西安工业大学
XI'AN TECHNOLOGICAL UNIVERSITY

本科毕业设计(论文)

题目：社区食堂系统的设计与实现

院 （系）：计算机科学与工程学院

专　　业：软件工程

班　　级：20060213

学　　生：张宇涛

学　　号：2020032951

指导教师：陈芳

2024 年 6 月

西安工业大学
XI'AN TECHNOLOGICAL UNIVERSITY

本科毕业设计(论文)

题目：社区食堂系统的设计与实现

院 （系）：计算机科学与工程学院

专　　业：软件工程

班　　级：20060213

学　　生：张宇涛

学　　号：2020032951

指导教师：陈芳

2024 年 6 月

社区食堂系统的设计与实现

摘要

随着我国逐渐扩大城市化发展，城市内已经出现越来越多的居民社区。每个社区内，居民的年龄阶段参差不齐，不同人群对不同营养的需求也大相径庭。随着类似美团、饿了么等在线外卖点餐软件的出现，社区内也急需一款更符合当地饮食习惯和商家需求的社区食堂点餐系统。

本研究旨在设计并实现一款社区食堂系统以提供更符合当地社区居民饮食文化的食堂系统软件。该系统基于 Qt 平台开发，使用 Mysql 数据库、信号槽和事件处理技术完成后端程序开发。同时，使用 Qt Designer 和 QSS 技术实现对软件前端的设计和渲染，从而实现软件 MVC 的程序架构。本研究首先对社区食堂系统的后端和前端功能需求进行全面详细的分析，对系统开发产生有效的指导性意见。接着，设计数据库表结构和 SQL 语句，持久化存储用户的账号和购买等信息，同时使用 Designer 组合多种控件，并结合信号槽和事件处理技术实现符合要求的控件交互功能。最后，采用 QSS 技术优化前端界面，实现系统的可视化展示和用户的交互。

本研究设计并实现了一款基于 Qt 平台的社区食堂系统，通过系统的实际应用和测试，本项目取得了显著的成果。社区食堂系统在 MVC 技术的支持下，能够对用户的操作做出精准的应答，具有良好的用户体验。同时，通过 QSS 界面优化技术，系统能够直观地呈现给用户，并得到了用户的积极反馈。该系统为社区居民提供了高效、便捷的餐饮获取方式，丰富了社区智能化建设的内涵，具有良好的推广和应用前景。

关键词：Qt；Mysql 数据库；SQL 语句；Qt Designer；QSS 技术；MVC 框架；信号槽；事件处理技术

Design And Implementation Of Community Canteen System

Abstract

As my country gradually expands urbanization, more and more residential communities have emerged in cities. In each community, the age of residents varies, and different groups have very different needs for different nutrition. With the emergence of online takeaway ordering software such as Meituan and Ele.me, the community is in urgent need of a community canteen ordering system that is more in line with local eating habits and business needs.

This study aims to design and implement a community canteen system to provide a canteen system software that is more in line with the local community residents' dietary culture. The system is developed based on the Qt platform, and the back-end program development is completed using Mysql database, signal slots and event processing technology. At the same time, Qt Designer and QSS technology are used to implement the design and rendering of the software front end, thereby realizing the software MVC program architecture. This study first conducts a comprehensive and detailed analysis of the back-end and front-end functional requirements of the community canteen system, and produces effective guiding opinions on system development. Then, the database table structure and SQL statements are designed to persistently store user accounts and purchase information. At the same time, Designer is used to combine multiple controls, and signal slots and event processing technology are combined to achieve the required control interaction functions. Finally, QSS technology is used to optimize the front-end interface to achieve system visualization and user interaction.

This study designed and implemented a community canteen system based on the Qt platform. Through the actual application and testing of the system, this project has achieved remarkable results. With the support of MVC technology, the community canteen system can accurately respond to user operations and has a good user experience. At the same time, through the QSS interface optimization technology, the system can be presented to users intuitively and has received positive feedback from users. The system provides community residents with an efficient and convenient way to obtain food and beverages, enriches the connotation of community intelligent construction, and has good prospects for promotion and application.

Keywords: Qt; Mysql database; SQL statement; Qt Designer; QSS technology; MVC framework; signal slot; event processing technology

目录

摘要 I

Abstract II

1 绪论 1

1.1 研究背景 1

1.2 研究目的及意义 1

1.3 国内外研究现状 2

1.4 论文主要工作安排 2

2 技术介绍 4

2.1 数据库的概念及应用 4

2.1.1 引言 4

2.1.2 数据库的定义与特点 4

2.1.3 数据库管理系统 DBMS 5

2.2 Qt 技术的概念及应用 6

2.2.1 引言 6

2.2.2 Qt 技术概述 6

2.2.3 Qt Designer 和 QSS 7

2.2.4 信号槽技术 7

2.2.5 事件处理技术 8

2.2.6 MVC 技术 10

2.3 本章小结 11

3 系统需求分析 12

3.1 引言 12

3.2 功能需求 12

3.3 非功能需求 13

3.4 用户需求 14

3.5 安全需求 15

3.6 可靠性需求 16

3.7 性能需求 17

3.8 本章小结 18

4 数据库设计 19
4.1 引言 19
4.2 数据库的设计与规范化 19
4.3 社区食堂系统数据库信息设计 20
4.3.1 用户登录注册信息表 20
4.3.2 用户头像表 21
4.3.3 用户购物车表 22
4.3.4 用户订单表 22
4.3.5 系统公告表 23
4.3.6 菜品详情表 24
4.4 系统 E-R 图 24
4.5 本章小结 25
5 系统功能介绍及实现 26
5.1 引言 26
5.2 社区食堂系统功能流程介绍 26
5.3 系统功能模块设计介绍 27
5.3.1 登录功能介绍 27
5.3.2 注册功能介绍 29
5.3.3 用户头像功能介绍 30
5.3.4 公告轮播图功能介绍 32
5.3.5 菜品浏览功能介绍 33
5.3.6 菜品展示器功能介绍 34
5.3.7 购物车功能介绍 35
5.3.8 订单详情功能介绍 37
5.3.9 用户管理功能介绍 39
5.4 系统功能实现 41
5.4.1 登录功能实现 41
5.4.2 注册功能实现 46
5.4.3 用户头像功能实现 51
5.4.4 公告轮播图功能实现 52
5.4.5 菜品浏览功能实现 55
5.4.6 购物车功能实现 60
5.4.7 订单详情功能实现 69
5.4.8 用户管理功能实现 74

