的。

B）采用悲观锁，并发粒度小：

在数据库领域，悲观锁与乐观锁争论一直不休，悲观锁通常采用锁机制控制并发，这会大大降低性能，而乐观锁则采用多版本并发控制(MVCC ,Multi-Version Concurrency Control)，典型代表是MySQL innoDB，这种机制通过多版本方式控制并发，可大大提升性能。在Mesos中，在任意一个时刻，Mesos资源调度器只会将所有资源推送给任意一个框架，等到该框架返回资源使用情况后，才能够将资源推动给其他框架，因此，Mesos资源调度器中实际上有一个全局锁，这大大限制了系统并发性。

1. 共享状态调度器（Shared state scheduler）

为了克服双层调度器的以上两个缺点，Google开发了下一代资源管理系统Omega，Omega是一种基于共享状态的调度器，该调度器将双层调度器中的集中式资源调度模块简化成了一些持久化的共享数据（状态）和针对这些数据的验证代码，而这里的“共享数据”实际上就是整个集群的实时资源使用信息。一旦引入共享数据后，共享数据的并发访问方式就成为该系统设计的核心，而Omega则采用了传统数据库中基于多版本的并发访问控制方式（也称为“乐观锁”, MVCC, Multi-Version Concurrency Control），这大大提升了Omega的并发性。

由于Omega不再有集中式的调度模块，因此，不能像Mesos或者YARN那样，在一个统一模块中完成以下功能：对整个集群中的所有资源分组，限制每类应用程序的资源使用量，限制每个用户的资源使用量等，这些全部由各个应用程序调度器自我管理和控制，根据论文所述，Omega只是将优先级这一限制放到了共享数据的验证代码中，即当同时由多个应用程序申请同一份资源时，优先级最高的那个应用程序将获得该资源，其他资源限制全部下放到各个子调度器。

引入多版本并发控制后，限制该机制性能的一个因素是资源访问冲突的次数，冲突次数越多，系统性能下降的越快，而google通过实际负载测试证明，这种方式的冲突次数是完全可以接受的。

## 服务治理

服务治理是分布式系统及面向服务系统架构的核心组成。客户端怎样知道存在于多台主机上的服务的IP以及端口即为服务治理在最初所面临的问题。然而，随着系统中服务的不断增多，在自动或手动扩容、主机失效、新服务部署的情况下，服务的地址变化将变得极其频繁。此时，为了避免服务终端，动态服务注册与发现重要性将大大增加。

在定位服务地址的问题上，有两个关键性问题：

1. 服务注册：服务将自身地址注册到一个中央注册中心当中。通常它将注册自己的IP及端口，验证凭据、使用的协议、版本号或一些使用环境细节信息。

a）自注册模式：

自注册模式意味着服务实例对自身的注册以及注销负责。同时，在必要的情况下需要发送心跳信息至服务中心防止其注册信息过期。图2-6展示了该模式的架构：

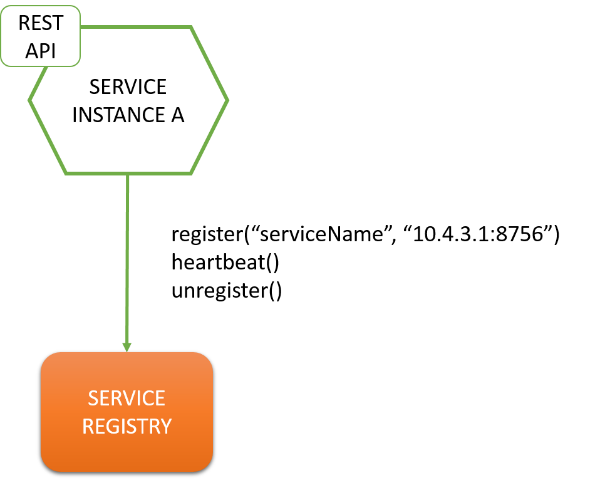


图 2‑6 自注册模式

Figure ‑ Self-registration pattern

自注册服务拥有许多的优势与缺陷，其中一个优势在于它的相对简单性，且不依赖于其他的系统组件。然而，其中一个主要的缺陷在于它导致服务实例与服务中心耦合。必须针对每种变成语言或框架实现特定的代码。

b）第三方注册模式：

使用第三方注册模式将服务实例与注册中心解耦，服务实例不再负责自身的注册与注销。反之，一个称为服务注册器的系统组件被用以处理服务注册。服务注册器通过拉取运行信息或订阅时间的方式跟踪一组运行实例，当它发现一个新的可用服务时就将该服务注册至服务中心。与此同时，服务注册器也会处理服务注销的问题。第三方注册模式如图2-7所示：

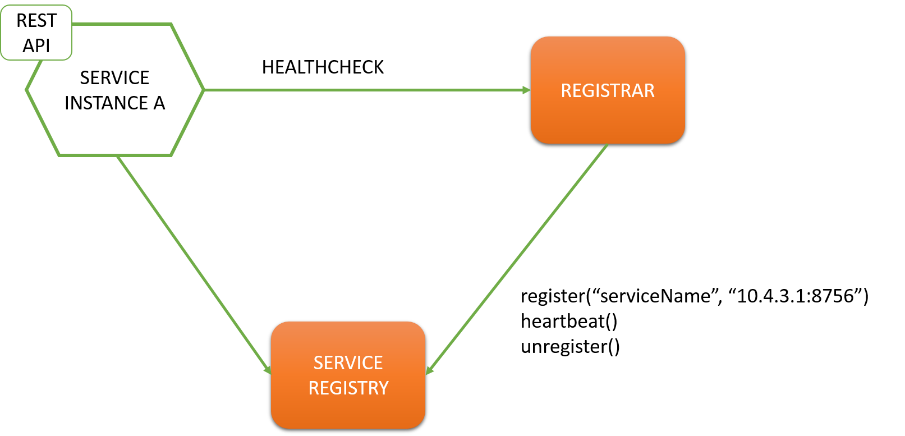


图 2‑7 第三方注册模式

Figure ‑ Third-party registration pattern

1. 服务发现：客户端程序查询中央注册中心获取服务地址。

a）客户端服务发现模式：

当使用客户端服务发现模式时，客户端负责在可用服务实例的网络地址中进行选择，并在其中进行负载均衡决策。客户端通过查询服务注册中心获取服务信息，并使用一个负载均衡算法来选择可用服务并发送请求。

客户端服务发现模式有自己的优势与缺陷。这个模式相对简单，并且除了注册中心之外，没有其他的系统部件需要移动。同时，由于客户端知道可用的服务实例地址，它能做出诸如全局一致性hash等智能的负载均衡决策。而客户端服务发现模式的一个重大的缺陷在于注册中心与客户端耦合严重，对于每个变成语言或框架都需要实现特定的服务发现注册逻辑。

图2-8详细描述了该模式的架构：

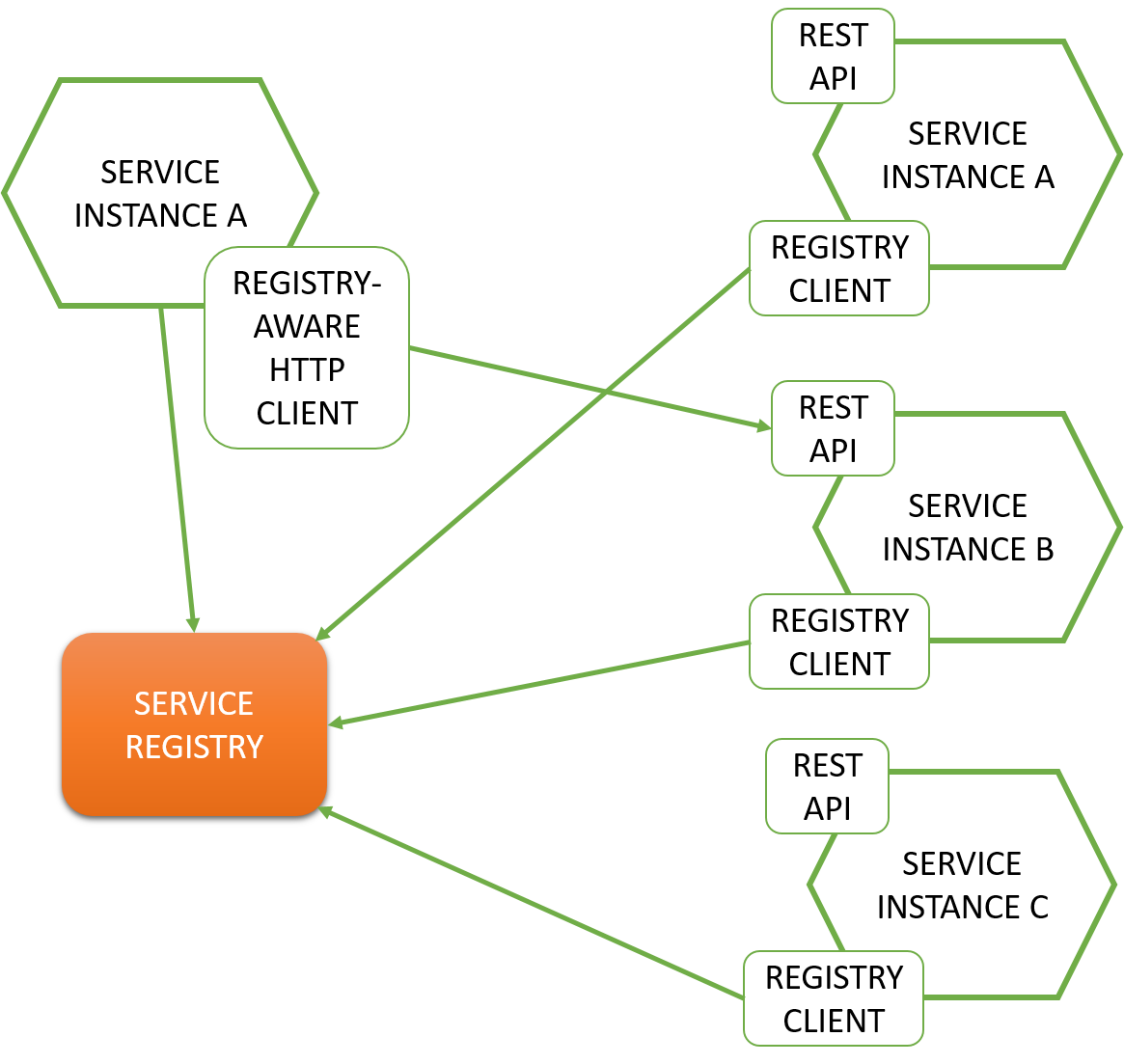


图 2‑8 客户端服务发现模式

Figure ‑ Client-side service discovery pattern

b）服务器端服务发现模式：

服务器端服务发现模式如图2-9所示：

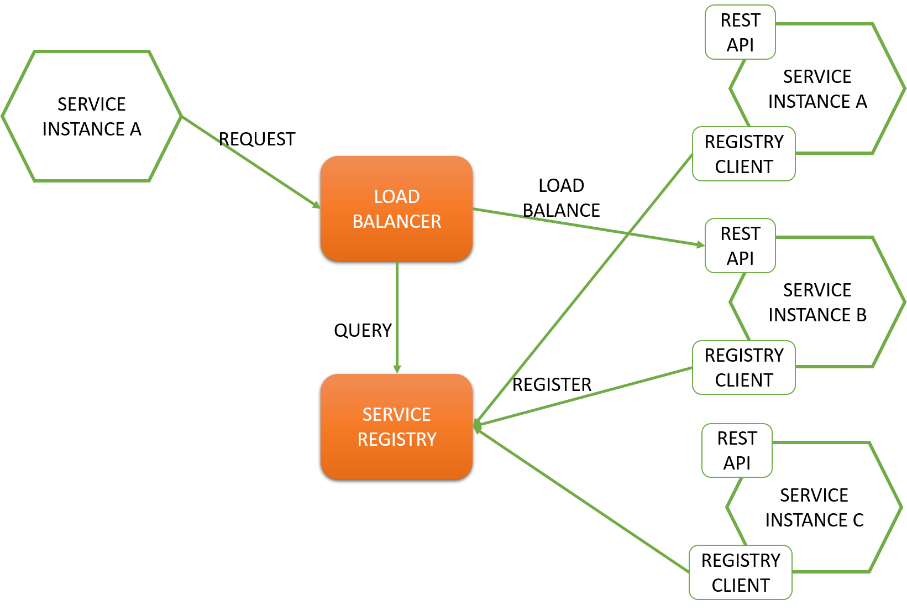


图 2‑9 服务器端服务发现模式

Figure ‑ Server-side service discovery pattern

客户端发送服务请求到一个负载均衡器。负载均衡器查询服务注册中心并将每个请求路由到一个可用的服务实例上。与客户端服务发现模式相同，服务实例通过服务注册中心注册与注销。

AWS Elastic Load Balancer（ELB）是服务器端服务发现模式的一个例子。一个ELB通常被用于对外部Internet流量进行负载均衡处理。HTTP服务器如Nginx也可用做服务器端服务发现负载均衡器。而例如Kubernetes和Marathon一类的部署环境在集群中每台主机上运行了一代理。这个代理担任着服务器端服务发现负载均衡器的职责。

在解决上述两大关键性问题的基础上，通常还需要考虑以下的开发或运营方面的问题：

1）监控：当一个注册的服务失效后应该怎样处理？立即注销、一段时间后注销、利用另一守护进程注销都是可行的解决方案。通常服务被要求实现一个心跳机制来确保自身的可用性，而将处理失效服务的责任交给客户端处理。

2）负载均衡：若一个服务注册了多个实例，客户端怎样在多个实例中平衡分配流量。若服务拥有主服务，客户端怎样发现这个主服务？

3）集成因素：注册中心支持哪些语言的注册绑定？集成需要内嵌的注册发现库还是使用外部进程来实现？

4）运行时依赖：服务需要什么样的运行环境？

5）可用性：在节点失效后服务是否仍然可用？是否可在运行时升级且不会导致服务中断？注册中心变成系统中核心部分后怎样解决其单点失效的问题？

目前国内外已经存在诸多的用于服务发现、注册的基础服务，表2-1对它们进行了详细的对比：

表格 2‑1 服务发现、注册产品比较

Table 2-1 Comparison between service registration & discovery products

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 名称 | 类型 | 一致性 | 语言 | 依赖 | 集成 |
| Zookeeper | 通用 | 强一致 | Java | JVM | 客户端集成 |
| Doozer | 通用 | 强一致 | Go |  | 客户端集成 |
| Etcd | 通用 | 混合 | Go |  | 客户端集成/HTTP |
| SmartStack | 专用 | 最终一致 | Ruby | Haproxy、Zookeepr | Sidekick |
| Eureka | 专用 | 最终一致 | Java | JVM | Java Client |
| NSQ | 专用 | 最终一致 | Go |  | 客户端集成 |
| Serf | 专用 | 最终一致 | Go |  | 本地命令行 |
| Spotify | 专用 | 最终一致 | N/A | Bind | DNS库 |
| SkyDNS | 专用 | 混合 | Go |  | HTTP/DNS库 |

## 微服务软件系统架构

微服务是一种软件架构风格，它代表着一种开发多个运行在独立进程、环境中的使用轻量级通信协议进行交互的小型服务来代替传统的单一应用的大型应用的软件架构风格。这些微服务是围绕着业务功能进行划分的，并且可独立部署、升级、管理。

为了更好的解释微服务架构，我们将其与传统的单一应用架构进行比较。企业级应用通常包括三个部分：1）客户端接口（通常包括运行在客户浏览器中的HTML页面以及javascript代码），2）数据库（通常为关系型数据库中的多个表集合），3）服务器端应用（处理HTTP请求，执行业务逻辑，获取更新数据）。而服务器端应用通常是一个逻辑上的集合，对于服务器端的任何修改都会导致对整个服务器端应用的重新编译以及部署。单一应用架构是可行的，但随着应用范围以及复杂度的增加，其缺点也显得越来越明显：每次更改都要求整个服务器端重新编译以及部署，在扩容时也只能够对整个应用进行扩容，而不能够只针对资源需求更低的一个模块来进行扩容。微服务架构如图2-10所示

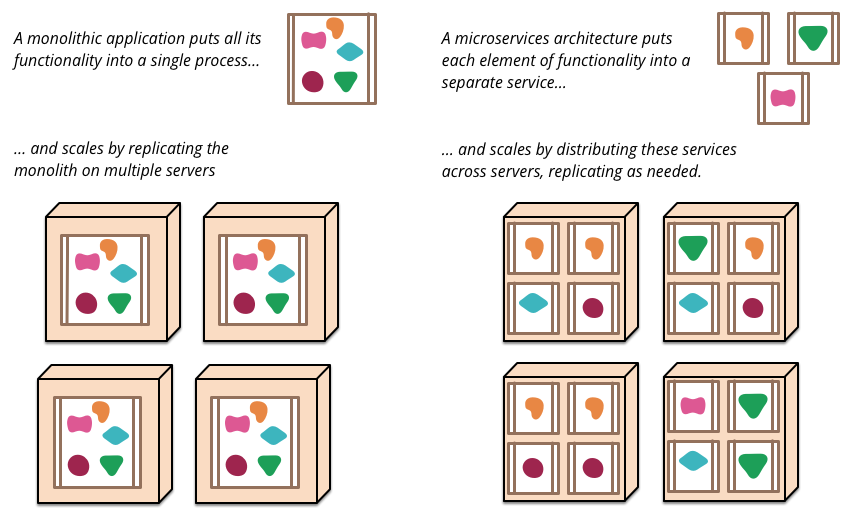


图 2‑10 微服务架构与垂直架构对比

Figure ‑ Comparison between micro-service and monolithic architecture

上述的问题就引出了一个新的软件架构：微服务架构。微服务架构将传统的单一应用以一组独立的服务来构成，与此同时，这些服务是可以单独部署以及水平扩展的。每一个这样的服务都拥有着严格的模块边界，甚至允许不同的服务使用不同的开发语言来进行开发。

微服务架构会使用依赖包，但其主要通过将软件分割成服务来进行模块化。我们定义依赖包为链接入程序并通过in-memory程序调用使用的程序模块，而服务为使用RPC或者网络请求进行交互的进程。使用服务作为系统模块有两大优势：首先服务可独立部署，这样对一个单独的模块进行更改不会影响到系统中的其他模块，另一个优势则是服务间通过显示的远程调用来进行通讯，从而减小程序间的接口定义难度。然而，微服务架构也是有缺点的。首先，远程调用比进程内函数调用开销更大，其次，将程序职责在服务间进行迁移的难度也会更加的复杂。

去中心化治理：中心化治理的会导致整个程序开发使用单一的技术和平台。然而，这样的解决方案是受到很大限制的，不是每个问题都能通过单一的技术来解决。针对具体的问题选择最适合的技术显然是一个更加的做法。微服务架构则将这样的方法带入了软件开发之中。

去中心化数据管理：在对概念模型以及业务逻辑进行去中心化的同时，微服务架构同样对数据进行了去中心化的管理。每个微服务都独自管理自己的数据，如图2-11所示，这意味着微服务可根据具体的业务需求选择合适的数据库（关系型或非关系型），但同时也意味着在服务间数据一致性问题会更加难以处理。

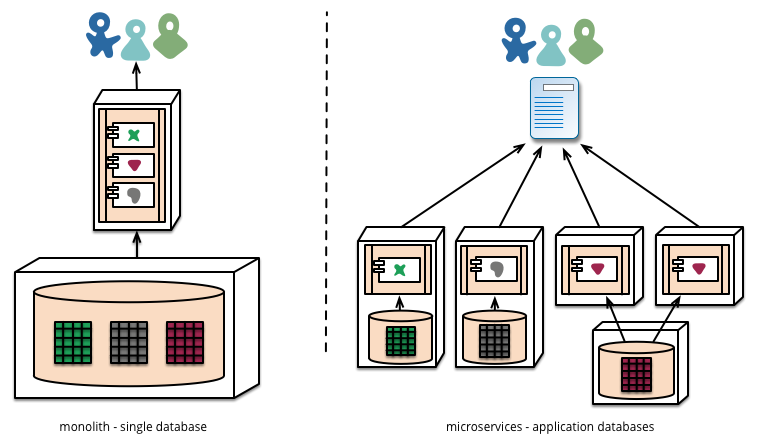


图 2‑11 微服务架构数据管理

Figure ‑ Data management for micro-service architecture

测试部署自动化：随着云平台的日渐成熟，测试部署自动化在近年中发展迅速。而云平台大大减少了构建、部署、运行微服务的复杂度。

预防软件错误：微服务架构需要在设计时将服务异常纳入考虑范围，由于任何服务调用都可能因为服务提供者失效或网络失联而失败，因此客户端需要很好地处理这一类情况。

迭代式设计：当尝试将一个系统服务化时，我们将面临一个至关重要的问题：怎样将系统划分成不同的服务？应该使用怎样的原则来进行划分？由于一个服务的关键要素是独立部署升级，因此在系统划分时应该考虑在其需要重构时不会影响到其他的协作者。