5.5 本章小结 83
6 系统测试 84
6.1 引言 84
6.2 测试内容 84
6.3 测试环境 84
6.4 功能性测试 84
6.4.1 登录功能 84
6.4.2 注册功能 85
6.4.3 用户头像功能 85
6.4.4 用户菜品浏览功能 85
6.4.5 用户公告浏览功能 85
6.4.6 购物车功能 85
6.4.7 立即购买功能 85
6.4.8 订单详情功能 86
6.4.9 管理员用户信息管理功能 86
6.4.10 管理员菜品管理功能 86
6.4.11 管理员系统给公告管理功能 86
6.5 非功能测试 86
6.5.1 安全性测试 86
6.5.2 可靠性测试 87
6.5.3 性能测试 87
6.5.4 可用性测试 87
6.6 本章小结 87
7 总结与展望 88
7.1 总结 88
7.2 展望 88
参考文献 89
致谢 91
毕业设计（论文）知识产权声明 92
毕业设计（论文）独创性声明 93
附录 94

1 绪论

1.1 研究背景

近年来，随着我国逐渐扩大城市化发展，城市内已经出现越来越多的居民社区。社区内人员点餐就餐成为困扰每个居民的社会问题。在这种背景下，一个基于 Qt 开发的社区食堂系统可以成为社区居民和饮食生活文化中不可或缺的工具，有助于提高社区信息服务的智能化水平。

基于 Qt 的社区食堂系统的设计与实现，是一个实用的、优化人民日常生活的研究课题。随着城市内人口的快速增长和人口老龄化问题的日益加剧，不同功能的主题社区也在城市内逐渐孕育而生，每个社区内，居民的年龄阶段参差不齐，不同人群对不同营养的需求也大相径庭，并且，其中部分人群存在无法自行烹饪的情况。零散化的餐厅餐饮，会导致食品安全、餐饮服务无法统一管理，增加居民生活负担的情况发生。因此，设计并实现一款能够符合当地社区居民饮食习惯的社区食堂系统，将大大提高居民的生活品质和幸福指数。

当前随着互联网技术的迅猛发展，诞生了众多软件开发技术，如 MVC 框架、数据库技术、Qt 前后端开发平台等方面的技术已经取得巨大进展。这些技术的发展为社区食堂系统的设计与实现提供了有力的技术支撑。通过利用信号槽和事件处理技术，可以使系统产生更高效和更符合用户要求功能，并以 MVC 架构的形式准确应答用户的请求。

此外，当前点餐系统迅猛发展，产生了如美团、饿了么等在线点餐平台，为构建更加本土化、亲民的点餐系统提供了更多的参考价值。应用数据库、信号槽和事件处理技术，可以使系统不断完善自身的功能，更好地适应用户的需求。

1.2 研究目的及意义

社区食堂系统在于解决社区居民用餐方面的问题，特别是针对那些有特殊营养需求或者无法自行烹饪的人群。社区食堂系统提供了一个集中的用餐场所，为居民提供健康、营养均衡的餐饮服务。社区食堂系统的产生能够在一定程度上改善社区内部居民的饮食结构、提高生活质量，同时社区食堂内居民之间每日相互接触也有助于促进居民互助和交流。在一些社区中，社区食堂系统也可以成为一种社区活动的场所，促进社区建设和凝聚力方面具有重大意义。

1.3 国内外研究现状

Dierinck P 对精神病保健社区中精神病专业人士、临床医生、社会工作者、护理人员以及照顾有复杂护理需求的精神病患者的部分心理健康专业人士进行了深入研究。借助社区系统内的补偿机制，通过临床案例材料、人类学和现象学方法以及社区护理的个人经验，Peter Dierinck 提出了庇护住房的新模式。[1]

刘佳影在《基于 MINA 框架的社区食堂智慧餐饮系统》中设计了一个基于微信平台的社区食堂智慧餐饮系统。社区食堂作为全世界居民的摇篮而备受关注。然而当今社区食堂内部依然存在着诸多挑战，例如浪费食物、无序就餐等行为严重影响着社区内部的居民和谐。本研究中基于这些问题提出了相应具有指导性的意见，对完善国内社区系统建设具有重大意义。[2]

崔娜，宋珂欣在《智慧社区管理系统建设概述》中以山东省某小区为例，结合其建设要求及建设内容，设计了一系列智慧化建设方案。在该智慧社区管理系统的建设上，对于社区内提升管理效率、提高居民的幸福指数都具有重要意义。本研究中还对社区管理系统存在的关键性问题进行了分析，并提出建设性意见，指明了智慧社区系统建设未来的研究方向。[3]

程嘉萱，邝慧仪在《智慧社区垃圾分类系统设计》中对社区内垃圾分类和回收的具体内容进行了详细的研究。本研究中应用了物联网、人工智能以及 VR 技术。社区内居民可以通过语音开箱进行垃圾投放，其智能系统将通过人工智能技术识别垃圾，并通过传送带传送至预处理装置压缩体积，最后机械臂将其分拣至指定垃圾桶。管理员可通过智能终端进行智慧管理，安全且卫生。该研究对社区智慧化社区中的垃圾分类问题提出了有意义的建设性意见。[4]

通过对国内外文献的查阅和阅读我们可以了解到，国内外智慧化社区系统都有得到不同程度的智慧化发展。然而国内外对本地社区食堂的饮食文化的设计和管理维护上还存在一定的问题。本文为实现社区管理系统，以数据库 Mysql 和前后端开发技术 Qt 作为开发工具入手，将更深程度的设计和使用，来为构建一个更加完善的社区食堂系统提供一些实际参考性意见。

1.4 论文主要工作安排

本文主要的研究项目是基于 Qt 平台的社区食堂系统的设计与实现，通过结合 Qt Designer、MVC 技术、SQL 技术、数据库连接技术、信号槽和事件处理等技术，来实现社区内居民点餐和食堂内就餐功能。

社区食堂系统的设计与实现毕业设计论文大体框架及思路如下：

第一章：绪论。简要介绍该论文的产生背景和意义。

第二章：技术介绍。对数据库基础、Qt 和 MVC 框架进行简要介绍，

第三章：系统需求分析。对社区食堂系统的功能性和非功能性需求进行分析。

第四章：数据库设计。深入探讨本系统采用的数据库结构的合理性。

第五章：系统功能介绍及实现。对社区食堂系统的功能流程和实现深入探讨。

第六章：系统测试。本章将对社区食堂系统的功能进行全面验证。

第七章：总结与展望。总结社区食堂系统的优势与不足，并对该研究的后续改进工作进行展望。

2 技术介绍

2.1 数据库的概念及应用

2.1.1 引言

当今时代是一个数据大爆发的时代。这些数据涵盖各种形式的信息，包括文本、图片、音频和视频等，涉及各个领域和行业。数据的快速增长给传统的数据管理、存储和处理带来了前所未有的挑战，而数据库技术应运而生，成为有效管理和利用数据的关键工具。信息时代的核心转变在于数据的重要性逐渐凸显，成为信息社会的基石。大数据概念的提出和普及，使得对海量数据的高效管理和分析成为迫切需求。数据库作为数据管理和存储的主要工具，为信息时代的数据处理提供了可靠和高效的支持。它不仅能存储海量数据，还能提供快速的数据检索和处理功能，为各行业提供了强大的信息化保障。信息技术的快速发展和不断创新，包括云计算、人工智能和物联网等技术的普及和应用，进一步推动了信息时代的发展。这些技术为数据库的发展和应用提供了更广阔的空间和可能性。数据库在云计算环境下的部署和应用，以及与人工智能和物联网的结合，使得数据库在信息时代中扮演着更为重要的角色，支撑着各种信息化技术的发展和应用。

在本研究中，计划使用 Mysql 数据库来开发和设计社区食堂系统的功能。Mysql 数据库本质上是 OLTP（Online Transaction Processing 在线事务处理）数据库中的一种。MySQL 是一种关系型数据库，采用表格来储存数据，并使用 SQL 语言进行快速查询。具有高性能的特点，支持多种存储引擎，适用于不同的应用场景，并提供了丰富的功能和工具。因此，本研究选择采用 Mysql 来构建社区食堂相关的数据库。

2.1.2 数据库的定义与特点

数据库是指按照一定数据模型组织、存储和管理数据的集合，具有持久化存储数据、方便数据检索和更新的特性。数据库系统是由数据库、数据库管理系统（DBMS）和相关应用程序组成的集成环境，用于对数据进行有效管理和处理。

数据库中的数据以表格形式结构化存储，便于组织和管理。数据库实现了数据与应用程序的独立性，使得应用程序能够独立于数据存储和结构的变化。多个用户可以同时访问数据库，并实现数据共享与协作，提高工作效率。数据库保证数据的一致性，通过事务管理和约束条件确保数据的完整性和准确性。数据库中的数据是持久存储的，不会因系统故障或断电而丢失，保证数据的安全性。数据库支持多用户同时对数据进行访问和操作，提供了数据并发处理和控制机制。数据库具有数据备份、日志记录和恢复机制，确保在数据丢失或损坏时能够快速进行恢复。数据库从类别方面还分为，关系型数据库与非关系型（key-value）数据库。

在本研究中，采用关系型数据库设计社区食堂系统的数据库关系。关系型数据库能够进行复杂的数据查询、连接和聚合操作，提供了丰富的数据分析和提取功能。通过事务管理和约束条件，关系型数据库确保数据的一致性，遵循 ACID（Atomicity 原子性、Consistency 一致性、Isolation 隔离性、Durability 持久性）原则，保证了数据的完整性。关系型数据库以表格形式组织数据，逻辑清晰，易于理解和管理，适合大多数企业应用的数据存储需求，同样也适用于应用在本研究中。

2.1.3 数据库管理系统 DBMS

数据库管理系统 DBMS（Database Management System）是一种软件工具，用于管理和组织数据库中的数据，其大致框架模式，如图 2.1 所示。它提供了一系列功能，包括数据存储、检索、更新、备份和安全性管理等，使用户能够有效地管理数据库。

DBMS 提供数据建模的能力，通过表格（表）的形式组织和存储数据，包括定义数据的结构、关系和约束条件等。支持使用查询语言（如 SQL）进行数据检索、更新和操作，使用户能够方便地访问和处理数据库中的数据。提供了数据安全性和完整性的管理机制，包括用户权限控制、数据加密、事务管理等功能，保障数据的安全和一致性。支持多用户并发访问和操作数据库，并提供并发控制机制，确保数据的一致性和隔离性。提供数据备份和恢复的功能，保障数据在意外丢失或损坏时的可靠性。提供了性能优化的功能和工具，包括索引、查询优化器、缓存管理等，以提高数据库的访问速度和效率。