## 本章小结

本章对本文的研究主题――资源管理、服务治理及微服务进行了详细的综述，对其概念进行说明，并对当前国内外研究现状进行了总结。从微服务架构的特征分析中引入了服务治理以及资源管理的观点，对其可行性及意义进行了分析，进而奠定了本文的研究思路。

# 系统需求分析与架构设计

本章对实验室承担的智慧邻里社区服务软件系统研发实际业务需求进行分析，并提出基于微服务架构的应用服务及资源管理系统设计方案。（3.1、3.2、3.3目录解释）

## 业务分析

邻里社区定义，有哪些业务。

本文以智慧邻里社区服务软件系统研发项目为背景，该系统旨在为居民、居委会、业委会、物业和其他服务机构等社区相关人员提供的一种智慧化的生活环境和新的社区管理形态。系统建立了统一的资源管理平台、运行环境管理平台、服务治理系统，对社区政务服务（党建服务、政务公开、社区自治、事务办理等）、社区便民服务（社区金融、生活服务、物业管理等）、社区邻里社交服务（点对点聊天、群组聊天、贴吧、兴趣组等）等服务的运行时环境及资源进行统一管理。

智慧邻里社区服务软件系统业务需求如下：

* 社区政务服务

社区政务服务包括党建服务、社区政务公开、社区自治、事务办理等服务。

* 社区邻里社交服务

社区邻里社交服务包括居民点对点聊天、群组聊天、社区贴吧、社区共同兴趣小组等服务。

* 社区便民服务

社区便民服务包括社区金融服务、生活服务（洗衣、外卖、超市等）、物业管理服务（物业报修、物业缴费、门禁管理等）。

本论文提出的系统以满足上述需求而进行设计，同时，本系统还需要解决当前系统在邻里社交、服务治理及资源管理中存在的问题：

1. 传统社交模型大多以用户间通讯为首要目标，用户角色单一。本系统在基于实名制的基础上需要与社区政务、便民服务等结合，提供更为复杂的邻里社交服务。因此，需要建立统一邻里社交模型，结合系统中多种角色以及应用服务对邻里社交进行建模。
2. 系统中服务种类繁多，服务间依赖关系复杂，这样造成了服务管理、升级、维护、扩容的困难。因此需要通过服务治理系统来对系统中服务进行综合管理。
3. 由于系统中所有服务种类繁多，怎样部署运行服务，从而尽量提高集群资源利用率成为一大难点。因此需要通过资源管理平台来对系统中服务的部署运行及资源分配进行统一管理。

针对上述问题，本文思路如下：

1. 建立基于实名制的邻里社交系统

本文结合社区中不同的角色定义与应用服务需求，在传统的即时通讯社交服务以及身份管理平台的基础上，对社交信息从身份、属性、内容等方面加以抽象，建立基于实名制的邻里社交模型，并基于MQTT协议实现应用层即时聊天协议，同时实现用户状态管理、应用信息推送服务，构建完整的邻里社交系统。

1. 基于微服务架构的服务治理系统

在对系统中应用服务微服务化的基础上，结合服务治理框架Dubbox，设计并实现服务系统，提供服务发现与注册、监控调用统计、容量评估、负载均衡等功能，并结合监控信息与统一资源管理系统，对服务进行动态扩容。

1. 支持长服务的统一资源管理平台

结合开源资源管理平台Apache Yarn，在其基础上改进Yarn自身基于jobId的服务管理机制，通过服务发现注册机制扩展服务管理的功能及灵活性。解决由于节点迁移后导致的整体服务不可用的问题，并提供更多的协议支持；改进Yarn对于服务日志的管理机制，通过重写日志管理器及客户端API，支持在服务运行过程中对日志的管理，并支持日志管理策略的配置；实现Yarn中Container资源动态变更功能，使Yarn能够在不重启Container的情况下对Container资源进行变更，从而增加服务的可用性。为系统中微服务、基础平台服务提供统一资源管理平台。

## 系统需求分析

### 功能性需求

智慧邻里社区服务软件系统需要将广泛的社区应用，如居民生活服务、居委会工作服务、物业服务、党建服务、便民便利服务、邻里社交服务等进行整合，形成完整的社区服务闭环。因此，在本系统中，首先由如下描述：

1. 邻里社交子系统：邻里社交系统提供生活群众间点对点聊天、群组聊天、社区贴吧、兴趣小组功能，政府管理群众与生活群众间政务咨询，生活群众与服务群众间服务购买、咨询功能的基于实名制信息的新型邻里社交系统。
2. 社区便民服务子系统：社区便民服务子系统提供如社区活动、二手市场、生活服务（外卖、超市、洗衣等）、物业管理（物业报修、物业缴费、评价管理）等生活服务。
3. 社区政务服务子系统：社区政务服务提供如事务办理、政务公开、社区选举、社区公告、党建服务。
4. 社区信息管理子系统：社区信息管理系统提供社区内诸如小区信息、楼栋信息、居民个人信息等基础数据信息管理服务。

根据以上描述，系统中用户分为三类：生活群众、社区服务群众、政府管理群众。生活群众、社区服务群众、政府管理群众这三类角色的职责如图3-1所示：（总的一个用例图，不要具体用例，再针对子系统单独细节用例图，用例详细）

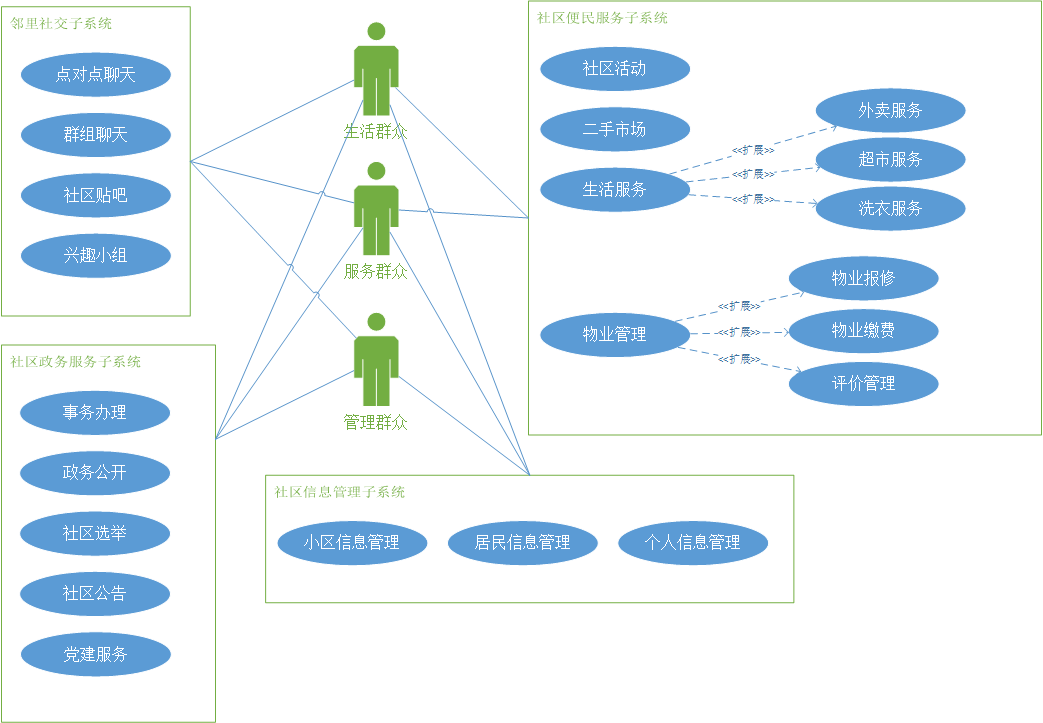


图 3‑1 智慧社区邻里服务软件系统用例

### 非功能性需求

本论文研究并实现的智慧社区应用服务及资源管理系统需要支持超过10W的用户以及超过30个具体应用服务的日常吞吐量。因此，系统在非功能性需求上主要有如下几点要求：（具体指标）

1. 可靠性：由于本系统主要为社区中各类角色群众提供生活、政务、社交服务，这些服务与居民生活每个方面息息相关，微小时间的故障即可对居民生活造成极大的不便，因此，系统应当具备高可靠性。
2. 性能：由于系统访问量大，服务类别繁杂，微小的延迟即可意味着用户的流失。因此在性能上需要满足对用户请求的快速响应。
3. 可维护性：？？由于系统采用微服务架构，且应用服务类别众多，因此需要系统在可维护性上达到一个较高的标准，从而应对可能的服务依赖、迁移、变更、升级时所带来的维护成本。

## 系统架构设计

### 技术架构

智慧社区应用服务及资源管理系统技术架构如图3-2所示：

1. 平台服务以Apache Yarn为基石，使所有应用服务运行与Yarn之上，对其进行统一资源管理。同时使用服务治理系统Dubbox结合协调服务中心Zookeeper解决服务管理中服务发现注册、监控统计、容量分析等相关功能。最后，消息中间件Rabbitmq用于缓存即时通讯消息，并交于Storm流式计算引擎进行处理。
2. 应用通用服务中表现层采用AngularJS，jQuery，Bootstrap，HTML管理页面，在前后端开发分离的同时，使用标准Restful接口以ajax异步方式与后端通信。
3. 业务逻辑层主要包括服务接口、实现以及服务治理相关库函数。该层包含了一系列应用服务逻辑，这些逻辑组件使用Spring IOC容器统一管理，并注入到相应的业务逻辑组件中。最后，通过服务治理库函数通过AOP方式管理服务发布、注册以及监控相关功能，对业务应用开发人员完全透明。
4. 持久层包括数据DAO接口，提供文档型数据存储（MongoDB），kv型数据缓存（Redis），大文件存储（HDFS）以及传统关系型数据存储（MySQL）服务。

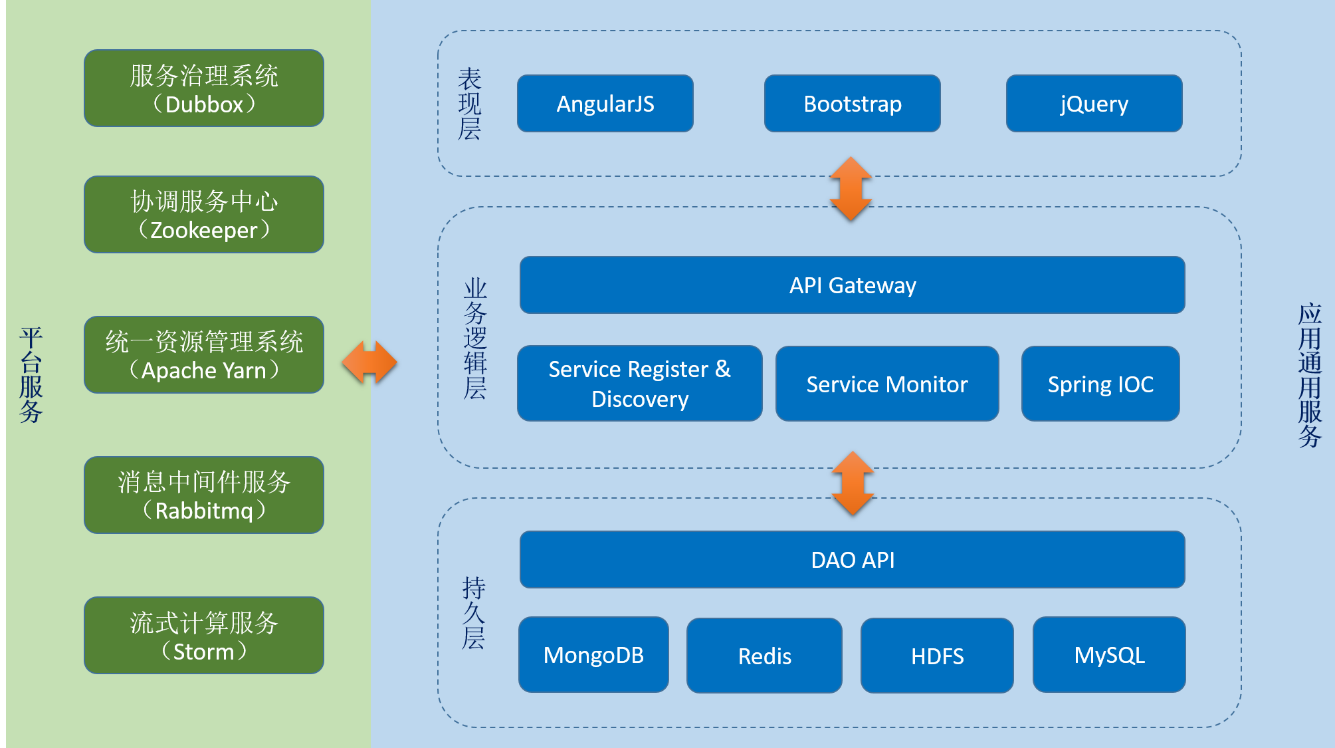


图 3‑2 智慧社区应用服务及资源管理系统技术架构图

### 逻辑架构

智慧社区应用服务及资源管理系统逻辑架构如图3-3所示，自顶向下供包括应用服务层、应用通用服务层、平台服务层三部分：

1. 应用服务层提供Web或移动端访问方式，包括五大模块：
   1. 党建服务：包含党员活动、组织关系、季度考察、党员活动、党费缴纳等模块，并在通用服务基础上提供党组群交流、身份管理以及支付功能。
   2. 居委会工作：包含事务办理、人事信息管理、社区公告、政务公开、民意投票等功能，为社区政务服务提供便捷。
   3. 物业服务：提供物业缴费、物业报修、评价管理等服务
   4. 居民生活：提供二手市场、兴趣小组、居民互助、邻里圈、互动贴吧等服务。
   5. 便民服务： 提供外卖服务、洗衣服务、超市、家政等便民服务应用。
2. 应用通用服务层包含应用服务所公用的基础应用服务：邻里社交服务、市民基础数据服务、身份管理服务以及支付服务。
   1. 邻里社交服务包含用户状态管理用以管理用户设备在线状态，并为推送服务提供信息基础。信息分发服务用以获取即时聊天消息，并根据预设规则决定是否转发消息。消息推送服务用于提供应用信息、广告推送。
   2. 市民基础数据模块对社区中诸如人口、楼栋、小区、服务等基础数据进行管理，提供给政府管理人员使用，是整个系统的实名制信息基石。
   3. 身份管理服务提供通讯录、个人信息管理、访问控制等功能，提供给系统用户更新个人信息，并根据居民身份对其进行访问控制。
   4. 支付服务提供支付功能与应用服务进行集成。
3. 平台服务层包含支持长服务的资源管理平台、微服务治理、基础服务、数据存储服务四大模块：
   1. 支持长服务的资源管理平台：在Apache Yarn资源管理平台基础上对其日志管理、服务管理、容器资源变更三大方面进行改进，使其支持诸如Hbase，MongoDB等长时间运行的服务。同时将所有应用服务运行于Yarn上，对资源进行统一管理。
   2. 微服务治理：提供应用服务间服务发现注册、调用统计、服务监控、容量评估功能，并根据监控信息与资源管理平台集成，实现服务的水平扩展功能。
   3. 基础服务：包含消息队列服务以及应用协调服务，消息队列服务是邻里社交系统基石，用于缓存大量的用户聊天信息。应用协调服务作为服务发现注册功能基石，提供高可用的服务注册中心。
   4. 数据存储服务：包含数据缓存服务、文件存储服务、关系型数据、非关系型数据存储服务功能，提供给应用服务根据自身特点选择合适数据存储格式的能力。



图 3‑3 智慧社区应用服务与资源管理系统逻辑视图

### 部署视图

智慧社区应用服务及资源管理系统部署视图如图3-4所示。按功能划分，服务器资源可分为接入服务器（分为两类，一类为支持HTTP协议的Nginx服务器，一类为应用于即时聊天的MQTT服务器），消息缓存服务器，服务注册/发现服务器，流式处理引擎服务器，应用服务器，服务治理中心以及数据存储服务器。

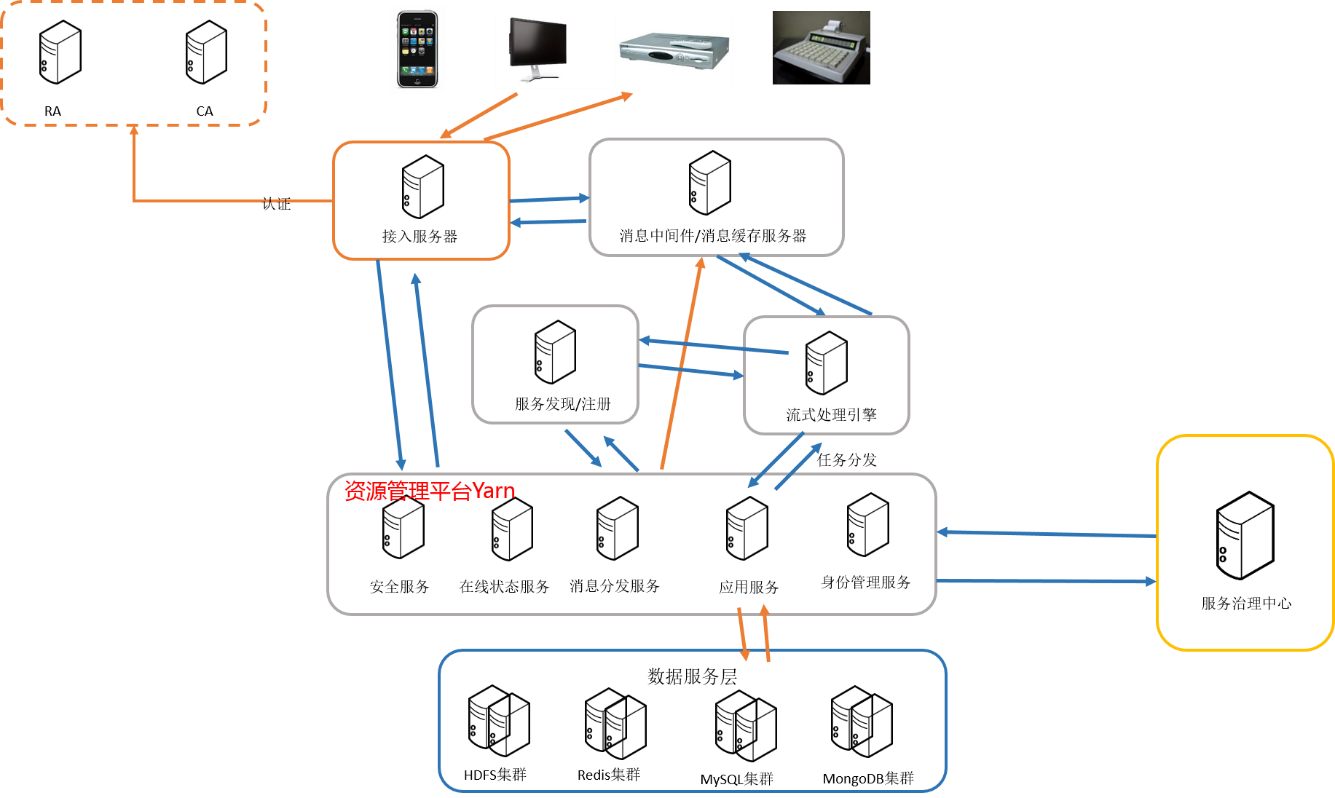


图 3‑4 智慧社区应用服务及资源管理系统部署视图

外部客户端接入系统均需通过PKI体系RA、CA验证。系统中所有通用、应用微服务均运行于资源管理平台Yarn之上。并与服务治理中心结合对服务进行动态资源管理。

### 系统实现环境

智慧社区应用服务及资源管理系统实现环境如表3-1所示：

|  |  |
| --- | --- |
|  | 系统实现环境 |
| JVM | Sun JRE/JDK 7.0 |
| 消息中间件 | Rabbitmq 3.5.6 |
| Web容器 | Tomcat 7.0.42 |
| Web前端 | Bootstrap 2.3.2 & AngularJS 1.3.9 & jQuery 1.11.3 |
| 应用服务后端 | 基于SSM进行开发 |
| 协调服务器 | Zookeeper 3.4.7 |
| 接入服务器 | Nginx 1.9.4 & rabbitmq-mqtt-server |
| 数据存储服务器 | MongoDB 3.2 & Redis 3.0.5 & MySQL 5.6 & HDFS 2.4 |
| 服务治理 | Dubbox 2.8.4 |
| 流式处理引擎 | Storm 0.9.6 |
| 统一资源管理平台 | 基于Apache Yarn 2.4进行开发 |

系统实现环境具体说明如下：

1. 整个系统主要开发语言为Java，使用JDK版本为7.0
2. 应用服务基于Spring MVC, Spring, Mybatis进行开发，并与服务治理框架Dubbox 2.8.4以及Zookeeper集成，实现基于微服务架构的应用服务治理。
3. Web端使用AngularJS + HTML + jQuery + Bootstrap框架进行开发，与后端完全分离，减小系统耦合度。
4. 统一资源管理平台基于Apache Yarn进行开发，在原有基础上对其服务管理、日志管理、容器资源变更三方面进行改进，实现支持长服务的统一资源管理平台。
5. 邻里社交模块基于MQTT协议进行开发，使用Rabbitmq作为消息缓存中间件，并利用Redis作为用户在线状态管理服务器。
6. 系统提供文档型存储（MongoDB），关系型存储（MySQL），文件存储（HDFS）以及缓存（Redis）服务，提供给应用服务不同存储服务选择。