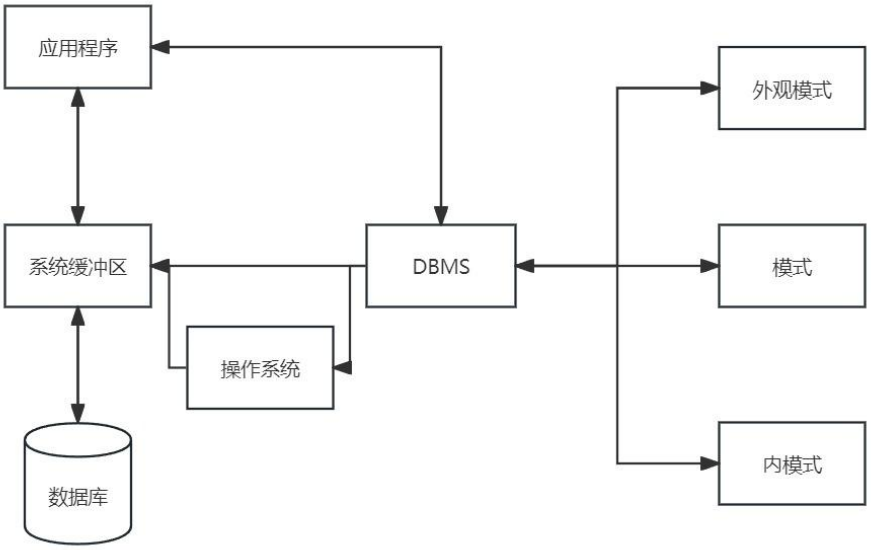

图 2.1DBMS 框架

2.2 Qt 技术的概念及应用

2.2.1 引言

近年来，随着跨平台应用程序开发需求的增长和不断变化的技术环境，Qt 技术作为一种强大的跨平台的 C++应用程序开发框架，受到了广泛关注。Qt 技术的出现填补了传统 C++开发框架的不足，为开发者提供了一种灵活、高效的跨平台开发决方案。Qt 中通过自身的 Qt Designer 设计界面直观的将常用控件的功能样式提供给开发者，并且结合 QSS 的样式表可以设计各种不同样式的控件，使得前端开发更具目标性和导向性。Qt 中的信号槽和事件处理技术的组合应用，可以实现大部分的控件交互功能，大大提升后端程序的可开发性和可扩展性。开发者可以通过自定义的控件封装，实现自己想要实现的控件的功能形式和控件样式，大大提高了 Qt 在前后端设计开发的能力和可操作性。

在本节中，我们将介绍 Qt 技术的定义和特点，探讨其作为跨平台开发框架的优势和重要性。随后，我们将重点讨论 Qt 框架提供的核心功能，并探讨其在实际应用中的广泛作用。最后，介绍在本项目中结合 Qt 技术和 Mysql 数据库实现的社区食堂系统的 MVC 程序架构。

2.2.2 Qt 技术概述

Qt 技术是一种跨平台的 C++应用程序开发框架，旨在简化开发人员在不同平台上构建用户界面和应用程序的过程。Qt 提供了丰富的工具和功能，使开发者

能够高效地创建各种类型的应用程序。

Qt 技术的特点包括跨平台性、模块化设计、用户界面设计工具、强大的图形引擎和许多预构建的类库。这些特点使得 Qt 成为一种灵活且功能强大的开发框架。使用 Qt 进行开发具有明显的跨平台优势，开发人员可以编写一次代码，然后在不同操作系统上部署运行，包括 Windows、macOS、Linux 等，节省了开发和维护成本。Qt 拥有庞大的生态系统，支持多个操作系统和设备类型，积极的开发者社区和丰富的资源库，为开发者提供了强大的支持和资源。Qt 技术广泛应用于各种领域，如移动应用开发、嵌入式系统、桌面应用程序、图形化用户界面设计等。其灵活性和可定制性使之成为各种应用程序开发的理想选择。

在本项目中，主要使用 Qt 平台开发视图和控制的程序开发，数据库实现模块的用户数据持久化处理任务，从而实现社区食堂系统的展示、逻辑和用户数据管理的功能。

2.2.3 Qt Designer 和 QSS

Qt Designer 是 Qt 框架中的一款可视化界面设计工具，旨在帮助开发者快速设计、布局和定制用户界面。它提供了丰富的控件库、布局管理器和可视化编辑功能，使用户可以通过拖拽、放置和调整来设计界面，而无需编写大量的代码。

Qt Designer 具有直观的用户界面，可以通过拖放控件、设置属性、定义信号槽等方式快速创建用户界面。它还支持自定义控件的设计器插件和国际化功能，帮助开发者更高效地设计界面。生成的设计文件可以直接由 Qt Creator 或其他 Qt 应用程序加载和使用。Qt Designer 的使用可以大大简化用户界面的设计过程，提高开发效率，降低出错几率，同时也方便设计和开发团队之间的协作。通过可视化设计，开发者可以更容易地预览和调整界面效果，快速响应需求变化。

QSS （Qt Style Sheets）是 Qt 提供的一种样式表技术，类似于 CSS，用于定制和美化 Qt 程序的外观和布局。借助 QSS，开发者可以通过简单的样式表语法定义控件的外观、字体、颜色、边框等，实现界面风格的定制和统一。

QSS 允许开发者全局或针对特定控定制样式，实现界面的定制化风格。通过选择器、属性设置和样式规则，可以轻松地实现按钮、标签、窗口等控件的美化和风格定制，实现视觉效果的个性化和统一。

Qt Designer 结合 QSS 可以实现实时预览样式效果，使开发者可以快速调整样式表并实时查看效果，帮助其更加高效和直观地定制和优化应用程序界面。

2.2.4 信号槽技术

信号槽是 Qt 框架中一种重要的通信机制，用于对象之间的事件通知和数据传递。通过信号槽机制，一个对象（发送者）可以发出信号（Signals），而另一个对象（接收者）能够捕获这个信号并执行相应的槽函数（Slots），实现对象之

间的交互和通信，如图 2.2 所示。

每个 QObject 派生类都可以定义信号和槽。信号是一种特殊的成员函数，用于发出通知，而槽则是普通的成员函数，用于响应信号。当产生信号的控件需要发送信号时，与该信号相连的另一个接收控件的槽函数将被调用执行指定的行为。

在 Qt 中，通过 connect()函数将一个信号和一个槽连接起来。这样，在信号发出时，与之连接的槽函数会被自动调用。连接可以是一对一、一对多、多对一甚至多对多的关系，灵活适应不同场景的需求。

信号槽机制支持槽函数中的参数传递。当信号被发出时，可以将参数传递给与之连接的槽函数，实现数据的交互和传递。这使得对象之间的通信更加灵活和高效。

信号槽机制在多线程环境下具有很好的线程安全性。Qt 使用了线程安全的事件队列来管理信号槽的触发和执行，确保在不同线程间的通信和操作安全可靠。

信号槽技术在 Qt 开发中被广泛应用，特别适合处理用户界面事件、数据传递和对象之间的交互。它简化了软件开发过程，提高了代码的可读性和可维护性。

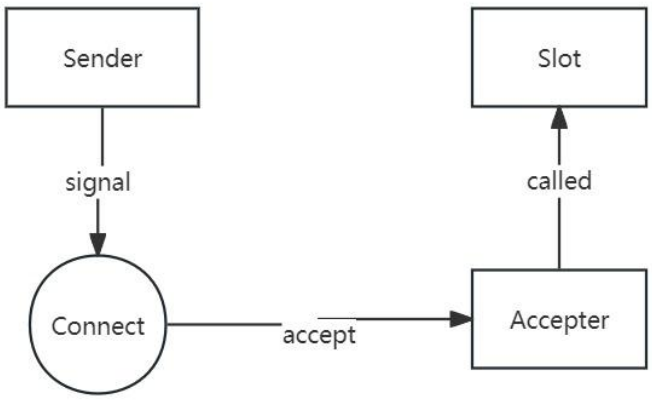

图 2.2 信号槽链接过程

2.2.5 事件处理技术

Qt 使用事件模型来处理用户交互和操作系统事件。每个用户操作或操作系统事件都可以被视为一个事件，并且 Qt 提供了一套灵活的机制来捕获和处理这些事件，如图 2.3 所示。