## 本章小结

本章首先对智慧社区应用服务及资源管理系统进行了需求分析，包括业务分析、功能性需求以及非功能性需求。然后从系统的技术架构、逻辑架构、部署视图以及系统实现环境等四个方面进行了系统架构设计。

# 系统核心功能及关键技术研究与实现

## 邻里社交子系统

### 邻里社交模型及通信协议

1. 邻里社交模型

社区用户分为居民、政府管理人员、社区服务人员三类，系统社交模型如图4-1所示：

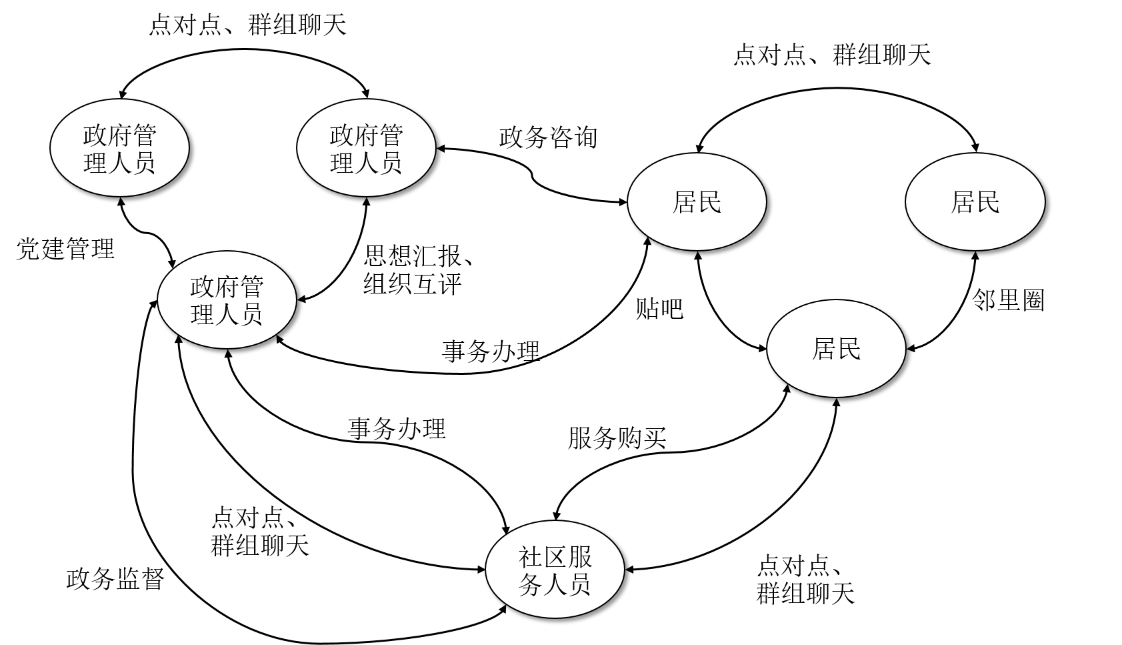


图 4‑1 社区邻里社交流程

1. 点对点、群组聊天方式（用户间通过即时通讯协议进行即时单人或以群组为单位的聊天交流方式）对系统内所有类型用户适用，用户间可通过聊天进行社交，信息交换。
2. 居民间独有社交方式有邻里圈，社区贴吧两种方式。
   1. 邻里圈：居民间近况分享平台，居民可发布自身状态或近况与好友进行分享。
   2. 社区贴吧：居民间社区信息交换平台，居民可发布、回复、关注各类主题。
3. 不同类型间用户社交系统提供服务号交流方式，社区服务人员、政府管理人员可发布管理自身所提供的服务，其他类型用户可通过选择使用该用户所发布的服务。例如，政府管理人员可发布社区居住证办理服务，而系统内其他人员使用该服务。
4. 通信协议

在对社区邻里社交模型抽象后，本系统选用MQTT协议构建整个社交系统信息交换协议基石。并在其上设计适用于本邻里社交模型的通信协议。

1. 协议消息语法：

通信协议消息语法完整格式如表4-1所示：

表格 4‑1 通信协议消息语法格式

|  |  |
| --- | --- |
| 属性 | 描述 |
| from | 信息发送方ID |
| to | 信息接收方ID |
| type | 通信消息类型，枚举类型，详情见表4-2 |
| subject | 消息主题，用于显示于聊天窗口标题栏处 |
| message | 具体消息内容 |
| timestamp | 消息时间戳 |

消息格式中type字段为枚举类型，用于描述通信消息类型，其枚举值如下表4-2所示：

表格 4‑2 type类型枚举值

|  |  |
| --- | --- |
| 属性 | 描述 |
| chat | 点对点聊天 |
| groupchat | 群组聊天 |
| headline | 系统通知、警告、实时数据更新类型 |
| error | 错误消息 |

例如，用户A与用户B点对点聊天时，用户A所发送的消息内容如图4-2所示，此时服务器接收到聊天消息后直接将信息转发推送至用户B。

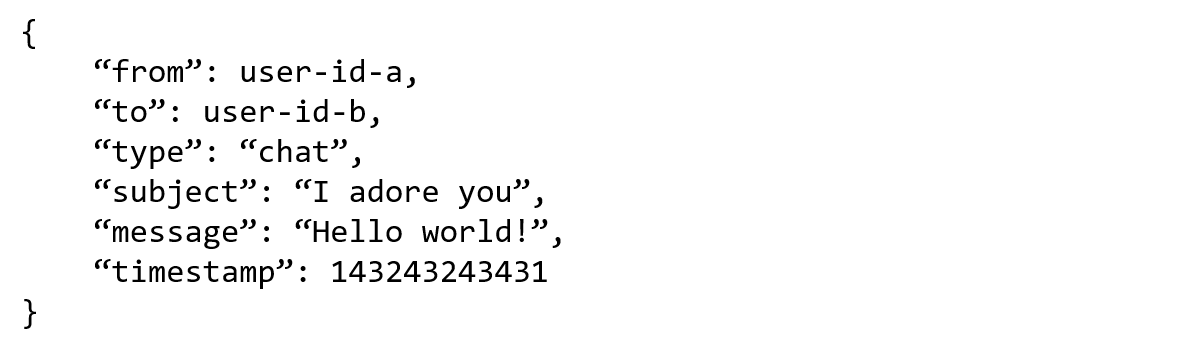


图 4‑2 点对点聊天消息示例

通信协议中另一重要部分为用户在线状态心跳信息，服务器需要通过该信息对用户在线状态进行管理，并进行相应的消息推送或持久化决策。在线状态信息格式如表4-3所示：

表格 4‑3 用户在线状态信息语法格式

|  |  |
| --- | --- |
| 属性 | 描述 |
| user-id | 用户ID |
| client-id | 用户设备ID |
| status | online/offline |

协议默认使用30秒为心跳发送时间间隔，客户端若保持在线状态，需间隔30秒时间发送一条status为online的心跳信息至服务器，若客户端被关闭，则在关闭前需向服务器发送status为offline的心跳信息及时通知服务器。

### 用户在线状态管理

用户在线状态管理模块用于对用户终端在线状态进行跟踪记录，并为消息缓存、推送、转发模块提供信息依据。该模块流程图如图4-3所示：

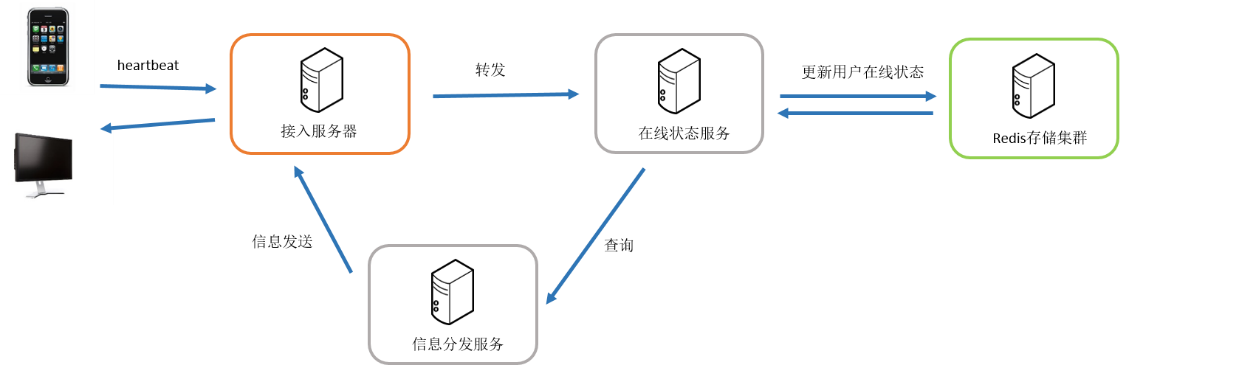


图 4‑3 用户在线状态管理流程图

诸如移动手机、PC机等客户端在上线后以30秒的时间间隔发送心跳信息至系统接入服务器，接入服务器将所有心跳信息转发至在线状态服务器，在线状态服务器根据心跳信息类型与Redis存储集群连接，相应更新或删除当前用户的在线状态。

同时，在线状态服务器提供用户在线状态查询功能，在信息分发服务进行信息分发时，需要首先查询该用户在线状态，若用户在线，则将聊天信息发送至用户在线客户端，若用户离线，则将聊天信息持久化保存并置为未阅读状态，方便用户上线后拉取信息。

在线状态服务实现的功能为，更新、查询用户在线状态信息。在线状态服务的类图如图4-4所示：

1. HeartbeatConsumer类：HeartbeatConsumer类用于建立与接入服务器连接，并获取客户端心跳信息。其consumeHeartbeatMsg方法获取心跳信息，并根据心跳信息类型字段type调用OnlineStatusServer对用户在线状态进行更新。
2. OnlineStatusServer类：用于处理所有用户在线状态相关逻辑
   1. notifyOnline方法：更新用户在线状态，首先从Redis存储中获得当前用户在线状态，加入心跳信息客户端，并更新该条目expire interval。
   2. notifyOffline方法：删除当前用户在线状态条目。
   3. query方法：用于外部客户端查询用户在线状态，输入为用户ID，以json形式返回用户所有在线客户端。
3. OnlineStatusQueryService接口：定义在线状态服务查询接口。
4. OnlineStatusQueryServiceImpl类：实现在线状态服务查询接口，调用OnlineStatusServer的query方法对用户在线状态进行查询。
5. DubboServiceRegistrator类：服务治理框架类，用于注册服务接口OnlineStatusQueryService，暴露该服务给其他服务调用。

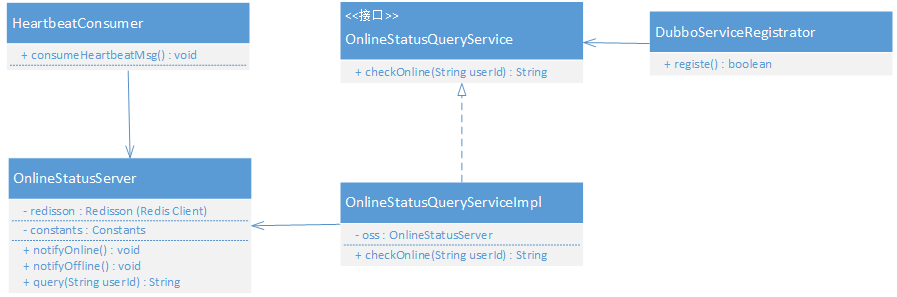


图 4‑4 在线状态服务类图

redis存储服务提供expired\_key功能，expired\_key提供在超过expire\_interval后自动删除该条目的功能。因此，用户在线状态信息使用redis的expired\_key进行保存，可利用该功能自动处理用户断线后在线状态信息删除的问题。同时，服务设置expire\_interval为6分钟，避免由于网络连接不稳定造成的用户假离线状态。

### 社交信息缓存、推送机制

由于移动客户端网络连接不稳定的特性，需要建立社交信息缓存、推送机制用于处理客户端与服务器连接暂时断开以及各类不同类型社交信息推送需求不同的问题，保证信息可靠性以及推送及时性。

系统中社交信息缓存、推送机制如图4-5所示：

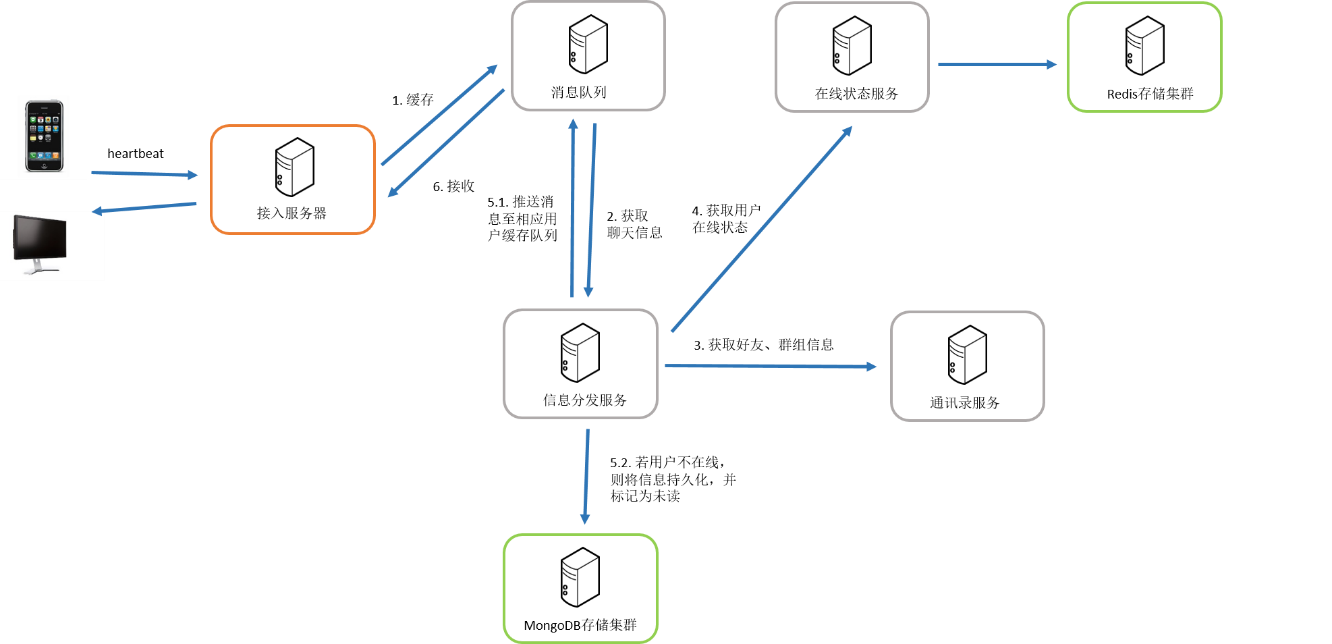


图 4‑5 社交信息缓存、推送机制

整个推送机制分为以下步骤：

1. 缓存：接入服务器将接受到的用户社交信息缓存至详细队列服务器，这样可以利用消息队列的异步处理机制避免用户量过大时造成的信息拥堵。
2. 信息分发服务不断从缓存队列中获得聊天信息并按如下表4-4步骤进行处理。

表格 4‑4 信息缓存推送机制

|  |  |
| --- | --- |
| 消息类型 | 处理步骤 |
| chat | 1. 通过通讯录服务获取用户好友信息，检查是否存在拉黑，屏蔽信息等特殊操作。若存在，则丢弃该消息，并返回给发送者一个带有相应信息的error消息；若不存在，则进入第二步。  2. 检查接收方在线状态，若在线，则将信息发送至接收方所属消息缓存队列，若不在线，则将信息持久化入MongoDB数据库，并标记该信息为未读 |
| groupchat | 1. 通过通讯录服务获取该群组信息，检查该用户是否属于该群。若不属于，则丢弃消息，并返回给发送者一个带有相应信息的error消息；若属于，则返回该群组所有成员信息。  2. 对群组内所有成员循环进行以下步骤：  a. 检查在线状态  b. 若在线，则推送信息  c. 若不在线，则将信息持久化并标记为未读 |
| headline | 该信息类型为系统管理员发送应用广告，推送通知时专用。流程为：  1. 通过通讯录服务获取所有用户信息  2. 将该信息推送至所有用户消息缓存队列 |
| error | 该信息一般为chat，groupchat类型消息异常时返回 |

## 支持长服务的资源管理服务子系统

本节主要做XXX，简单描述。

大规模集群调度/管理系统已成为各类程序及服务的首选平台。这些复杂的云平台为了达到最大的资源利用率，通常运行着数以千计的不可靠商业机器，并且必须能够处理各种各样的调度、部署、错误处理需求。本章以Apache Yarn为基础并对其进行深入分析，提出一种解决Yarn在服务长时间运行任务时所存在的关于服务注册、日志管理、容器资源伸缩性缺陷的解决方案，使得Yarn不仅能够运行传统的基于map-reduce模型的批量任务，还能成功的运行分布式计算框架（Storm, Spark等），以及各类长时间运行的微服务。

目前所有的资源管理系统拥有两点共同的目标：提升集群资源利用率以及实现服务的自动部署。在大数据时代，我们的数据中心通常会同时运行多种类型的服务器以及程序（例如离线、流式、迭代计算任务，网络服务器，数据库服务器等）。在传统情况下，我们将分配单一类型的服务给一个独立的集群，但由于不同服务的不同的资源需求，这些集群的资源利用率的差别非常明显。因此，诸如Yarn, Mesos, Borg, Omega, Corona, Torca等资源管理系统被设计以解决此类问题。

在对Yarn进行全面的分析之后，Yarn目前善于处理不需要与外部系统或用户交互的批量任务。它接收一个任务，并决定诸如Mapper、Reducer之类的模块在集群中何处运行，并将这些模块部署到相应的节点，令其启动，并完成其所被安排的任务。但当诸如Hbase, Storm, Spark等长时间运行的分布式计算框架出现时，Yarn难以告诉外部系统这些服务存在于哪里，以及怎样与这些计算框架进行交互。 除此之外，当Yarn中一个节点失效之后，Yarn所做的只是简单的将那个节点上所存在的模块调度到另一个节点之上并再次启动。这样的解决方案在处理map-reduce模型的任务时是可行的，因为所有map-reduce任务的模块都是无状态的，并且不需要直接与模型中其他模块进行通讯。但Storm这一类分布式计算框架出现时，问题变得不再那么简单。当Storm中Nimbus模块失效后，Yarn会将该模块调度到另一节点。然而由于Yarn是随机重调度失效节点的，Storm中所有需要与Nimbus节点进行交互的节点将找不到该节点，从而导致整个Storm的不可用。在Yarn中运行长服务的另一个问题在于日志管理，长服务通常都会随着运行时间的增长而产生大量的日志，而这些日志信息通常对于调试系统问题，以及记录系统状态起着重要的作用。因此Yarn需要提供一种有效的汇聚这一类日志以及在日志不再被需要时快速删除的解决方案。最后，Yarn的容器资源可伸缩性对于运行长服务同样重要。试想这样一种场景，Yarn上运行着一个tomcat的集群，但当整个集群的吞吐量上升时，需要动态提高这些服务所占用的资源以提高集群的吞吐性。

### 服务发现与注册

服务注册与发现在分布式计算中是一个历史悠久的问题。Apache Yarn允许服务运行于Hadoop集群。其中一类服务属于可使用Yarn现有API以及jobId属性进行管理的批量任务，此外Yarn还可以支持运行诸如Hbase集群或者tomcat集群之类的长服务。Yarn根据特定的资源需求以及集群节点资源使用情况来部署该类服务。传统情况下，这一类服务的IP会被注册到DNS服务器或者写入配置文件中使其能够被外部系统发现并使用，但由于Yarn部署的集群中节点的IP或者端口并不能被提前确定，这样的方案并不可行。因此，外部系统想要与Yarn动态部署的服务进行交互并不容易。

Yarn支持一个简单的允许Yarn Application Master注册一个网络URL或者IPC地址的简单注册器。然而目前它还存在如下的问题：

1. 不支持诸如Restful URL，集群模块RPC端口等复杂的注册信息。
2. 注册信息被映射到一个随着可变的Yarn application ID之上，这使得Yarn不可能通过静态引用来使用服务或者探测该服务是否在线。

针对这样的问题，我们提出的解决方案应该满足以下四个要求：

1. Yarn部署的服务节点应拥有注册它们自己的服务信息的能力，并且这些服务信息能够被外部系统或者内部系统发现并使用。
2. 注册的服务信息在服务迁移或热备切换之后能够被即时修正。
3. 一个服务节点应该能注册各种类型的服务信息（RPC，REST等）。
4. 注册服务器应该是高可用的，并能够支撑大型集群

基于Zookeeper的高可用性、强一致性、扩展性好、使用灵活的特性，本文选择Zookeeper来作为服务注册、发现服务器。并选择其命名空间中的一个分支来作为服务注册的根节点。（例如：“<ZK-ROOT>/serviceRegistryRoot”）