事件过滤器允许开发者对特定类型的事件进行拦截和处理。通过安装事件过滤器，可以在事件发送给对象之前对其进行拦截，检查和修改。这为开发者提供了额外的控制和定制能力。

在 Qt 中，每个 QObject 派生类都可以重写事件处理器函数来处理特定类型的事件。例如，重写 event 函数来处理各种事件，或者重写 keyPressEvent、mousePressEvent 等函数来处理特定的键盘或鼠标事件。

Qt 支持自定义事件，开发者可以通过继承 QEvent 类创建自定义事件，并在相关的对象中处理这些事件。这为开发者提供了一种扩展 Qt 事件模型的能力，以满足特定的应用需求。

定时器是一种特殊类型的事件处理技术，在 Qt 中被广泛使用。通过定时器，开发者可以安排和处理来自操作系统的定时事件，或者自定义的定时任务，用于实现定时刷新、动画效果、后台任务等功能。

Qt 提供了多线程事件处理的支持，允许在不同线程中处理事件。这使得开发者能够更好地管理和处理多线程环境下的用户交互、并发操作和系统事件。

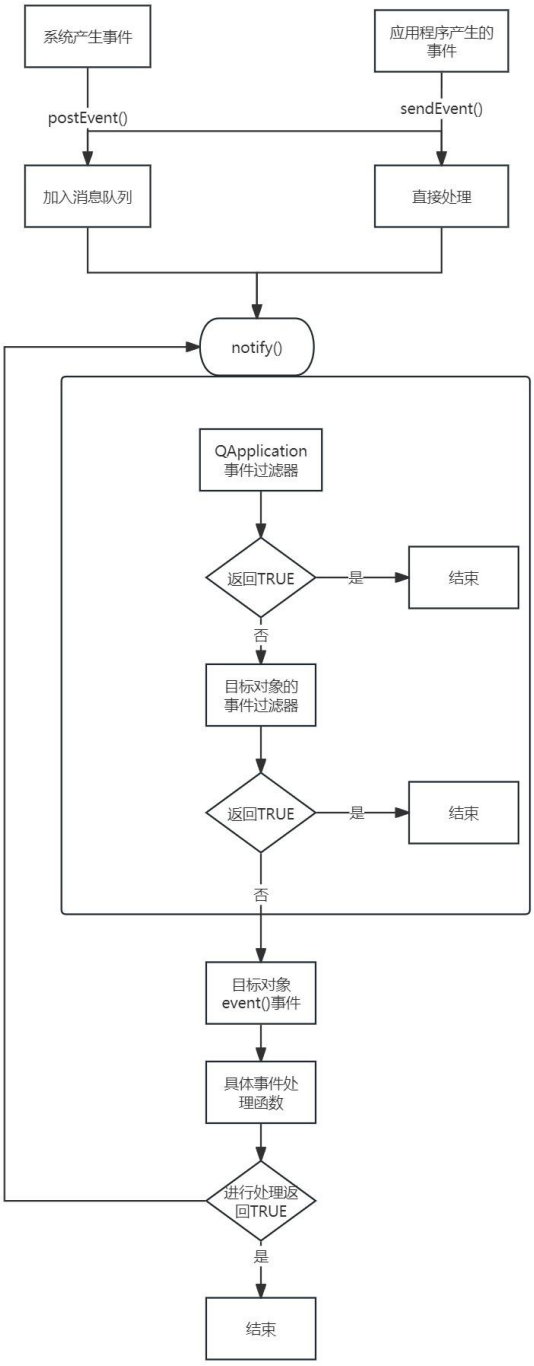

图 2.3 事件处理过程

2.2.6 MVC 技术

MVC（Model-View-Controller）是一种软件架构模式，用于将应用程序的用户界面、数据和操作分离，以便实现更好的代码组织和模块化，如图 2.4 所示。

模型（Model）是应用程序的数据层，负责管理数据的获取、存储、处理和业务逻辑。模型不依赖视图或控制器，因此可以独立于用户界面进行测试和维护。在 Qt 中，模型可以是数据结构、数据库连接、数据操作接口等，通过模型可以实现数据的封装和管理。

视图（View）是用户界面的表示层，负责显示数据和与用户进行交互。视图不直接处理数据，它从模型中获取数据并将其呈现给用户。在 Qt 中，视图可以是窗口、对话框、控件等，通过视图实现用户界面的布局和展示。

控制器（Controller）是用户界面和应用程序逻辑之间的中介，负责接收用户的输入并根据输入更新模型或视图。它将用户操作转换为对模型和视图的操作。在 Qt 中，可以通过信号和槽机制来实现控制器的逻辑处理。

MVC 模式中，模型、视图和控制器之间的通信是通过严格定义的接口和规则进行的。通过这种机制，实现了各部分之间的解耦，增加了模块的可维护性和扩展性。MVC 技术在 Qt 中被广泛应用于图形用户界面（Graphical User Interface）应用程序的开发中，通过将业务逻辑、数据和用户界面分离，实现了代码的重用和可维护性。

MVC 模式使应用程序代码更易于管理和测试，可扩展性更强。通过 MVC 框架的应用不同部分的职责明确，使得开发者能够更专注于各自部分的开发和优化。

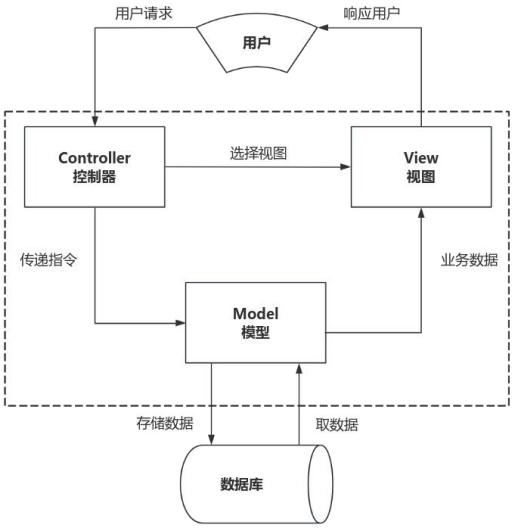

图 2.4MVC 框架

2.3 本章小结

本章深入探讨了数据库和 Qt 技术的概念和实际应用。数据库是一种组织数据的结构化方式，而 DBMS 则提供数据管理的工具和功能。这些概念和技术对于数据库应用的合理运用和管理至关重要，为数据驱动的应用提供了可靠的基础。在本章中，深入探讨了 QT 技术的核心内容。介绍了 Qt 技术的定义和特点，探讨了其作为跨平台开发框架的优势和重要性。详细描述了 Qt Designer 作为可视化界面设计工具的功能特点以及 QSS 作为样式表技术的应用和优势。介绍了在 Qt 中控制各控件功能流水的信号槽和事件处理技术。展示了在本研究项目中使用到的 MVC 技术的具体内容，包括模型、视图、控制器之间的关系和通信机制，以及在 Qt 中的应用场景和优势。

3 系统需求分析

3.1 引言

社区食堂作为社区内重要的饮食服务设施，在满足居民日常饮食需求的同时，也承担着社区互动和公共服务的功能。随着社区食堂管理的信息化需求日益增长，构建一套高效、便捷的社区食堂系统成为当前急需解决的问题。

本章旨在进行社区食堂系统的需求分析，以期能够准确把握社区食堂系统所需满足的各项功能性和非功能性需求，为系统的设计和开发提供有力支撑。从功能需求、非功能需求、用户需求、安全需求、可靠性需求、性能需求等多个方面进行全面的阐述和讨论。通过本章的需求分析，旨在为社区食堂系统的设计和实施提供有力的依据和指导，为社区食堂服务的提升和现代化提供有效的支持。

3.2 功能需求

社区食堂系统作为社区重要的饮食服务设施，旨在为社区居民提供便捷的用餐服务和社交互动空间。社区食堂系统面向社区居民及相关工作人员，提供餐饮预订、配餐管理、结算等服务内容。该功能需求主要包括以下几个方面：