图4-1展示了Yarn在部署一个新服务时的流程。首先，系统管理员向Yarn提交一个部署新服务的请求。随后Yarn Resource Manager对服务进行资源分配并在相应的节点上启动Application Master。Application Master启动后会将服务的相应模块启动。最后，这些服务会将他们自身的信息注册到服务发现中心。

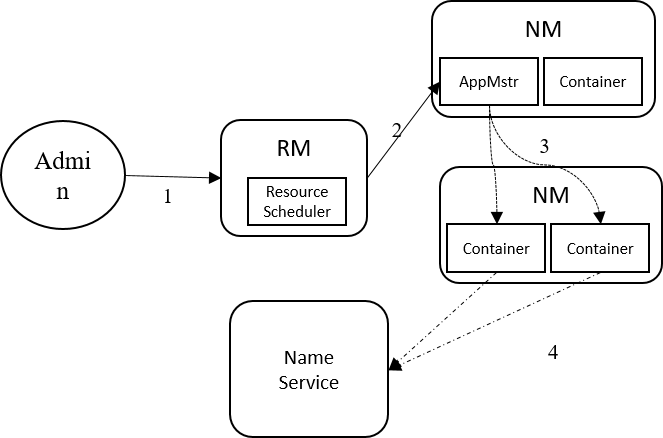


图 4‑6服务部署及注册流程

图4-2 展示了当发生节点迁移时的处理流程。当节点迁移发生时，Yarn Application Master无需做任何的变更，只需要被迁移的节点在被启动时重新向注册中心注册一下当前的信息即可。这样，无论内部还是外部的服务都能在该节点重新上线后第一时间找到它所在的位置。

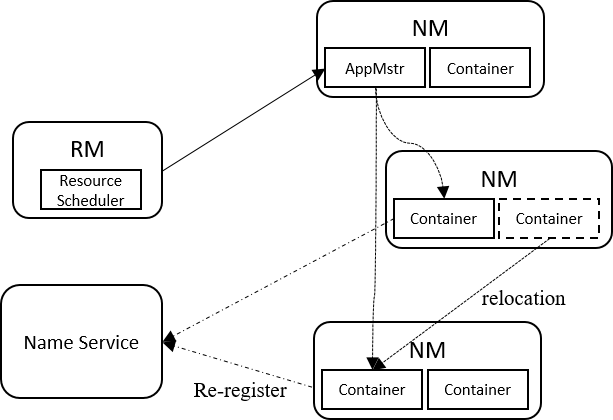


图 4‑7 节点迁移时服务注册流程

在深入讨论解决方案之前，我们首先需要定义一些基础概念，具体概念如下表所示。

表格 4‑5基本概念

|  |  |
| --- | --- |
| 名称 | 概念 |
| Service | 在Yarn集群中部署的分布式应用 |
| Service Record | 注册中心中描述服务实例的记录 |
| Component | 一个分布式服务的组件（例如Storm Nimbus节点） |
| Endpoint | 服务实例或组件实例的注册信息（可为内部Endpoint：服务内部通讯，外部Endpoint：服务与外部通讯接口） |

对于每一个Service，我们都将向注册中心注册一个Service Record。注册路径采用分层结构，并以“/”作为根节点以及“/”符号作为分隔符。同时，所有的路径描述必须是满足RFC1123的，这样使得我们的服务命名策略符合DNS的命名策略，从而使得我们随后能够通过DNS协议来对服务进行访问。同时，每一个Service Record将在Zookeeper中注册为persistent znodes。这样可以保证在任何情况下（计划服务升级，服务失效），记录始终存在。

每个Service实例的Service Record内都会记录该Service所暴露出的所有的Endpoints。Service Record详细信息如表格4-2所示。

表格 4‑6 Service Record详细结构

|  |  |
| --- | --- |
| 名称 | 描述 |
| Id: string | YARN程序或容器ID （静态部署服务为空）。 |
| Description: string | 描述信息 |
| registrationTime: long | Unix时间戳，注册时间 |
| Persistence: int | 持久化策略 |
| External: List<Endpoint> | 提供给外部访问者的所有Endpoint信息 |
| Internal: List<Endpoint> | 提供给集群内部访问者的所有Endpoint信息 |

同时persistence属性定义了一个Service Record及其子节点什么时候被删除，所有persistence属性列举如下表4-3。

表格 4‑7 persistence属性列表

|  |  |  |
| --- | --- | --- |
| 策略 # | 名称 | 描述 |
| 0 | 永久 | Record一直存在直到被显示删除 |
| 1 | 临时 | 连接断开时自动移除 |
| 2 | 程序结束 | 当Yarn程序结束时移除 |
| 3 | 程序请求结束 | 当前程序请求结束时移除 |
| 4 | 容器 | 容器生命周期结束时移除 |
| 5 | 集群重启 | Yarn集群重启时移除。不包括HA切换的情况 |

这里我们以在Yarn中部署一个tomcat集群为例，在创建并注册程序后，注册中心结构如下图4-1所示：

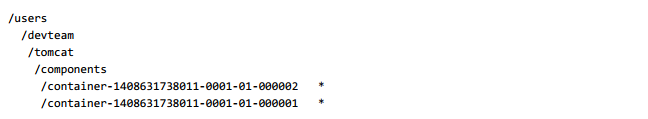


图 4‑8 注册中心示例

如图4-2所示，/users/devteam/tomcat这个Service Record描述了整个tomcat集群的信息。它列出了Yarn Application ID，并将URL暴露给一个负载均衡器，这里他的persistence策略是0（手动删除）。

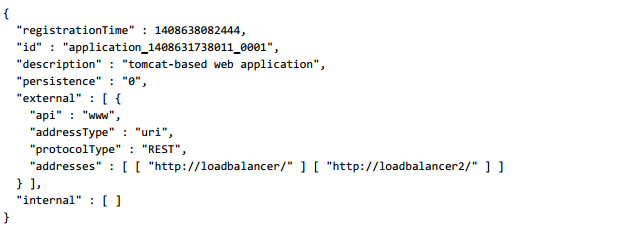


图 4‑9 Service Record示例

由上可以看出，这个Service实例是由两个component组成的，每个都用他们独有的container ID注册，并被标记为临时记录。因此当这个container被释放或者这个component出错时，这个记录会被自动删除。



图 4‑10 container记录示例

### 日志管理

在Yarn中运行长服务将不可避免地带来大量的服务日志数据，在日志数据产生一段时间后，这些数据中的绝大部分对于整个系统已无关紧要。目前，Yarn的日志处理机制仅仅支持在map-reduce任务完成后进行一个日志的收集或清理，这将带来如下的问题：

1. 因为长服务的长期在线特性，Yarn无法即时处理长服务中产生的大量日志，从而导致无用日志数据占满整个磁盘空间。
2. 没有灵活的日志处理机制，不允许服务选择日志管理方案，例如允许用户随时发起日志处理请求，允许用户选择日志管理策略等。

图4-6展示了目前Yarn在进行日志管理时的步骤。首先，当Application Master在Container中启动NM以完成相应的（map/reduce）任务时，它会启动一个与之对应的日志收集服务。这个日志收集服务间隔一段时间检查一次这个程序是否结束或被中断，如果已结束，那么它开始收集并传输日志信息到外部存储系统中（一般为HDFS）。

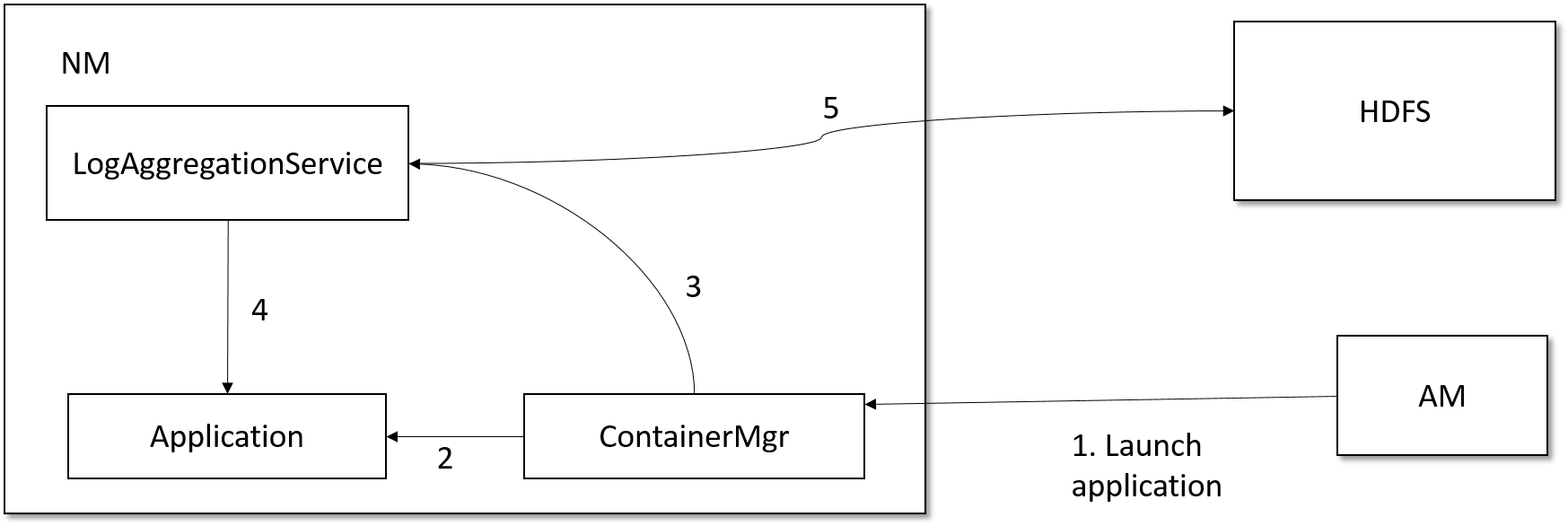


图 4‑11 Yarn日志处理工作流

本文实现了一个全新的Log Aggregation Service实例，该实例可以不同的策略来对日志进行相应处理（例如基于时间的处理策略，基于大小的日志处理策略），同时可以接受来自client端的指令，进行策略的更改或立即执行日志处理。改进后的流程如图4-7所示。

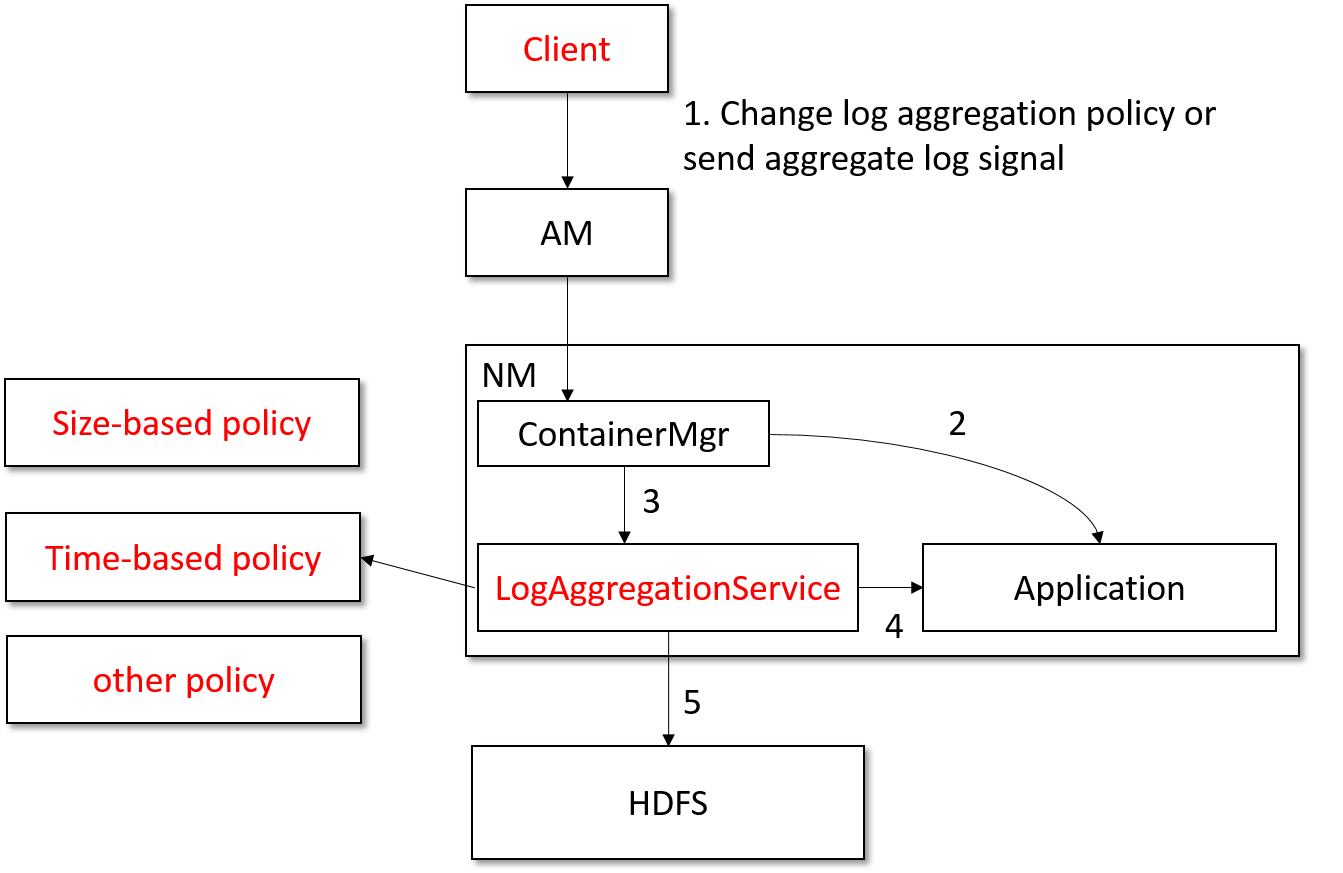


图 4‑12 改进后日志处理流程

改进后Client端能够在服务运行期间任意时间向该服务发送两类请求，一为更改日志处理策略请求，二为日志处理请求。从而给予Client最大的控制权限。

### 容器资源可伸缩性

诸如MongoDB，Tomcat Server一类的长服务在运行过程中随着服务吞吐量的变化不可避免地会遇到资源消耗变更的问题。在传统的资源管理方式下，一种解决方案是将该类服务迁移到内存更大、CPU处理能力更强的服务器上，或在虚拟机环境下增大虚拟机资源。然而，这类解决方案都需要将服务停止，修改配置或迁移服务之后才能使其生效。目前Yarn的容器资源管理方式仅仅支持在服务启动时对资源进行一次性分配，同样不能满足服务资源变更的需求。因此，需要对Yarn容器资源可变更性进行设计优化。

Yarn容器资源可变更性包括两个方面：资源的增加以及资源的减少。其整个变更流程中主要设计到Yarn中Resource Manager，Node Manager，Application Master，Container Monitor，Container Manager，Fair/Capacity Scheduler六大模块的通信交互，状态同步。因此，在设计整个Container资源变更机制时，本文对如下方面进行了相应的设计权衡：

1. 资源增加、减少请求都应通过Resource Manager来进行，原因如下：
   1. 避免race condition
   2. Resource Manager是验证、协调所有资源请求的地方
2. 容器资源增加请求是同步请求：资源增加请求包含更新容器资源、通知Container Monitor，调整Container Monitor三个步骤，而我们使用同步方式来处理这三个请求而不是使用异步方式处理的原因在于：
   1. 请求是轻量级的
   2. 确保返回状态的可靠性
3. 只有处于Running State的Container才能进行资源变更：这样做的主要原因是避免race condition。不支持ACQUIRED状态的Container进行资源变更的原因是那样会引入Container当前token过期的异常情况。
4. Container资源增加请求通过Node Manager来实现：这主要是基于性能的考虑。不同于Container资源减少请求，一旦Application Master将资源增加请求发给了Resource Manager，它必须确保在使用。

在满足上述基本设计需求的情况下，容器资源增加的抽象流程如图4-8所示：

1. Application Master发送资源更改请求给Resource Manager，并指明相应的Container ID以及目标容量
2. Resource Manager在调度周期中对相应Container增加资源
3. 如果资源增加请求被允许，Resource Manager向Application Master发送一个新令牌，并启动过期计时器
4. Application Master发送带有新令牌的资源增加请求给Node Manager。此时Container Manager以原子方式通知Container Monitor更新资源信息并更新内部状态信息
5. Node Status Updater在心跳发送周期向Resource Manager发送资源已更改消息
6. 资源调度器将该Container从过期计时器中注销，完成整个资源增加流程

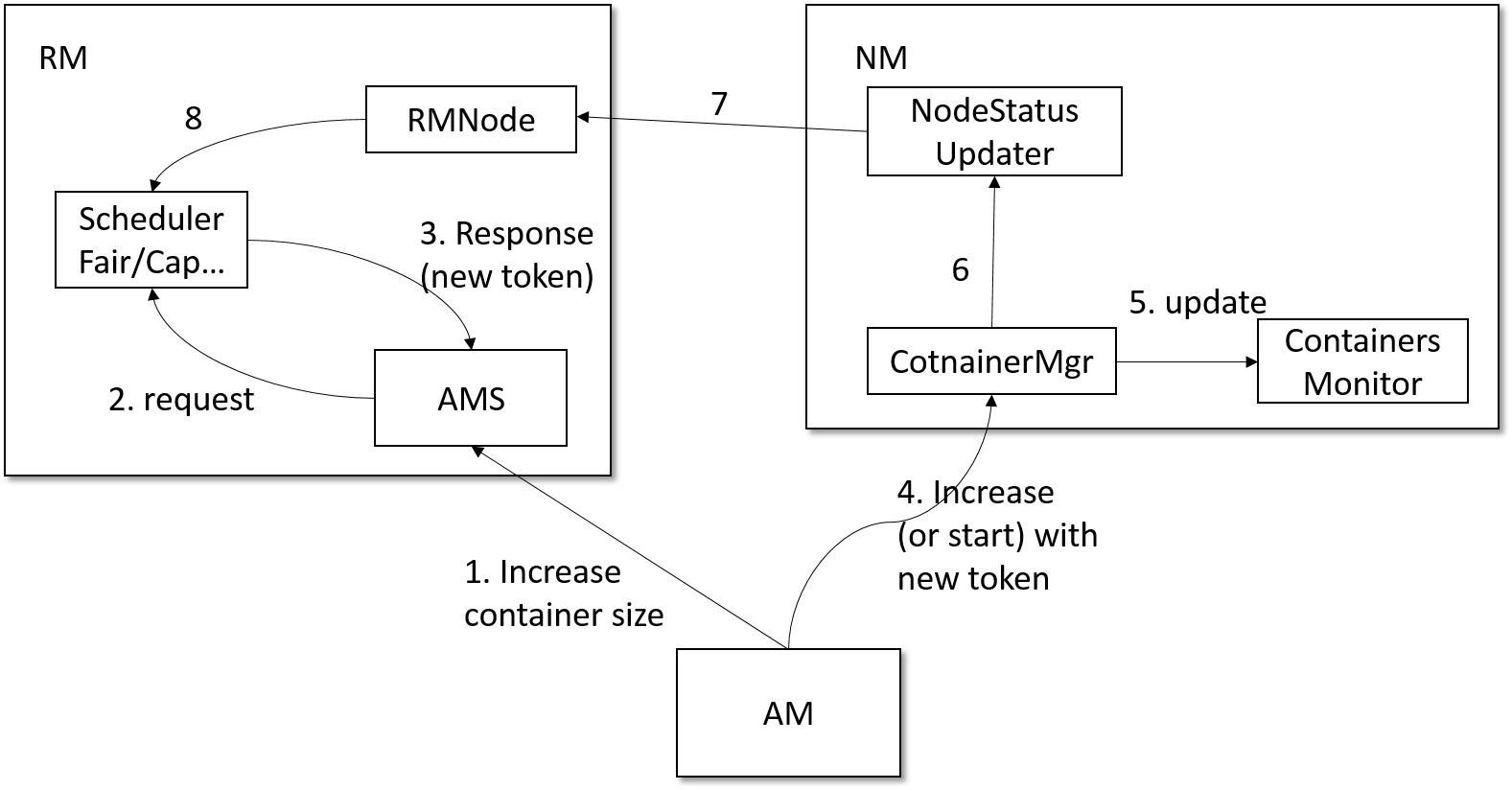


图 4‑13 增加container资源流程

在处理Container资源增加流程时，引入了新的过期计时器Container Resource Increase Expirer（CRIE）用以确保Container资源在token时间内增加成功。CRIE将通过container ID跟踪当前Container以及其原有资源占有量，这样可以防止token过期后Container资源没法回到之前状态的情况。

Container资源增加失效以及成功的逻辑流程图如下4-9、4-10所示：

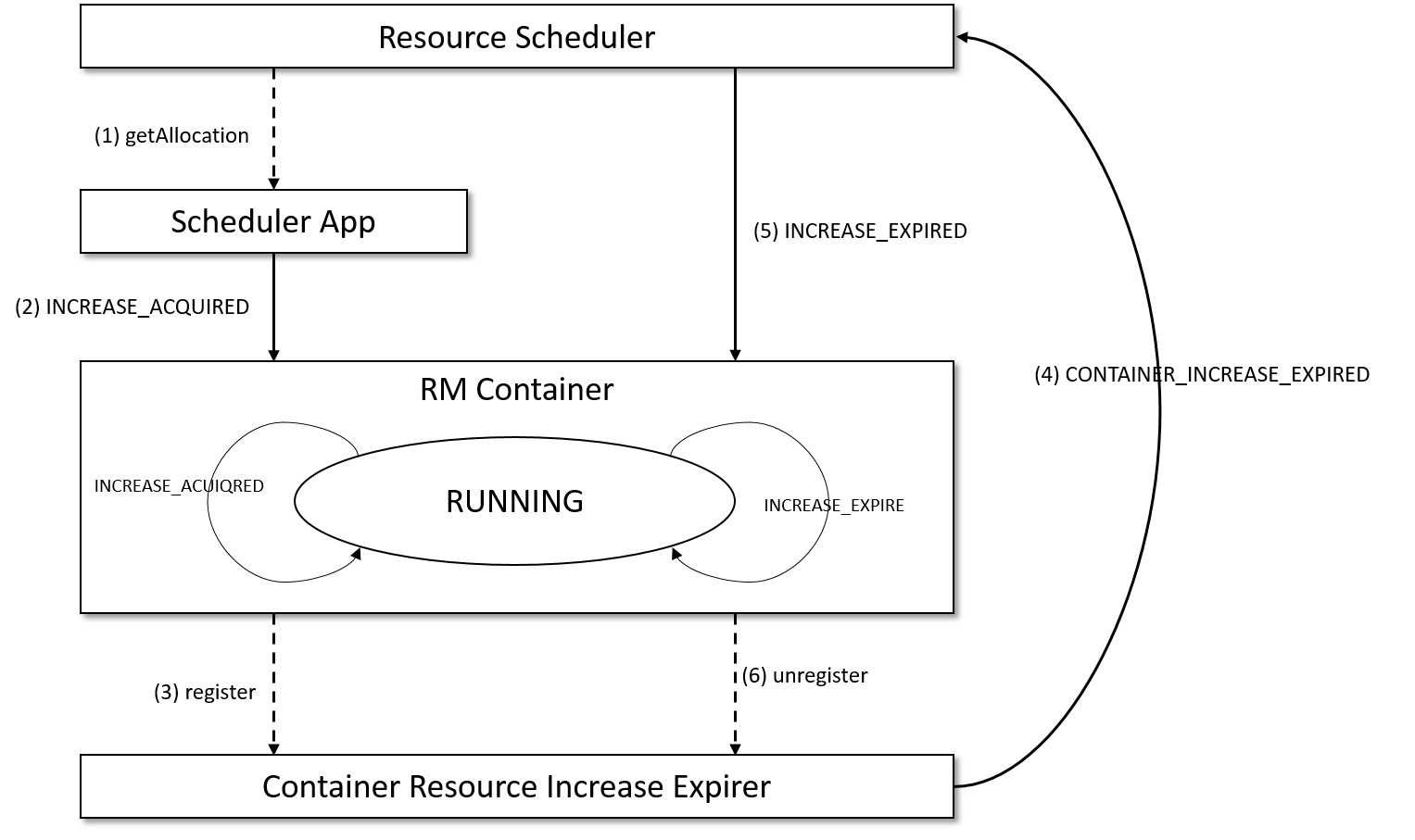


图 4‑14 Container资源增加失效逻辑图

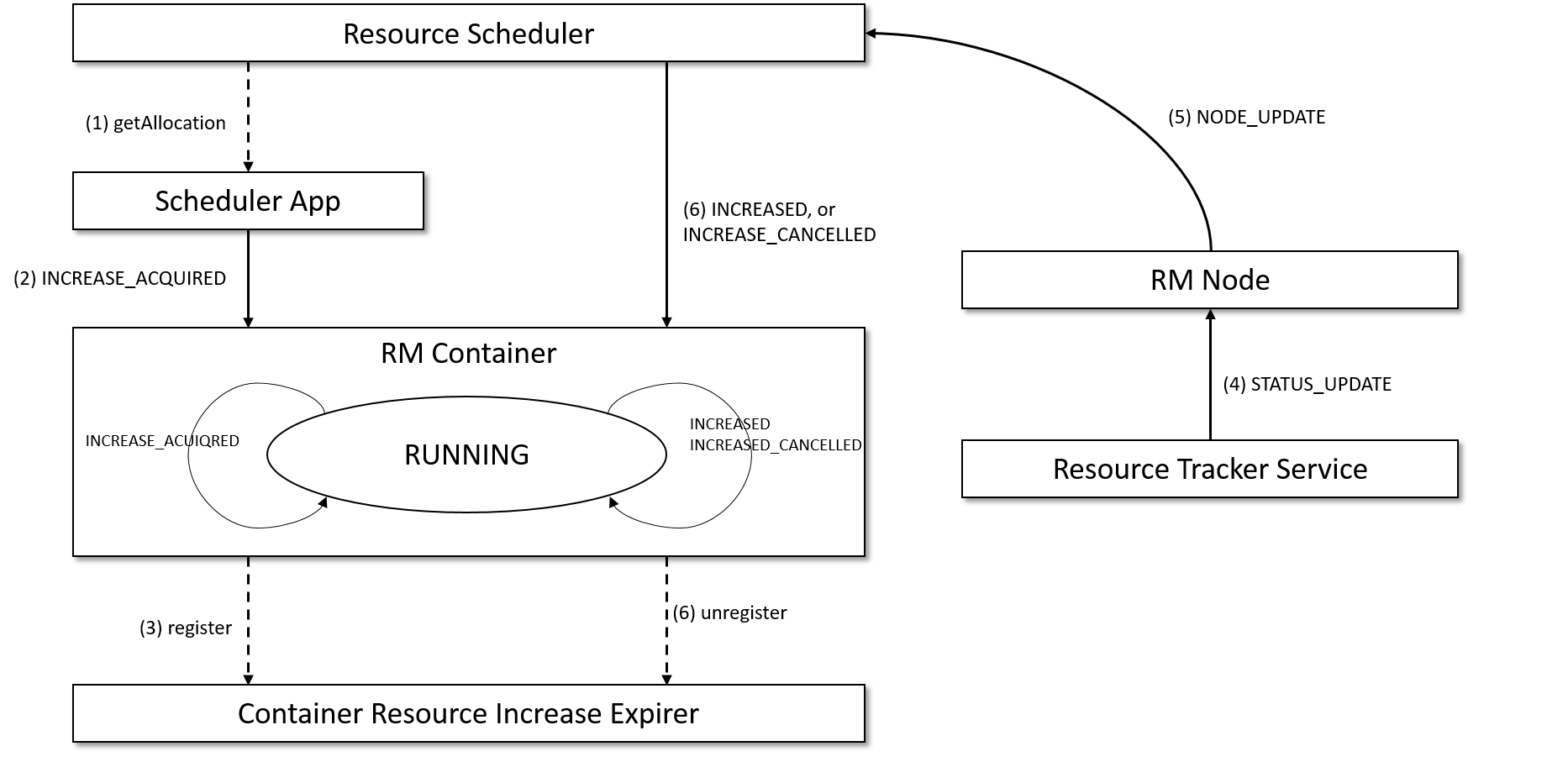


图 4‑15 容器资源增加成功流程图

而由于资源的缩减仅仅涉及到Container自身状态的管理，因此整体流程较增加更为直观简单。资源减少的抽象流程图如4-11所示：

1. Application Master发送资源更改请求给Resource Manager，并指明相应的Container ID以及目标容量
2. Resource Manager在调度周期中对相应Container减少资源
3. Resource Manager将Container资源减少信息发送给RMNode以等待Node Status Updater在下次心跳时拉取。
4. Node Manager从心跳信息回复中获取回复信息
5. Node Manager将减少资源消息发给Container Manager，并且更新自身的状态信息
6. Container Manager通知Container Monitor自己的状态变更，并且更新自己的状态。

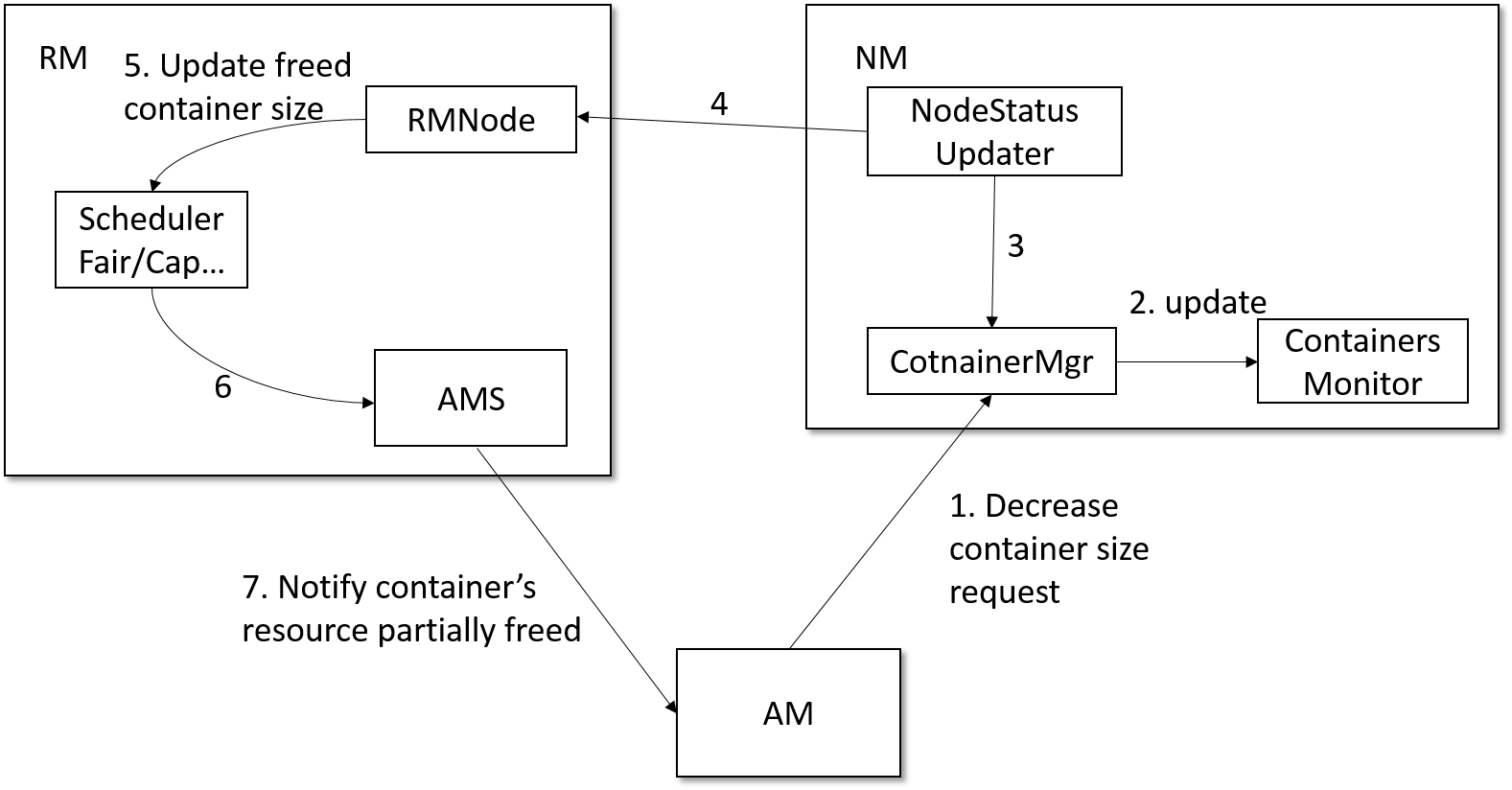


图 4‑16 缩减Container资源流程

在资源变更完成之后，Yarn会进入资源调度流程。资源调度器将首先处理已存在的Application请求，然后处理普通的新容器资源请求。在这两种情况下，资源增加请求都会比普通的新容器请求具有更高的优先级。这样即可保证在同一时间，只有一个资源申请请求能够被Container处理。

## 基于微服务的社区应用服务管理子系统

### 服务规划

系统采用微服务架构对所有应用服务进行规划，抽象出应用服务、应用通用服务以及数据存储服务三类服务。其中，应用通用服务为系统中应用服务提供诸如安全认证、用户通讯、身份管理等通用基础服务；应用服务实现具体的业务逻辑并选择使用最符合自身数据存储需求的存储服务；数据存储服务提供关系型数据库MySQL、文档型数据库MongoDB、缓存数据库Redis以及文件存储系统HDFS。

智慧社区应用服务整体规划如图4-17所示：

1） 应用服务包括党建服务、居委会工作、居民生活、物业服务、便民服务五大类，该类服务均使用关系型数据库MySQL作为后端存储，以Restful API向外部暴露服务，并调用安全服务进行安全认证，使用支付服务提供党费缴纳、物业缴费等服务支付功能。

2） 应用通用服务包括邻里社交服务、基础信息服务以及支付服务三大类。其中邻里社交服务使用MongoDB作为聊天信息存储数据库，以Redis数据库作为用户在线状态缓存数据库。以HDFS文件系统作为聊天信息中图片、视频、文件等存储系统。

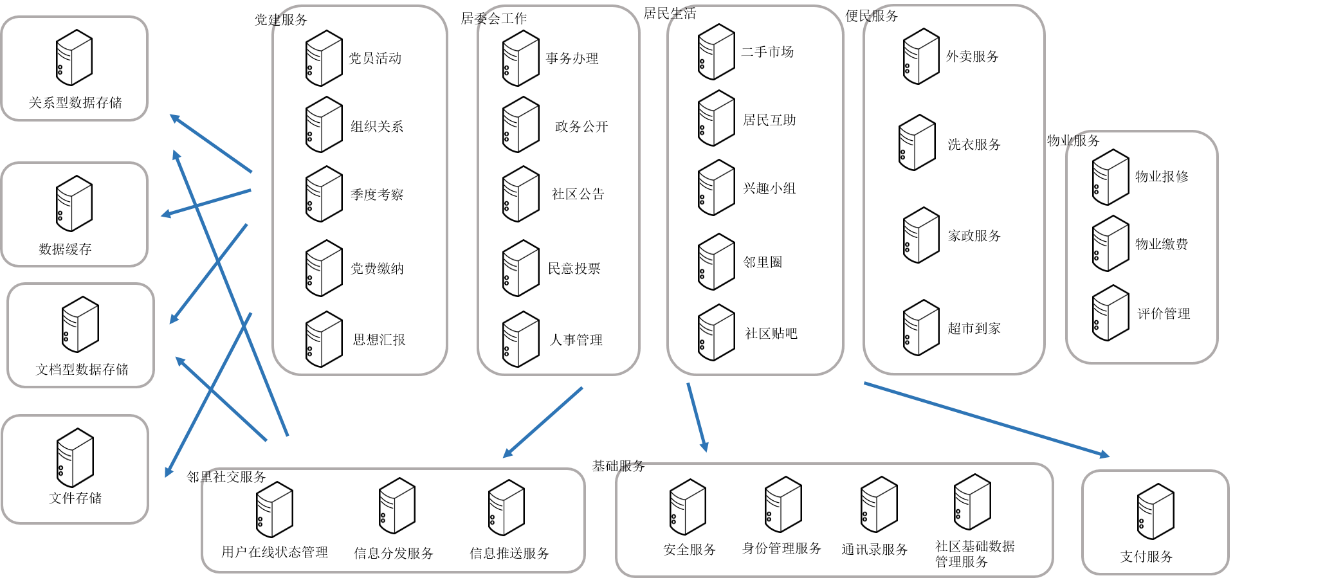


图 4‑17 智慧社区应用服务整体规划

### 服务治理

目前系统中服务治理的需求包括：

1） 当服务数量不断增大时，服务URL的配置管理变得非常困难。需要一个服务注册中心，动态注册和发现服务，使服务的位置透明。并通过在服务消费方获取服务提供方地址列表，实现服务的软负载均衡和Failover，以降低对F5一类硬件的依赖。

2） 当服务数量进一步增多，服务间依赖关系将变得错综复杂，此时需要服务治理中心能够提供服务间依赖关系图，以帮助开发人员理清服务间关系。

3） 随着服务调用量增大，服务的容量问题将变得日益关键。怎样判断一个服务需要多少服务器的支撑，何时需要对服务进行扩容都是需要考虑的问题。而为了解决这样的问题，需要将服务的调用量、相应时间进行相应监控统计，作为容量规划的参考指标。

针对上述需求，设计服务治理架构图如图4-18所示：

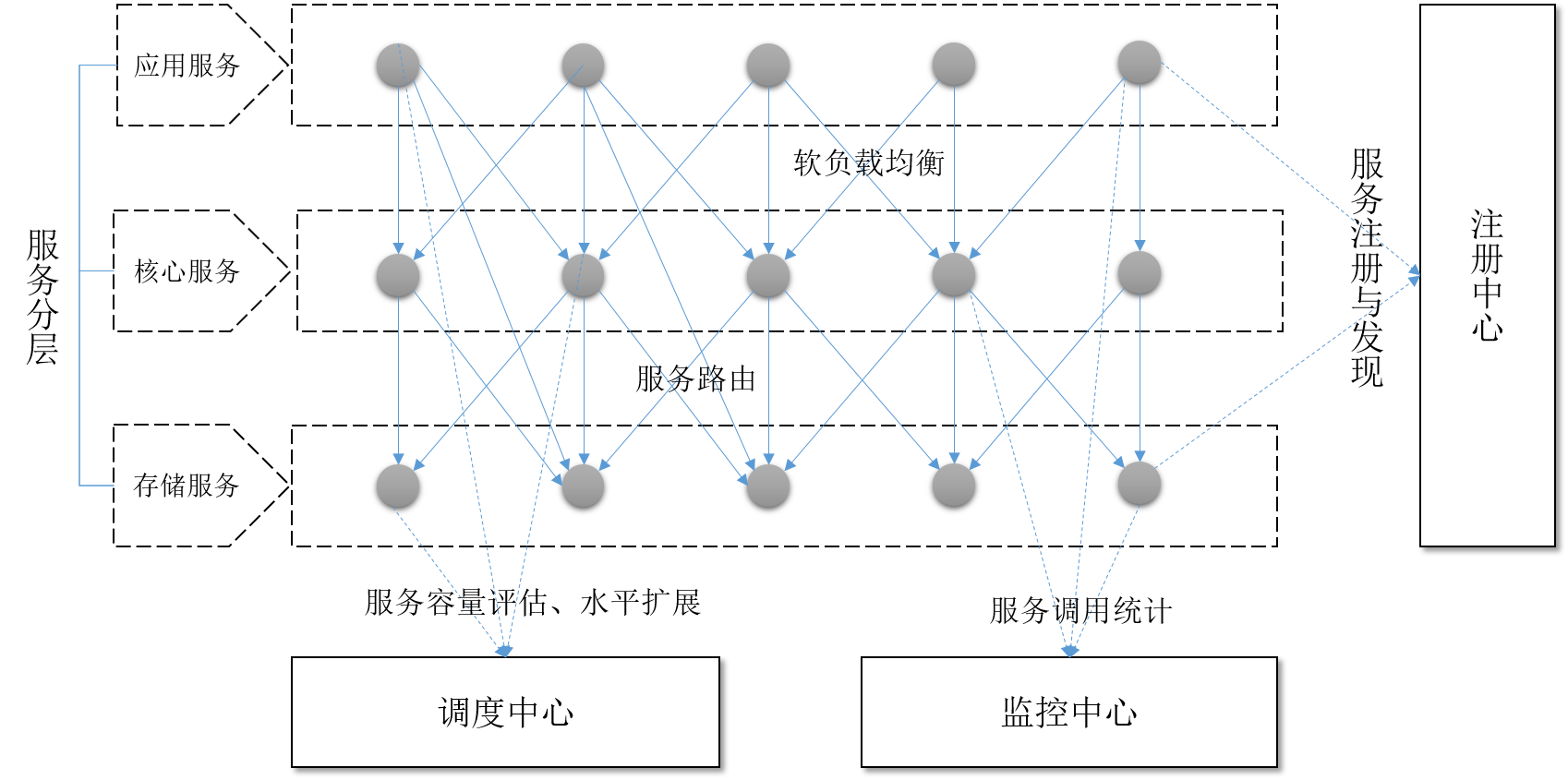


图 4‑18 服务治理架构图

一、注册中心：

为保证服务注册中心的高可用性，系统选用Apache Zookeeper作为服务注册中心。注册中心提供服务发现与注册功能，系统内所有服务间均需要向服务中心注册自身服务，并从服务中心中查找所需服务，获取相应信息后进行调用。