（1）用户身份识别：用户包括社区居民和相关工作人员，他们需通过系统进行身份识别和注册，以便享受餐饮服务。

（2）菜品浏览与订餐：用户可通过系统浏览菜品信息、了解价格及供应时间，并进行预订或预先点餐。

（3）配餐管理：系统需支持对预订菜品进行配餐管理，确保菜品按时准确地送达给用户。

（4）结算与支付：用户通过系统完成用餐后的结算，同时系统需支持现金和在线支付等多种支付方式。

（5）用户反馈与评价：用户可通过系统对餐品进行评价和反馈，提供意见和建议，以便完善菜品及服务质量。

（6）定期营养菜谱推送：系统应支持定期发布社区营养菜谱，并向用户推送，促进健康饮食意识。

（7）订单管理：管理用户订单，包括查看订单状态、历史订单、订单统计等功能。

（8）权限管理：确保系统权限安全，包括管理员权限和普通用户权限的管理

以上功能需求内容涵盖了社区食堂系统建设中的重要功能模块，可为系统开发和设计提供具体的需求细则。详见图 3.1 所示。

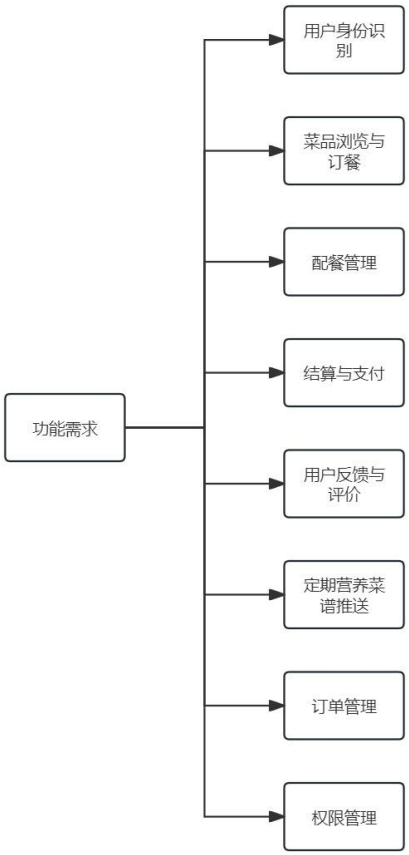

图 3.1 功能性需求

3.3 非功能需求

非功能需求是社区食堂系统设计中同等重要的一部分，以下是具体的非功能需求内容：

（1）性能需求：系统响应时间应控制在 2 秒以内，以保证用户良好的交互体验。支持高并发处理，系统应能同时处理大量用户订餐和配餐操作。

（2）安全性需求：用户数据应加密存储，确保用户隐私数据的安全性。系统应设有严格的访问控制，保证只有授权用户可以进行敏感操作，如订单管理和菜品编辑等。

（3）可靠性需求：系统应具备较高的稳定性和可靠性，确保服务 24/7 持续可用性。数据备份和恢复机制，以防止数据丢失和系统故障导致的影响。

（4）用户体验：系统界面应直观友好，易于上手和操作。用户交互流畅，保证用户操作的便捷性和流畅度。

（5）可扩展性：系统应具有良好的可扩展性，以适应未来业务发展和增长。

（6）兼容性：支持跨平台使用，能够在不同设备和不同操作系统上良好运行。

（7）文档和培训：提供系统相关的用户手册和培训资料，以便用户快速上手并提高系统使用效率。

综合考虑这些非功能需求，将有助于确保社区食堂系统具备高效、安全、稳定、易用的特性，满足用户的多样化需求。详见图 3.2 所示。

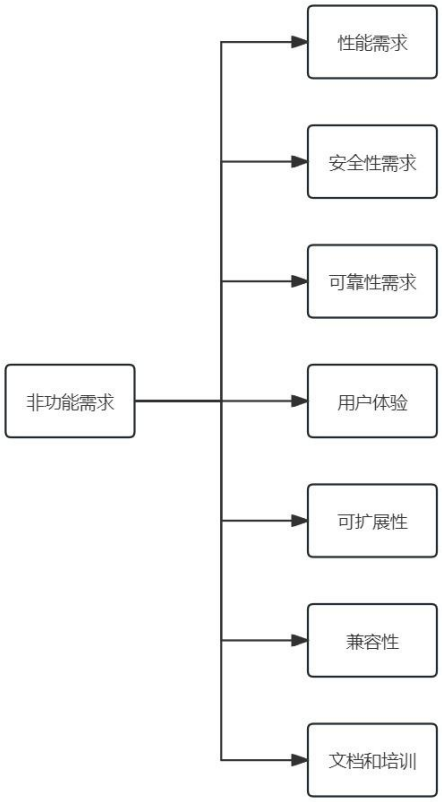

图 3.2 非功能性需求

3.4 用户需求

用户需求主要站在不同用户的角度对系统的需求进行分析，以下是具体内容：

（1）社区居民：社区居民希望能够方便快捷地预订餐品，了解菜品信息和价格，享受优质的餐饮服务体验。希望系统界面简洁明了，操作便捷，能够快速完成订餐、支付和评价等功能。

（2）食堂工作人员：食堂工作人员需要一个高效的系统来管理菜品库存、配餐进度和订单信息，确保餐品准确送达。希望系统能够提供清晰的订单管理和配餐通知功能，以提升工作效率和服务质量。

（3）系统管理员：管理员需要具备系统管理权限，进行用户管理、菜品管理、数据分析等功能。希望系统能够提供实时监控和报表功能，以便及时处理异常情况并进行数据分析优化。