服务注册、发现实例如图4-19所示：

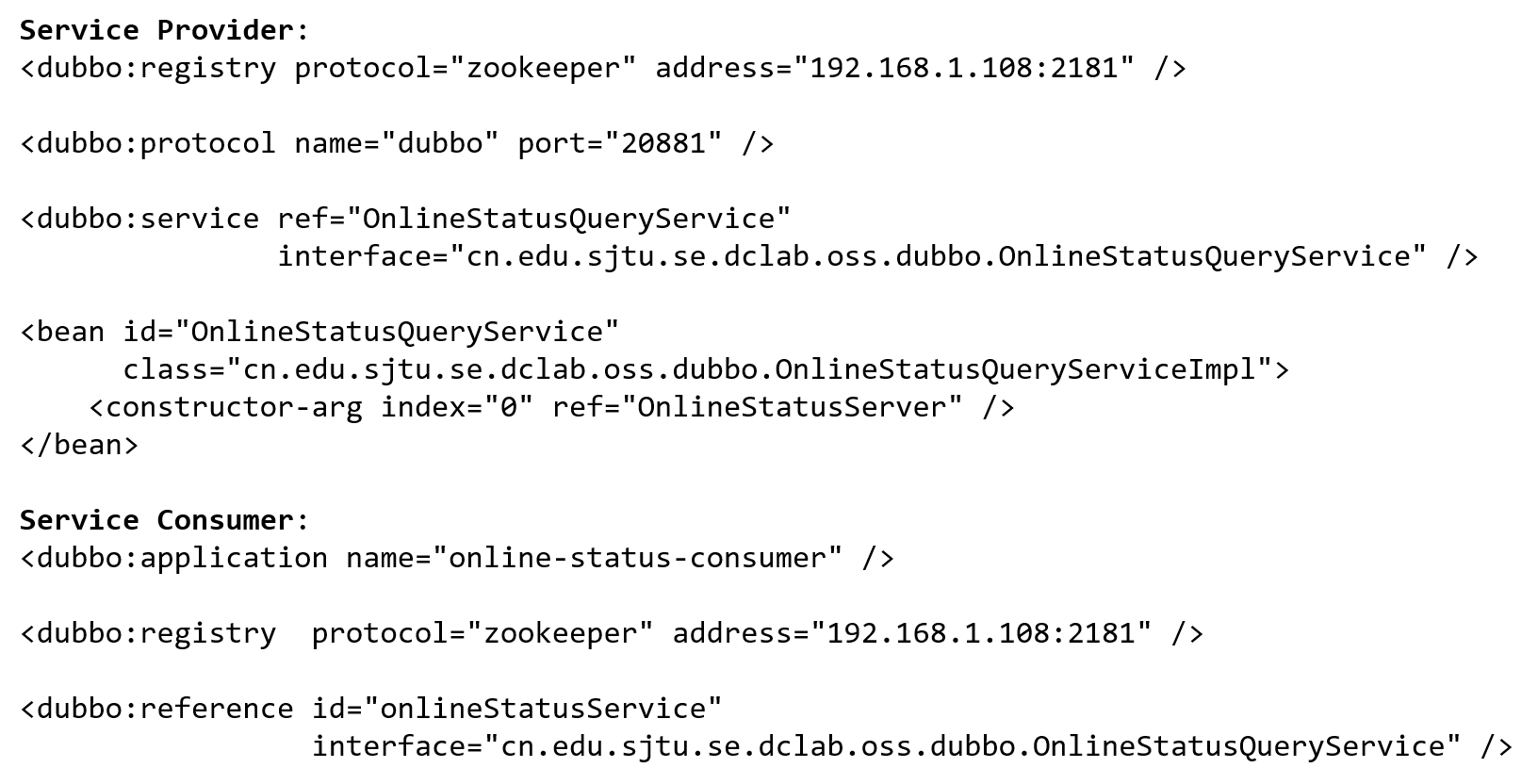


图 4‑19 服务注册、发现实例

服务提供方仅需在配置文件中加入注册中心地址，并将相应服务通过dubbo协议进行配置即可。而服务消费者同样只需配置注册中心地址以及所需消费服务的名称即可像使用本地服务一样进行服务调用。

二、监控中心

监控中心提供应用、服务、主机、注册中心、服务器五大监控模块。监控中心截图如图4-20所示：

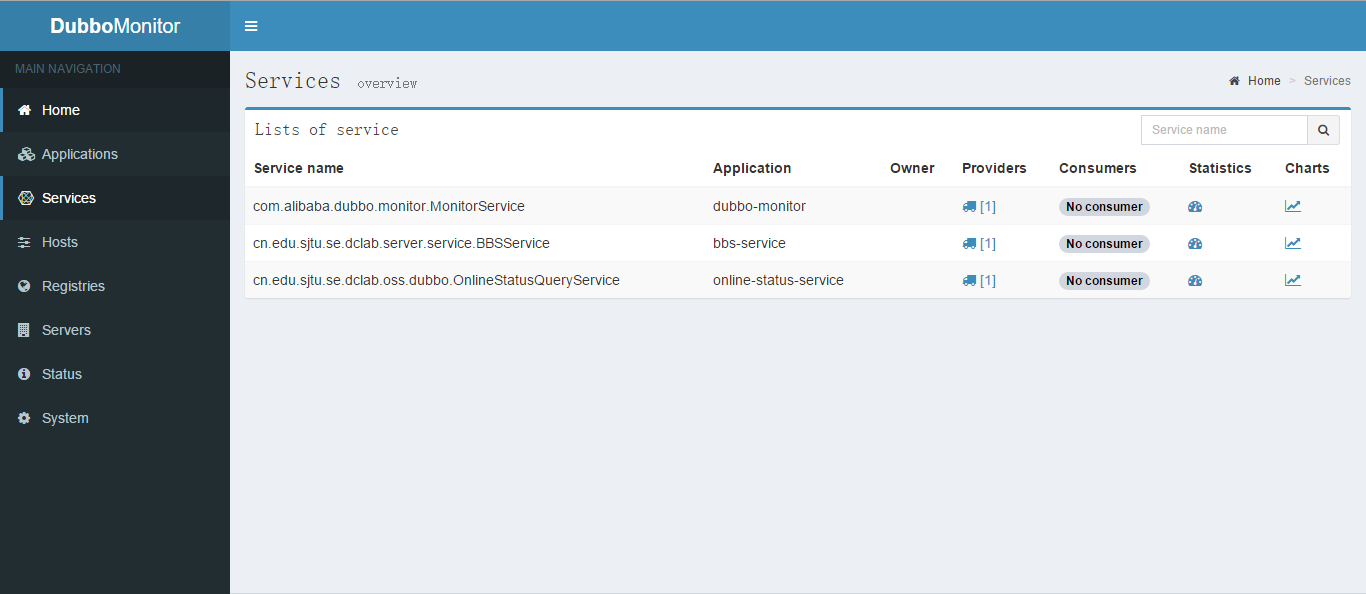


图 4‑20 监控中心截图

1） 应用监控：提供应用所属、提供者、消费者、依赖关系相关信息。

2） 服务监控：提供服务提供者、消费者、调用统计、调用图表信息。其中调用统计包括方法名，成功、失败次数、平均调用时间、最长调用时间、最大并发数等信息。

3） 主机监控：提供主机列表，并提供相应主机上服务提供者、消费者统计。

4） 注册中心监控：提供注册中心列表，并提供相应注册中心的注册、订阅者信息。

5） 服务器监控：提供服务器列表及服务器客户端监控信息。

三、服务路由

服务路由模块提供根据具体信息对服务提供者进行筛选的功能。服务路由模块目前支持基于表达式的条件路由规则设置。图4-21为服务路由实例。

条件过滤规则为：

* + "=>"之前的为消费者匹配条件，所有参数和消费者的URL进行对比，当消费者满足匹配条件时，对该消费者执行后面的过滤规则。
  + "=>"之后为提供者地址列表的过滤条件，所有参数和提供者的URL进行对比，消费者最终只拿到过滤后的地址列表。
  + 如果匹配条件为空，表示对所有消费方应用，如：=> host != 10.20.153.11
  + 如果过滤条件为空，表示禁止访问，如：host = 10.20.153.10 =>

条件过滤表达式定义为：

* 条件支持：
  + 等号"="表示"匹配"，如：host = 10.20.153.10
  + 不等号"!="表示"不匹配"，如：host != 10.20.153.10
* 值支持：
  + 不等号"!="表示"不匹配"，如：host != 10.20.153.10
  + 不等号"!="表示"不匹配"，如：host != 10.20.153.10
  + 不等号"!="表示"不匹配"，如：host != 10.20.153.10

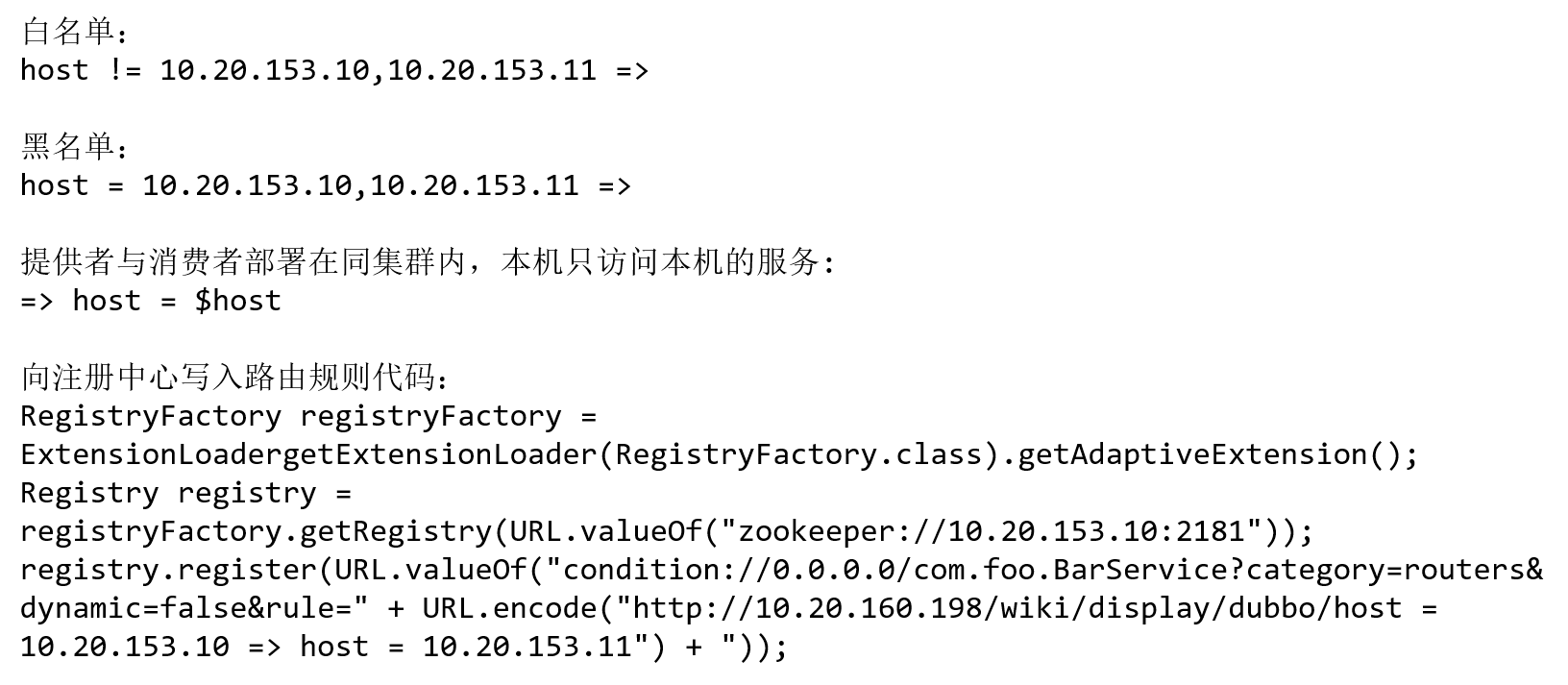


图 4‑21 服务路由实例

四、软负载均衡

软负载均衡模块提供软件层面服务消费者选择服务提供者的策略。模块默认提供四种负载均衡策略：

* + Random: 随机选择服务，调用量越大请求分配越平均，可能存在某个截面请求碰撞的情形。
  + Round-robin: 轮训，按服务权重设置轮询比例。可能存在请求累积情形，例如，第二台服务器响应慢，当请求调到第二台服务器时出现停滞，久而久之大量请求将停滞于第二台服务器上。
  + Least-active: 对于响应慢的机器少分配请求，对于相应快的多分配。
  + Consistent-hashing： 一致性哈希，相同参数的请求总是发到统一提供者。

### 服务水平扩展

服务的水平扩展模块根据服务监控信息对服务目前的负载做出评估，在当前容量不能满足服务需求时或在当前服务占用资源大大过剩时调用统一资源管理平台Yarn接口对服务进行相应地扩容或缩容。

目前容量评估基于服务方法平均调用时间，成功、失败次数，服务提供者、消费者数量等信息进行评估，具体评估流程如图4-22所示：

其中令服务平均调用时间为α，调用成功次数为λ，失败次数为β，服务提供者数量为p，服务消费者数量为c。

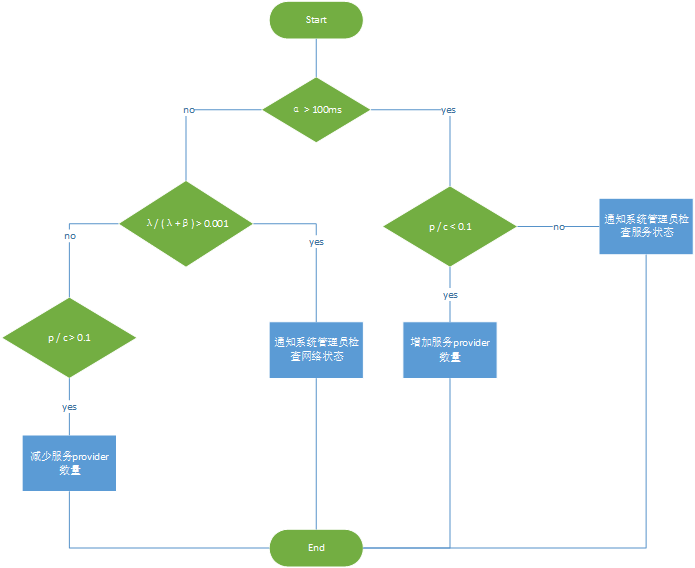


图 4‑22 服务评估流程

１）检查服务平均调用时间α

a）若大于100ms，进入步骤2

b）若小于或等于100ms，进入步骤3

2) 检查服务提供者p与消费者c数量比例

a）若p / c小于0.1，则表明服务消费者数量过多，造成大量请求在提供者处拥塞，因此需要添加provider数量。

b）若p / c 大于等于0.1，则表明服务消费者与提供者数量处于一个合理比例，而此时服务响应速度仍不能达到系统要求。需要通知系统管理员，在收集更多的监控信息后进行更为全面的决定。

3）检查服务调用失败次数与服务调用总次数比例

a）若大于0.001，则表明服务间网络抖动明显，需要通知系统管理员，检查系统网络。

b）若小于等于0.001，则进入步骤4

4） 检查服务提供者p与消费者c数量比例，若比例大于0.1，则表明服务运行良好，且服务提供者数量过多，因此需要减少服务提供者的数量。

## 本章小结

本章重点介绍了智慧社区应用服务与资源管理系统核心功能的设计与实现，包括邻里社交子系统、支持长服务的资源管理子系统、基于微服务的社区应用服务管理子系统三大核心子系统。

# 系统测试及应用

本章对智慧社区应用服务与资源管理系统进行系统测试及应用。首先对系统进行功能和性能测试，然后以部署运行在杨浦区的系统为例，对系统进行可行性及有效性分析。

## 系统测试

### 功能测试

本次测试将智慧社区应用服务与资源管理系统分为三部分进行测试：邻里社交子系统、支持长服务的资源管理子系统、基于微服务的社区应用服务管理子系统。

1）邻里社交子系统

邻里社交子系统的测试用例及测试结果如表5-1所示：

表格 5‑1 邻里社交子系统测试用例及测试结果

Table 5-1 Test cases and results for neighborhood social network subsystem

|  |  |  |
| --- | --- | --- |
| 功能 | 测试用例 | 测试结果 |
| 点对点聊天 | 测试系统中居民与居民间，居民与政府管理人员、居民与服务人员、政府管理人员与服务人员间点对点聊天功能 | 各类不同用户间均可进行点对点聊天  测试通过 |
| 好友管理 | 测试用户对于好友的添加、删除、拉黑、备注功能 | 能够正确对好友进行添加、删除、拉黑、备注  测试通过 |
| 群组聊天 | 测试系统中群组成员间聊天功能 | 群组成员间能够进行实时聊天，一个成员发送的聊天消息能被其他成员所接收。  测试通过 |
| 群组管理 | 测试用户对群组的创建、加入、退出、删除、备注功能 | 用户可对群组进行创建、加入、退出、删除、备注操作。  测试通过 |

表格 5‑2 邻里社交子系统测试用例及测试结果（续）

Table 5-1 Test cases and results for neighborhood social network subsystem (Cont.)

|  |  |  |
| --- | --- | --- |
| 功能 | 测试用例 | 测试结果 |
| 用户在线状态管理 | 测试用户在线状态的更新、查询 | 用户在线心跳信息能够及时反应到在线状态变化上  测试通过 |

2）支持长服务的资源管理子系统

支持长服务的资源管理子系统测试用例及测试结果如表5-2所示：

表格 5‑2 支持长服务的资源管理子系统测试用例及测试结果

Table 5-2 Test cases and results of resource management system for long-lived services

|  |  |  |
| --- | --- | --- |
| 功能 | 测试用例 | 测试结果 |
| 服务注册与发现 | 测试tomcat集群、Storm集群运行实例，并测试在集群中节点故障时集群服务可用性 | tomcat、Storm集群均可运行于子系统上，且在集群中节点故障时可用性不受影响  测试通过 |
| 日志管理 | 测试长服务运行时能否随时对其日志进行收集管理 | 在长服务运行时可以随时对其日志进行收集管理，并能够设置、变更日志管理策略  测试通过 |
| 容器资源伸缩 | 测试长服务运行时能否对容器资源进行变更 | 长服务运行时可对容器资源进行动态增加、减少  测试通过 |
| 服务水平扩展 | 测试服务在运行时能否动态对其进行水平扩展，增加、减少节点 | 服务在运行时能够对其进行水平扩展，增加、减少相应节点  测试通过 |

3）基于微服务的社区应用服务管理子系统

基于微服务的社区应用服务管理子系统测试用例及测试结果如表5-3所示：

表格 5‑3 基于微服务的社区应用服务管理子系统测试用例及测试结果

Table 5-3 Test cases and results for community application service management system based on micro-service structure

|  |  |  |
| --- | --- | --- |
| 功能 | 测试用例 | 测试结果 |
| 服务发现与注册 | 测试各类应用微服务间能否互相发现、调用，并注册自身服务 | 微服务能够自动注册自身服务，并发现所依赖服务位置，进行正确调用。  测试通过 |
| 服务监控 | 测试系统是否能够对应用、服务、主机、注册中心相关信息进行监控 | 能够对应用、服务、主机、注册中心相关信息进行监控  测试通过 |
| 服务路由 | 测试系统能否根据服务路由对服务提供者、消费者进行动态筛选和过滤 | 能够通过服务路由对服务提供者、消费者进行动态筛选过滤  测试通过 |
| 负载均衡 | 测试系统预设Random、Round-robin、Least-active、Consistent-hashing负载均衡算法效果 | 在大量客户请求环境下各个负载均衡算法如预期运行  测试通过 |

### 性能测试

本节分别对对邻里社交子系统、资源管理子系统进行性能测试。结果包括社交系统用户并发量，未改进资源管理系统与改进后性能比较，以及应用微服务化前后服务响应时间对比。

一、邻里社交子系统

1）测试环境

本次测试mqtt接入服务器，rabbitmq消息中间件服务，用户在线状态服务，社交信息分发、推送服务，Redis存储服务各占一台服务器，具体测试系统环境如表格5-4所示：

表5-4 邻里社交子系统测试环境

Table 5-4 Test environment for neighborhood social network system

|  |  |
| --- | --- |
| TYPE | CPU/MEMORY/DISK/NETWORK |
| MQTT接入服务器 | 2.2GHz 2 Core / 2GB / 200G/ 100Mbs |
| Rabbitmq消息中间件 | 2.2GHz 2 Core / 2GB / 200G/ 100Mbs |
| 用户在线状态服务 | 2.2GHz 2 Core / 2GB / 200G/ 100Mbs |
| 社交信息分发、推送服务 | 2.2GHz 2 Core / 2GB / 200G/ 100Mbs |
| Redis存储服务 | 2.2GHz 2 Core / 2GB / 200G/ 100Mbs |

2）测试方案

实验通过分布式测试应用Apache Jmeter以及jmeter-mqtt plugin结合模拟社交用户连接，测试参数配置如图5-1、5-2所示：

* 测试共使用3台jmeter服务器
* 每台服务器共启动1000个thread，每次测试循环运行10次
* 协议使用信息QoS级别为At least once级别
* 选用Retained选项标示需要服务器在用户离线时持久化消息直到用户上线
* 采用公网IP进行测试，尽力模拟真实环境

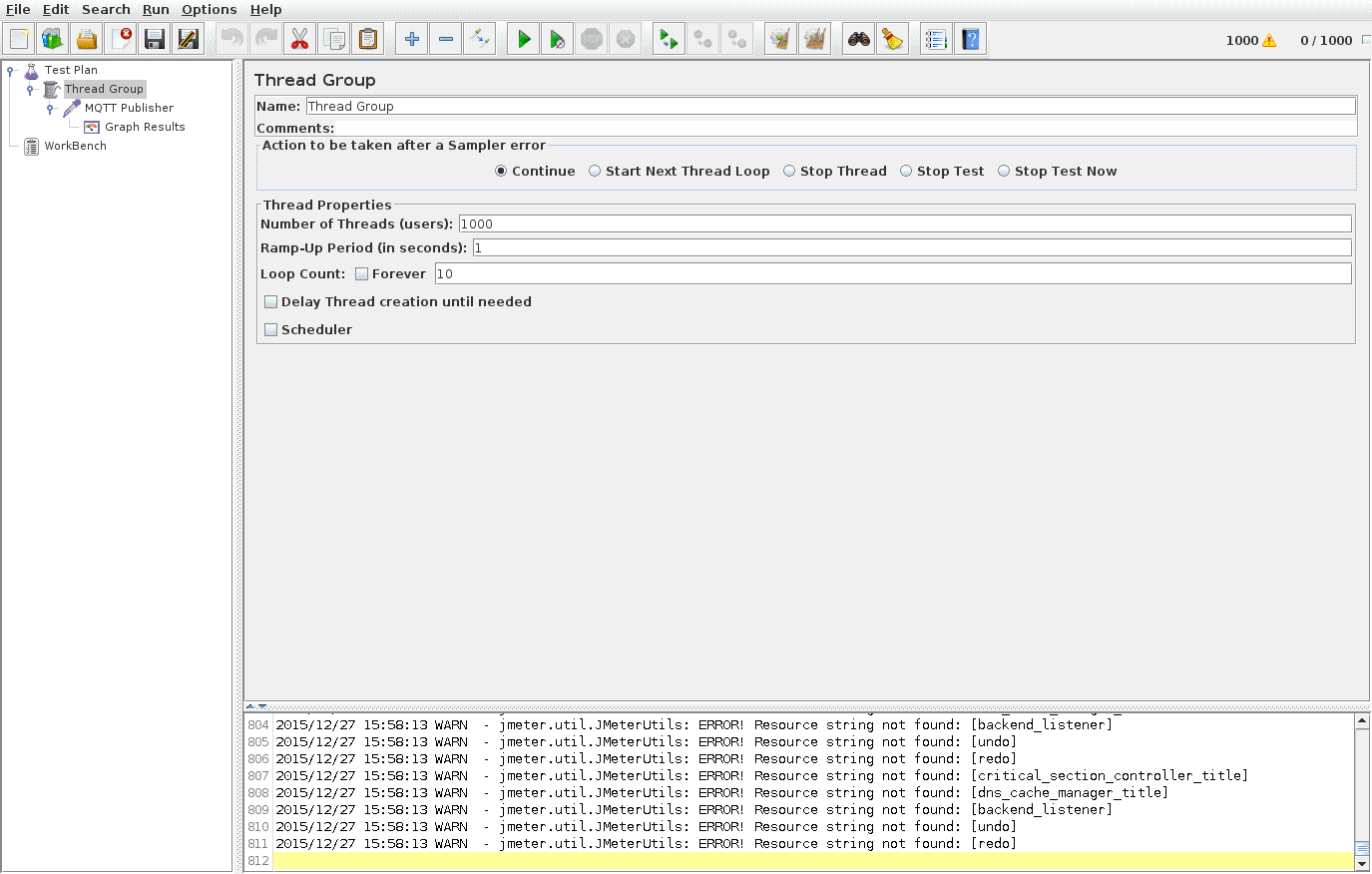


图 5‑1 测试并发度配置

Figure ‑ concurrency test configuration

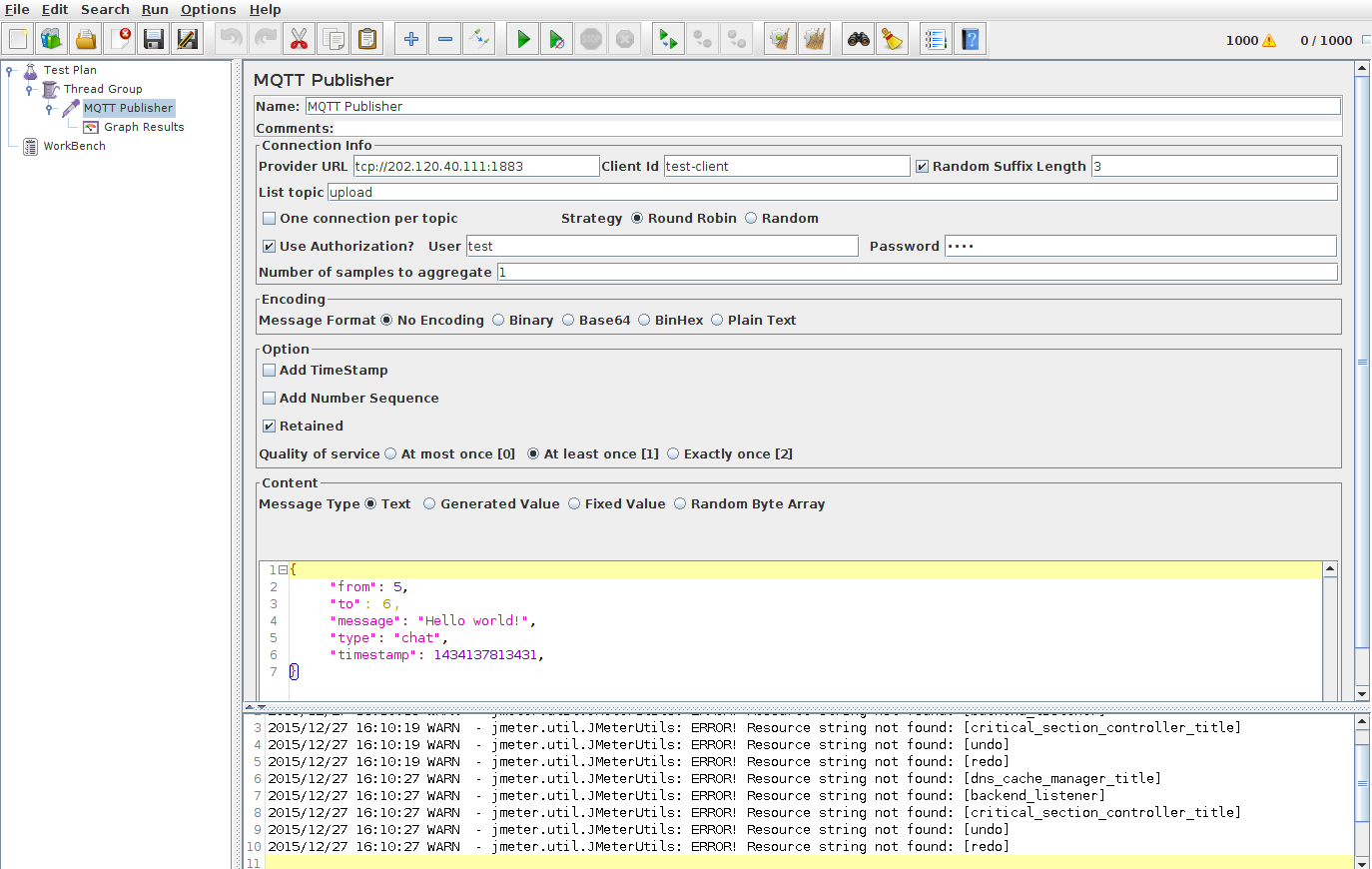


图 5‑2 MQTT协议信息配置

Figure ‑ MQTT-protocol message configuration

3）测试结果及分析

系统测试结果如图5-3所示：

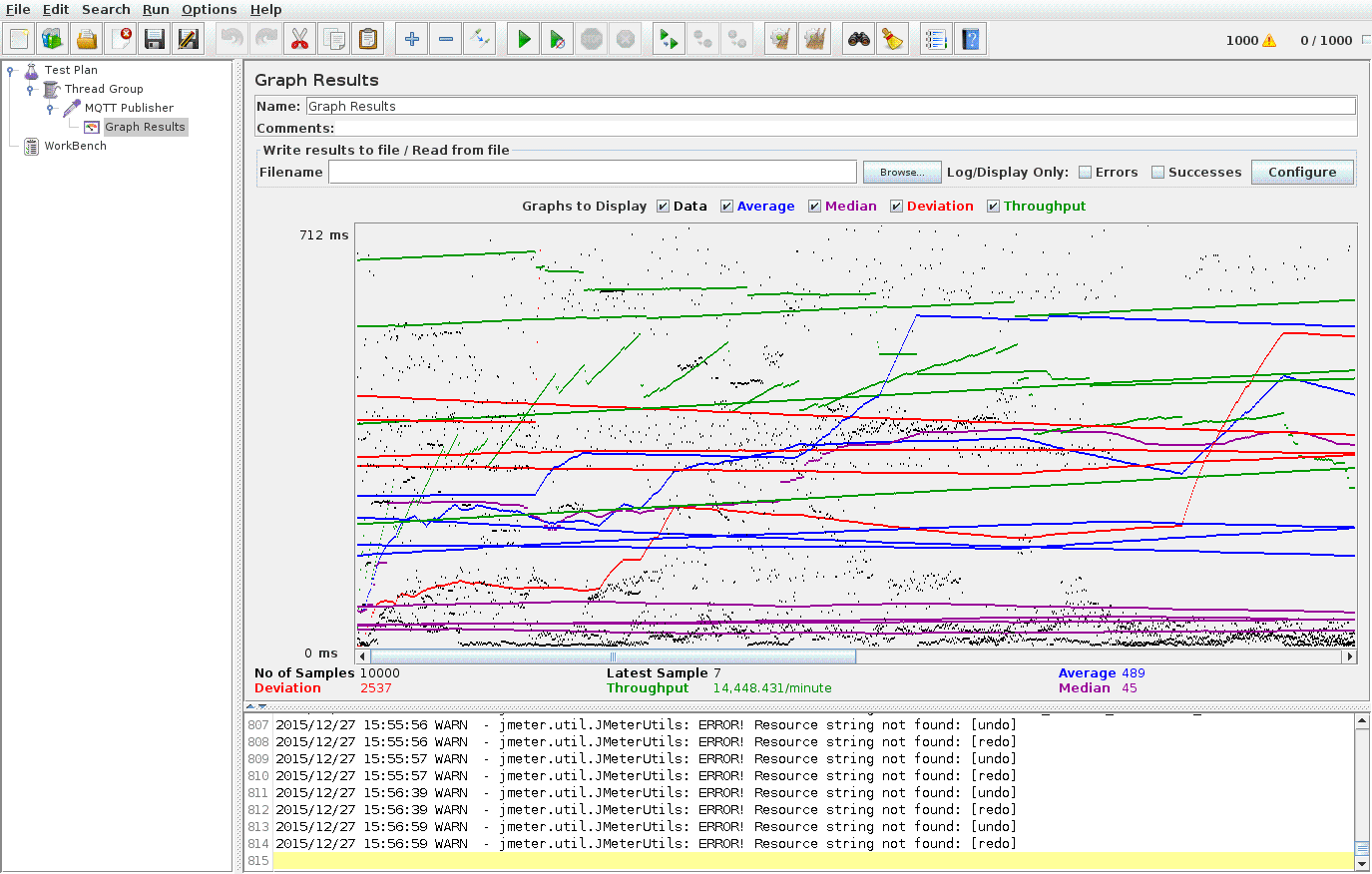


图 5‑3 并发度测试结果

Figure ‑ Test result for concurrency test

如图所示，单台双核2GHz, 2GB内存接入服务器的吞吐量为15000条/秒，CPU利用率接近90%，内存利用率50%。因此，邻里社交系统吞吐量完全符合系统需求。

二、支持长服务的资源管理子系统

1）测试环境

本次测试环境采用3台双核2.2GHz Ubuntu14.04虚拟机组成Yarn集群，一台双核2.2GHz Ubuntu14.04虚拟机运行zookeeper协调服务器。

2）测试方案

对资源管理系统测试分为两方面：

* + 测试改进前与改进后系统对于原生map-reduce模型性能的影响
  + 测试运行在资源管理系统上长服务与原生长服务间的性能差别

对于改进前后对于原生map-reduce模型性能影响，我们选用四种不同的测试数据集大小分别运行同一个map-reduce任务，并比较任务间运行时间的差异。

对于在资源管理系统上长服务与原生长服务间的性能差别，测试选用Tomcat集群以及Storm集群进行分析，分别比较Tomcat集群独立运行与运行于本系统上的性能区别，以及Storm集群独立运行与运行于本系统上的性能差异。

3）测试结果及分析：

对于改进前后对于原生map-reduce模型性能影响测试结果如图5-1所示：

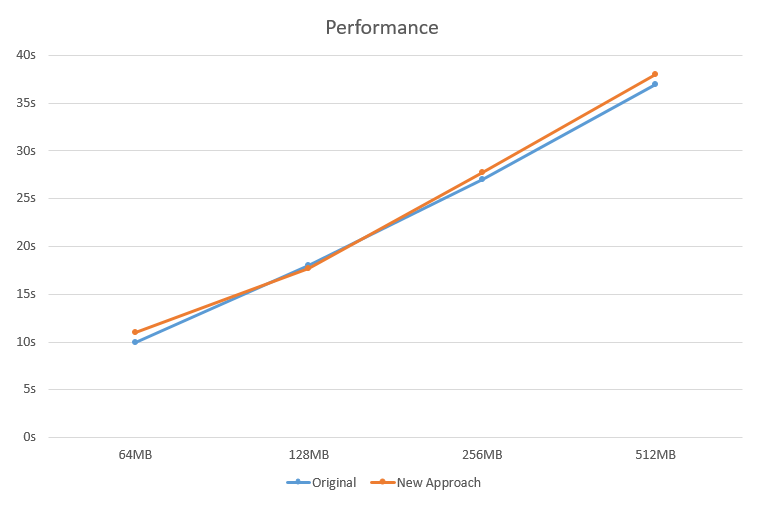


图 5‑4 原生map-reduce模型性能测试结果

Figure ‑ Performance test result for map-reduce model

如图所示，我们使用四种不同大小的测试数据集运行map-reduce任务，结果表明改进后的系统并未对原生系统运行模型带来任何的性能开销。

对于运行于或不运行于本系统上的性能开销测试结果如图5-2所示：

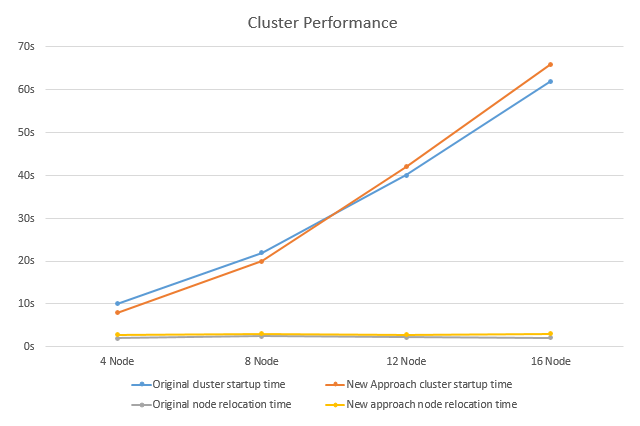


图 5‑5 集群运行性能测试结果

Figure ‑ Cluster performance test result

如测试结果显示，在集群启动时间上，随着集群节点的增多，改进后Yarn上运行的集群启动时间稍长，这是由于集群每个节点在启动时需要向服务中心注册自身信息造成的通信overhead引起的。而在集群节点迁移时间上，运行于Yarn上的服集群与原生集群间性能没有差别。

综合系统整体改进方案看，两系统间造成的主要性能开销增大来自于节点启动及运行时与服务注册中心通信的开销上，而这一类通信频率低，且是数据中心内部通信，因此对整体系统性能影响小。

## 系统应用

### 运行实例

本文以杨浦区社区治理系统为例，介绍本系统邻里社交、资源管理、应用服务管理子系统的应用。

一、邻里社交子系统的应用

图5-3为居民邻里社交应用的使用效果截图：



图 5‑6 邻里社交应用使用效果图

Figure ‑ Screenshot of neighborhood social application

居民在登陆社区治理应用后可选择下方状态栏中联系人按钮进入联系人界面。联系人界面中显示居民当前所有好友以及当前所加入的群组信息。居民可选择任意联系人或群组进行聊天。同时，邻里社交应用提供聊天信息记录页面，提供最近联系人列表，方便居民快速查找。

图5-4为系统后台居民状态信息管理页面截图：



图 5‑7 系统后台在线状态管理页面截图

Figure ‑ Screenshot for backend online status management page

系统管理员可于系统后台实时监控查看目前用户在线状态，并查看在线用户连接、客户端版本等相关信息。同时可发送应用通知给用户。

二、资源管理子系统的应用：

系统测试环境采用4个节点（单节点4G内存，4核）搭建Yarn集群，使用1个节点运行服务注册中心zookeeper服务。资源管理子系统应用管理界面如图5-5所示，系统中共提交了三种类型的任务，map-reduce任务，Apache Tomcat Server集群以及Storm集群，其中三类任务都处于运行中状态，内存总消耗为10GB。

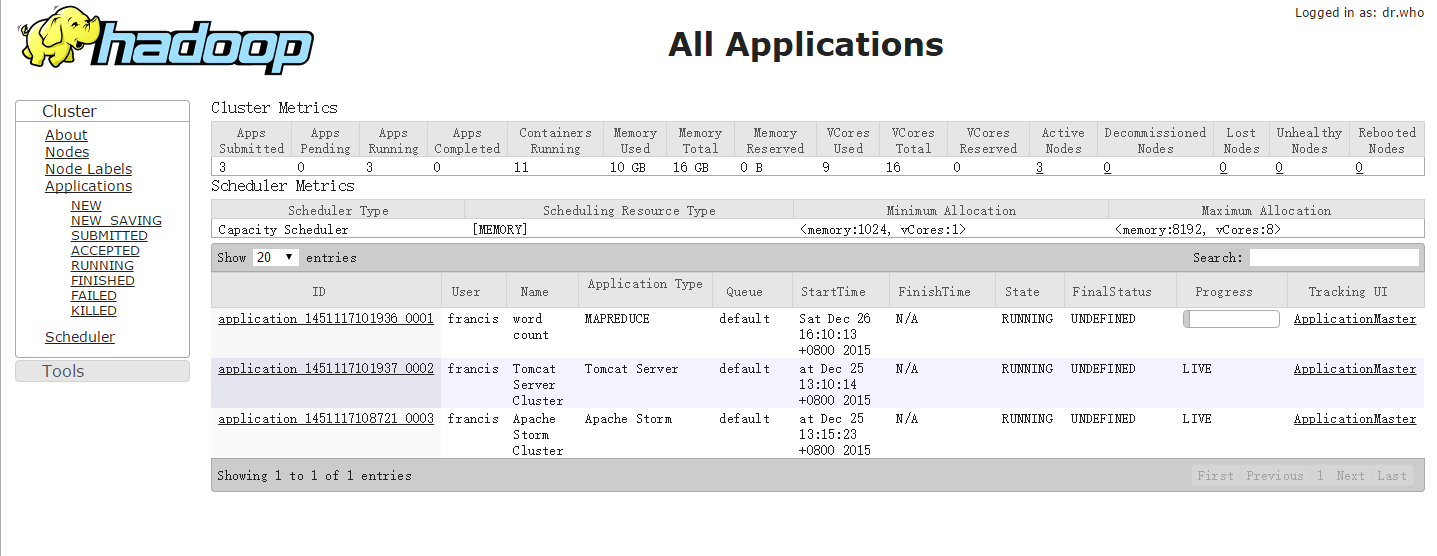


图 5‑8 资源管理系统管理界面

Figure ‑ Screenshot for resource management system admin page

其中，注册中心注册信息结构示意图如图5-6所示，注册中心中共注册了两类服务，一类为tomcat服务，一类为storm服务。这里可以注意到传统的map-reduce服务是没有被注册到注册中心的，这是因为这类服务的无状态特性及Yarn本身对其的完善管理决定的。tomcat服务共启动了3个container，而storm集群总共启动了一个nimbus节点以及3个supervisor节点。

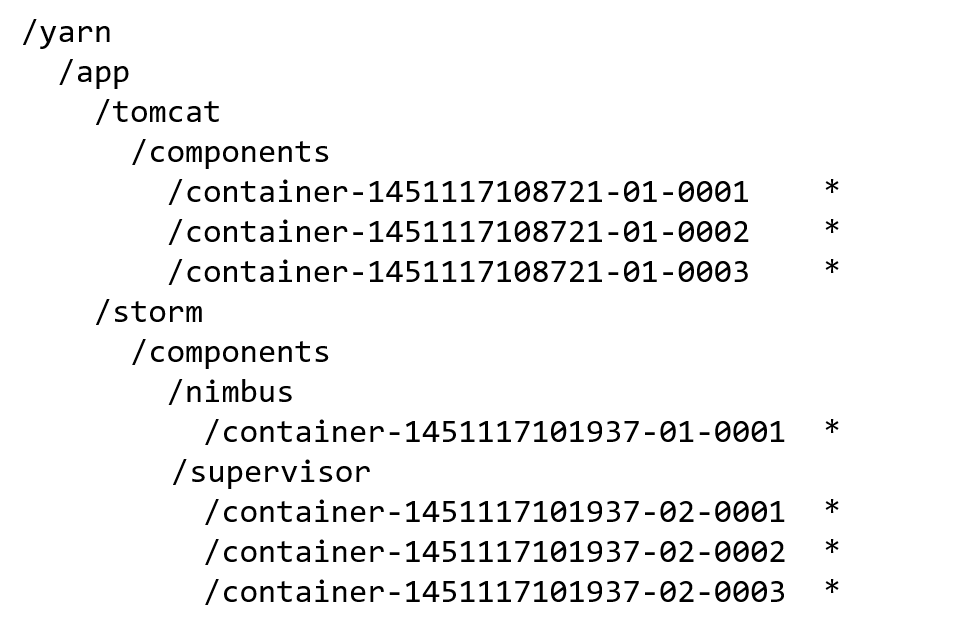


图 5‑9 注册中心信息结构示意图

Figure ‑ General view of service registration center data structure

三、应用服务管理子系统的应用

1）服务注册与发现

系统开发人员使用系统服务注册发现功能需要进行如下步骤的操作。

首先需要在pom.xml文件中引入dubbo依赖库，该依赖库用于处理服务发现、注册相关、监控信息发送、服务路由选择、负载均衡配置等功能。具体配置信息如图所示：

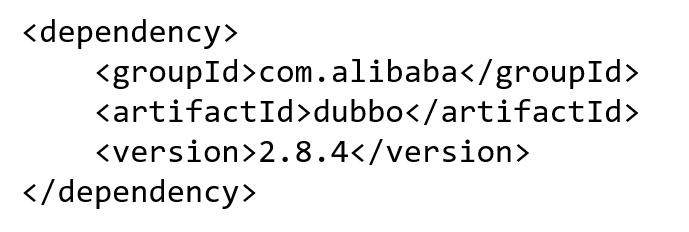


图 5‑10 依赖库引入实例

Figure ‑ dependency library configuration example

其次，需要定义服务接口并实现该接口，这里以系统中在线状态服务为例进行说明。在线状态服务接口定义如图5-8所示：

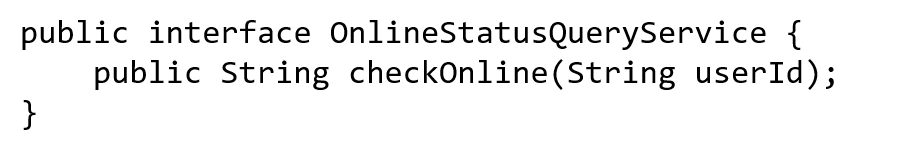


图 5‑11 服务接口定义实例

Figure ‑ service interface definition example

最后，在配置文件中配置该服务即可完成服务注册的所有工作，配置实例如图5-9所示：

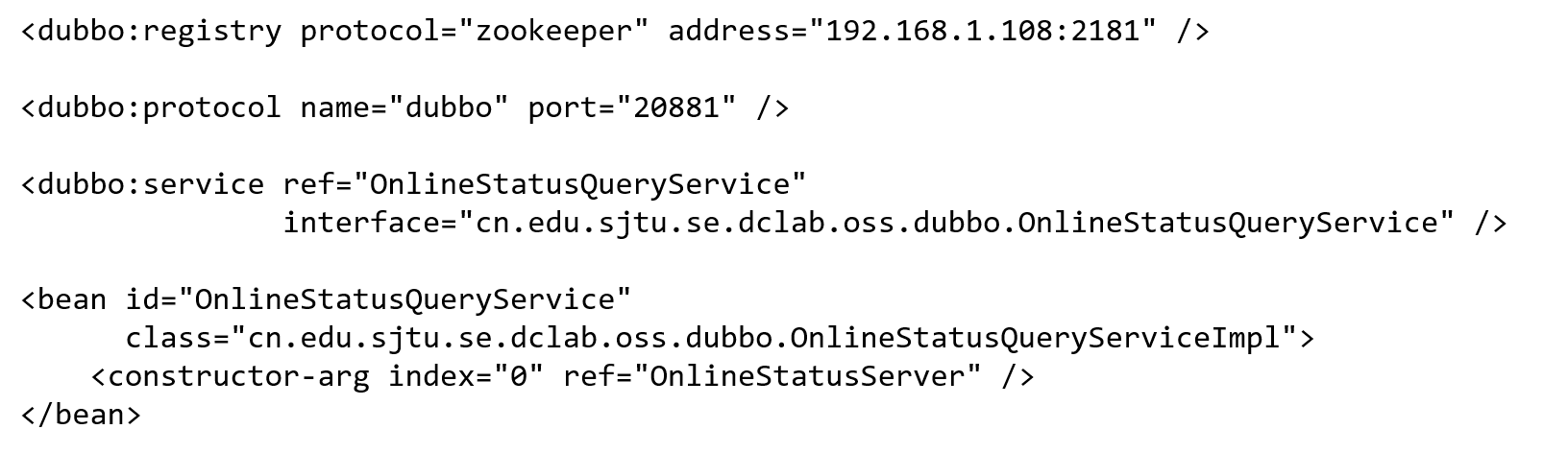


图 5‑12 服务配置实例

Figure ‑ service configuration example

在进行如上步骤后，服务在启动运行时将自动注册到服务注册中心，并可被系统中其他服务或应用使用。

2）服务监控

服务监控中心提供应用程序，服务，主机，注册中心等信息的监控，运行实例如图5-10所示：

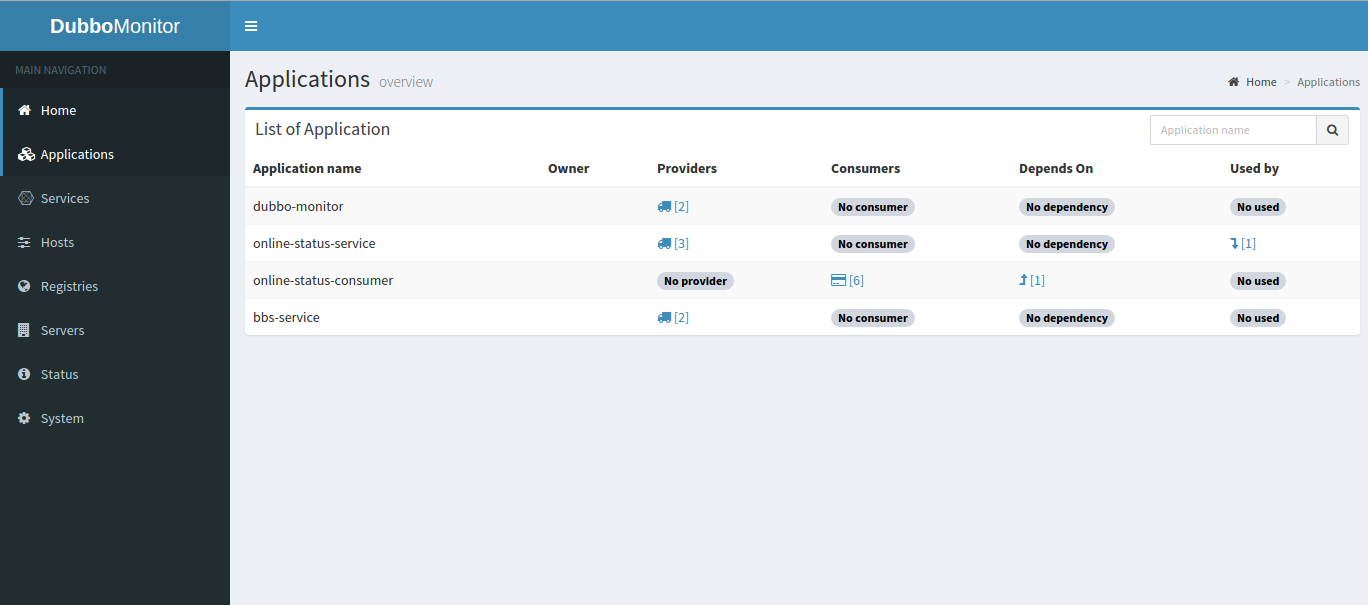


图 5‑13 服务监控中心运行截图

Figure ‑ Screenshot of service monitor center

系统管理员可以通过服务监控中心实时查看服务间依赖关系、服务调用频率，服务内的具体方法调用平均时间，最大调用时间等信息，从而为系统的运维管理提供依据。

### 应用效果及分析

本系统通过在杨浦区试运行，为杨浦区社区内政府管理人员、居民、服务人员提供邻里社交、党建、居委会工作、便民便利、物业管理、生活助手等服务。同时，系统应用通过应用服务管理与资源管理系统进行管理，简化了服务的部署、监控，同时提升了整个系统的资源利用率。整个系统的应用如表5-4所示：

表格 5‑3 系统的应用

Table 5-3 Application of the system

|  |  |  |
| --- | --- | --- |
| 服务 | 使用人群 | 服务功能 |
| 邻里社交 | 政府管理人员、居民、社区服务人员 | 提供社区内人员间即时聊天功能 |
| 党建服务 | 政府管理人员、居民 | 提供基于社区层次的党组织建设管理功能，包括党员活动、组织关系、季度考察、党费缴纳等功能 |

表格 5‑4 系统的应用（续）

Table 5-3 Application of the system (Cont.)

|  |  |  |
| --- | --- | --- |
| 服务 | 使用人群 | 服务功能 |
| 居委会工作 | 政府管理人员发布服务，居民、社区服务人员使用该服务 | 提供事务办理、政务公开、社区公告、民议投票等功能，将繁琐的办事流程迁移至互联网，方便社区内居民与管理人员的沟通 |
| 生活服务 | 居民 | 提供二手市场、居民互助、兴趣小组、邻里圈、社区贴吧等居民间交流的途径，增强社区居民交流 |
| 物业服务 | 居民 | 提供与物业公司的集成，为居民提供物业报修、缴费等功能 |

与此同时，系统为开发和运维人员提供了一套应用服务与资源管理的方案与系统，为应用开发及系统运维人员提供了更加便捷的系统资源管理，应用部署，服务治理支持。目前，在整个系统中，每日发生的服务调用次数约为50000次，共有微服务20个。系统的应用效果如下：

* 通过应用服务管理子系统提供的服务注册、监控、调度中心以及服务容量评估、软负载均衡功能，使得服务间完全解耦，管理难度大大降低，同时也提高了应用服务的开发效率。
* 通过资源管理子系统，服务器资源利用率得到大大提升，同时简化了整个集群的运维难度。

结果表明，本系统能够有效支持应用服务的管理以及系统集群资源的管理，降低了服务的开发、管理难度的同时，提升了集群的资源利用率，降低运维成本。因此，本系统是可行且有效的，达到了设计目标。

## 本章小结

本章首先对本系统进行功能测试和性能测试，然后以杨浦区社区治理系统的应用场景为例，分析了本系统的应用效果，并验证了本系统的可行性及有效性。

# 总结与展望

## 工作总结

本文以智慧邻里社区服务软件系统研发项目为背景，该系统旨在为居民、居委会、业委会、物业和其他服务机构等社区相关人员提供的一种智慧化的生活环境和新的社区管理形态。本文针对以上需求，设计并实现一个可有效支持系统需求的基于为服务架构的应用服务与资源管理系统。

本文工作主要包括以下几个方面：

1）对邻里社交模型进行抽象，设计并实现实名制邻里社交平台，该系统包括即时通讯、用户状态管理、应用消息推送三大服务，并实现提供易于使用的应用接口，简化外部服务的集成复杂度。

2）在Yarn基础上对其服务注册、日志管理、容器资源动态变更三个方面进行改进，同时提供客户端接口，在支持长服务的同时简化服务的运行时管理难度。

3）结合微服务架构以及服务治理框架Dubbox，实现服务注册、发现、监控、调用统计、容量评估等功能，并通过监控获得的服务状态与资源管理平台进行集成，对服务进行自动水平扩展。从而实现对应用服务的综合管理。

## 下一步工作

本课题后续研究工作包括以下几个方面：

1）研究资源调度算法，进一步提升系统资源利用率。

2）进一步研究邻里社交模型，使其适用范围更广，更易于扩展。

3）为开发人员订制社区应用服务末班，使应用开发人员能够更快速高效的进行应用整合开发。

参考文献

1. Vavilapalli, Vinod Kumar, et al. "Apache hadoop yarn: Yet another resource negotiator." Proceedings of the 4th annual Symposium on Cloud Computing. ACM, 2013.
2. Hindman, Benjamin, et al. "Mesos: A Platform for Fine-Grained Resource Sharing in the Data Center." NSDI. Vol. 11. 2011
3. Schwarzkopf, Malte, et al. "Omega: flexible, scalable schedulers for large compute clusters." Proceedings of the 8th ACM European Conference on Computer Systems. ACM, 2013.
4. Verma, Abhishek, et al. "Large-scale cluster management at Google with Borg." Proceedings of the Tenth European Conference on Computer Systems. ACM, 2015.
5. Corona facebook https://github.com/facebookarchive/hadoop-20/tree/master/src/contrib/corona.
6. Sharma P P, Navdeti C P. Securing Big Data Hadoop: A Review of Security Issues, Threats and Solution[J]. IJCSIT) International Journal of Computer Science and Information Technologies, 2014, 5(2).
7. Hunt, Patrick, et al. "ZooKeeper: Wait-free Coordination for Internet-scale Systems." USENIX Annual Technical Conference. Vol. 8. 2010.
8. RFC1123: http://www.ietf.org/rfc/rfc1123.txt
9. George, Lars. HBase: the definitive guide. " O'Reilly Media, Inc.", 2011.
10. Toshniwal, Ankit, et al. "Storm@ twitter." Proceedings of the 2014 ACM SIGMOD international conference on Management of data. ACM, 2014.
11. Zaharia, Matei, et al. "Spark: cluster computing with working sets." Proceedings of the 2nd USENIX conference on Hot topics in cloud computing. Vol. 10. 2010.
12. Zaharia, Matei, et al. "Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing." Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation. USENIX Association, 2012.
13. Apache Slider: <http://slider.incubator.apache.org/index.html>.
14. Ousterhout K, Wendell P, Zaharia M, et al. Sparrow: distributed, low latency scheduling[C]//Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles. ACM, 2013: 69-84.
15. Hitesh Ballani , Paolo Costa , Thomas Karagiannis , Ant Rowstron, Towards predictable datacenter networks, ACM SIGCOMM Computer Communication Review, v.41 n.4, August 2011
16. N. Capit , G. Da Costa , Y. Georgiou , G. Huard , C. Martin , G. Mounie , P. Neyron , O. Richard, A batch scheduler with high level components, Proceedings of the Fifth IEEE International Symposium on Cluster Computing and the Grid (CCGrid'05) - Volume 2, p.776-783, May 09-12, 2005
17. Byung-Gon Chun , Tyson Condie , Carlo Curino , Chris Douglas , Sergiy Matusevych , Brandon Myers , Shravan Narayanamurthy , Raghu Ramakrishnan , Sriram Rao , Josh Rosen , Russell Sears , Markus Weimer, REEF: retainable evaluator execution framework, Proceedings of the VLDB Endowment, v.6 n.12, p.1370-1373, August 2013
18. Jeffrey Dean , Sanjay Ghemawat, MapReduce: simplified data processing on large clusters, Communications of the ACM, v.51 n.1, January 2008
19. Wesley Emeneker , Dave Jackson , Joshua Butikofer , Dan Stanzione, Dynamic virtual clustering with xen and moab, Proceedings of the 2006 international conference on Frontiers of High Performance Computing and Networking, December 04-07, 2006, Sorrento, Italy
20. Michael Isard , Mihai Budiu , Yuan Yu , Andrew Birrell , Dennis Fetterly, Dryad: distributed data-parallel programs from sequential building blocks, Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007, March 21-23, 2007, Lisbon, Portugal
21. Mohammad Islam , Angelo K. Huang , Mohamed Battisha , Michelle Chiang , Santhosh Srinivasan , Craig Peters , Andreas Neumann , Alejandro Abdelnur, Oozie: towards a scalable workflow management system for Hadoop, Proceedings of the 1st ACM SIGMOD Workshop on Scalable Workflow Execution Engines and Technologies, p.1-10, May 20-20, 2012, Scottsdale, Arizona
22. David B. Jackson , Quinn Snell , Mark J. Clement, Core Algorithms of the Maui Scheduler, Revised Papers from the 7th International Workshop on Job Scheduling Strategies for Parallel Processing, p.87-102, June 16, 2001
23. Grzegorz Malewicz , Matthew H. Austern , Aart J.C Bik , James C. Dehnert , Ilan Horn , Naty Leiser , Grzegorz Czajkowski, Pregel: a system for large-scale graph processing, Proceedings of the 2010 ACM SIGMOD International Conference on Management of data, June 06-10, 2010, Indianapolis, Indiana, USA
24. Christopher Olston , Benjamin Reed , Utkarsh Srivastava , Ravi Kumar , Andrew Tomkins, Pig latin: a not-so-foreign language for data processing, Proceedings of the 2008 ACM SIGMOD international conference on Management of data, June 09-12, 2008, Vancouver, Canada
25. Konstantin Shvachko , Hairong Kuang , Sanjay Radia , Robert Chansler, The Hadoop Distributed File System, Proceedings of the 2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST), p.1-10, May 03-07, 2010
26. Douglas Thain , Todd Tannenbaum , Miron Livny, Distributed computing in practice: the Condor experience: Research Articles, Concurrency and Computation: Practice & Experience, v.17 n.2-4, p.323-356, February 2005
27. Yuan Yu, Michael Isard , Dennis Fetterly , Mihai Budiu , Úlfar Erlingsson , Pradeep Kumar Gunda , Jon Currey, DryadLINQ: a system for general-purpose distributed data-parallel computing using a high-level language, Proceedings of the 8th USENIX conference on Operating systems design and implementation, p.1-14, December 08-10, 2008, San Diego, California
28. D. R. Engler, M. F. Kaashoek, and J. O’Toole. Exokernel: An operating system architecture for application-level resource management. In SOSP, pages 251–266, 1995.
29. A. Ghodsi, M. Zaharia, B. Hindman, A. Konwinski, S. Shenker,
30. and I. Stoica. Dominant resource fairness: fair allocation of multiple resource types. In NSDI, 2011.
31. M. Isard, V. Prabhakaran, J. Currey, U. Wieder, K. Talwar, and A. Goldberg. Quincy: Fair scheduling for distributed computing clusters. In SOSP, November 2009.
32. S. Y. Ko, I. Hoque, B. Cho, and I. Gupta. On availability of intermediate data in cloud computations. In HOTOS, May 2009.
33. J. N. Matthews, W. Hu, M. Hapuarachchi, T. Deshane, D. Dimatos, G. Hamilton, M. McCabe, and J. Owens. Quantifying the performance isolation properties of virtualization systems. In ExpCS ’07, 2007.
34. D. G. Murray, M. Schwarzkopf, C. Smowton, S. Smith, A. Madhavapeddy, and S. Hand. Ciel: a universal execution engine for distributed data-flow computing. In NSDI, 2011.
35. D. Nurmi, R. Wolski, C. Grzegorczyk, G. Obertelli, S. Soman, L. Youseff, and D. Zagorodnov. The Eucalyptus open-source cloud-computing system. In CCA ’08, 2008.
36. R. Raman, M. Livny, and M. Solomon. Matchmaking: An extensible framework for distributed resource management. Cluster Computing, 2:129–138, April 1999.
37. C. A. Waldspurger and W. E. Weihl. Lottery scheduling: flexible proportional-share resource management. In OSDI, 1994.
38. Y. Yu, P. K. Gunda, and M. Isard. Distributed aggregation for data-parallel computing: interfaces and implementations. In SOSP ’09. pages 247-260, 2009.
39. M. Zaharia, D. Borthakur, J. Sen Sarma, K. Elmeleegy, S. Shenker, and I. Stoica. Delay scheduling: A simple technique for achieving locality and fairness in cluster scheduling. In EuroSys 10, 2010.
40. Namiot D, Sneps-Sneppe M. On micro-services architecture[J]. International Journal of Open Information Technologies, 2014, 2(9): 24-27.
41. Gütl C, García-Barrios V M. Towards an Advanced Modeling System applying a Service-based Approach[C]//Advanced Learning Technologies, 2005. ICALT 2005. Fifth IEEE International Conference on. IEEE, 2005: 860-862.
42. Day M, Rosenberg J, Sugano H. A model for presence and instant messaging[R]. 2000.
43. Campbell B, Rosenberg J, Schulzrinne H, et al. Session initiation protocol (SIP) extension for instant messaging[J]. 2002.
44. Saint-Andre P. Extensible messaging and presence protocol (XMPP): Core[J]. 2011.

# 攻读学位期间发表的学术论文

1. First Author. Accommodate Apache Yarn to Long-lived Services. The 2015 International Conference on Computer Science and Network Technology, December 20-21, 2015, Heilongjiang.