（4）健康管理者：希望系统能够支持发布营养菜谱和健康信息，促进社区居民的健康饮食意识。需要系统能够追踪用户的健康信息和反馈，以便调整菜品种类和营养搭配。

综合考虑各类用户的需求，社区食堂系统应针对不同用户角色提供相应的功能和体验，以满足其多样化的需求和期望。详见图 3.3 所示。

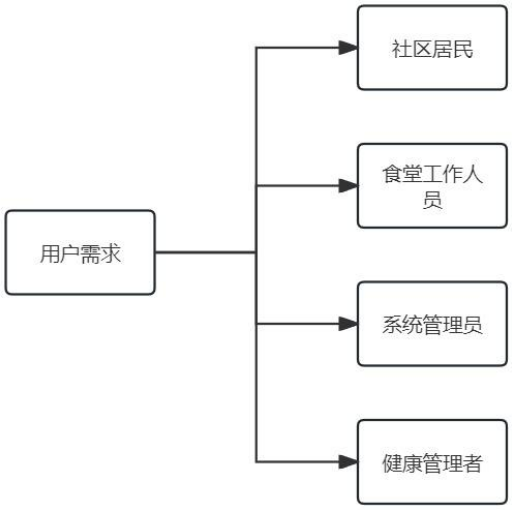

图 3.3 用户需求

3.5 安全需求

安全需求有保证系统对用户数据和核心数据管理的重要地位，其具体内容如下：

（1）用户身份认证：实现用户登录和注册时的严格身份验证机制，包括密码加密存储、多因素认证等方式，保障用户账号的安全。

（2）访问控制：区分系统管理员、食堂工作人员和普通用户的访问权限，实现对不同角色的精细访问管控。

（3）数据加密：在传输和存储过程中对用户个人信息、支付信息等敏感数据进行加密，避免数据泄露和非法获取。

（4）系统审计：系统应具备完善的审计功能，记录用户操作日志，以便追踪用户行为和系统操作，保障系统操作的可追溯性。

（5）漏洞修补：及时更新系统和组件的补丁，确保系统漏洞修补及时，防范潜在的安全威胁。

（6）灾备和恢复：设立数据备份与恢复机制，以便应对潜在的数据丢失和系统故障风险，保证系统的可靠性和持续性。

通过明确这些安全需求，社区食堂系统能够确保用户信息和交易数据的安全性，同时维护系统的可靠性和稳定性。详见图 3.4 所示。

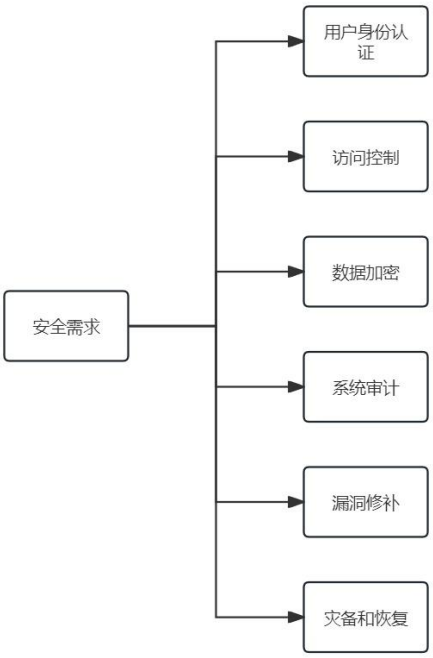

图 3.4 安全需求

3.6 可靠性需求

可靠性需求对社区食堂系统至关重要，以下是具体的内容：

（1）系统稳定性：系统应能够持续稳定运行，最大程度减少系统故障和宕机时间，确保用户服务不间断。

（2）数据完整性：系统需要保证数据的完整性，防止数据丢失、损坏或篡改，通过备份和恢复功能保障数据的可靠性。

（3）故障恢复：系统应具备快速故障定位和恢复功能，能够在发生故障时快速恢复服务，降低故障对用户的影响。

（4）容错性：系统应当具备一定的容错能力，对一些可预期的错误和异常情况进行处理和自我修复，确保系统的稳定性和可靠性。

（5）服务质量监控：实现对系统性能、负载、响应时间等关键指标进行实时监控和统计，以保证系统的稳定性和性能质量。

考虑到这些可靠性需求，社区食堂系统将能够为用户提供稳定可靠的餐饮预订服务，增强用户对系统的信任和满意度。详见图 3.5 所示。

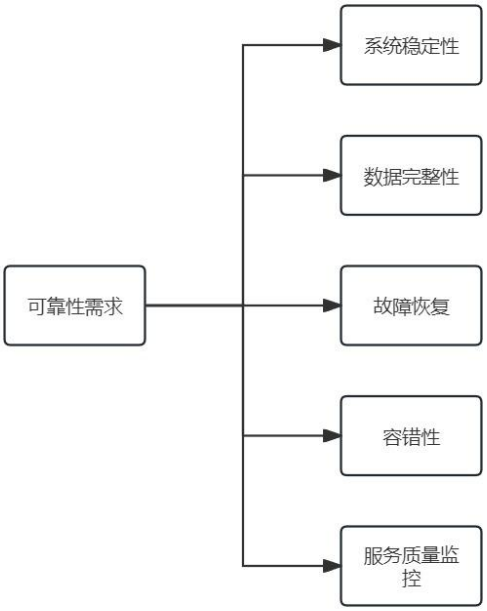

图 3.5 可靠性需求

3.7 性能需求

社区食堂系统的性能需求直接影响到用户的体验感受，以下是具体的需求内容：

（1）响应时间：系统应具有快速的响应时间，用户的操作应该在 2 秒内得到响应，以提供良好的用户体验。

（2）并发处理：系统应具备处理大量并发用户请求的能力，特别是在用餐高峰期，能够有效处理大量用户的订餐和配餐操作。

（3）负载均衡：在系统设计中考虑负载均衡机制，以保证系统各个功能模块的负载均衡，提高系统整体性能和稳定性。

（4）数据查询效率：数据库查询效率要求较高，系统需具备快速的数据检索和处理能力，确保菜品浏览、订餐记录查询等功能的高效完成。

（5）系统可扩展性：系统需要具备一定的可扩展性和灵活性，以便在业务增长和流量增加时能够方便地进行系统升级和扩展。

通过满足这些性能需求，社区食堂系统将能够提供高效、稳定和可靠的服务，满足用户在不同场景下的使用需求。详见图 3.6 所示。

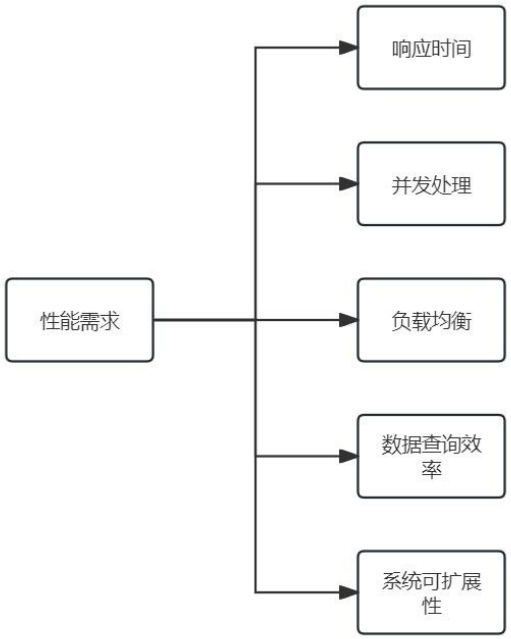

图 3.6 性能需求

3.8 本章小结

本章对社区食堂系统的需求进行了深入探讨，其中涵盖了安全性、可靠性和性能需求等多个方面。在安全需求方面，我们强调了用户身份认证、访问控制和数据加密的重要性，以确保用户信息和交易数据的安全。可靠性需求方面，系统稳定性、数据完整性和故障恢复被视为保障系统可靠性的关键要素。而在性能需求方面，我们关注系统响应时间、并发处理能力和负载均衡，以提供高效和稳定的服务。

通过本章的需求分析，我们对社区食堂系统的设计和实施提供了有力的依据和指导，并为相关研究和实践提供了有益的参考和借鉴。

4 数据库设计

4.1 引言

数据库设计是信息技术领域中至关重要的一环，它直接影响着数据管理系统的性能、可靠性和效率。随着数据量的不断增长和信息化程度的提升，设计出合理、高效的数据库结构变得愈发重要。一个精心设计的数据库不仅能够提升数据的存储和检索效率，还能够为企业决策提供有力的支持。本章将探讨数据库设计的基本概念和方法并实现系统的数据库设计，旨在深入了解社区食堂系统数据库设计的重要性和合理性。通过本章的学习，将能够掌握一些常见的数据库设计原则，为日后的数据库设计工作奠定坚实的基础。

4.2 数据库的设计与规范化

数据库设计是一项杂而重要的任务，它涉及到数据结构、关系建立、数据存储等方面。规范化是数据库设计的重要概念，旨在消除数据冗余和维护数据一致性。

确定数据库系统的需求和目标，包括数据存储需求、查询需求、性能需求等。建立数据模型，包括实体关系图（ER 图 Entity Relationship Diagram）、数据表结构等，确定数据之间的关系。将概念模型转化为关系模式，确定数据表的结构、主键、外键等。确定实际存储方案、索引设计、性能优化等，将数据库设计映射到具体的数据库管理系统中。

第一范式（1NF First Normal Form）：确保每个数据项都是不可再分的原子值，即每个数据项都是最小的数据单元。第二范式（2NF）：满足第一范式，并且非主键属性完全依赖于候选键，消除了部分依赖。第三范式（3NF）：满足第二范式，所有非主键属性都不能依赖于其他非主键属性，消除了传递依赖。三者之间的大致关系如图 4.1 所示。

通过规范化，可以减少数据冗余，避免数据存储不一致导致的问题。规范化可以提高数据库中数据的质量和一致性，减少数据更新异常的发生。规范化的数据库结构更易于维护、扩展和修改，使数据库更灵活、易于管理。

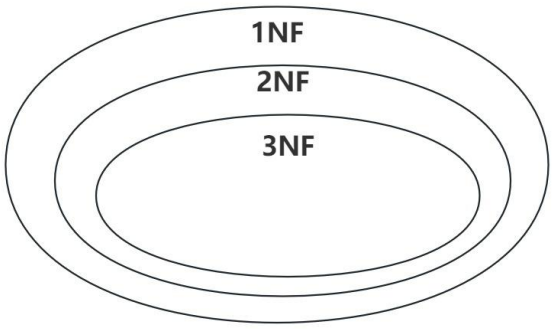

图 4.1 范式概念图

在本研究中，将使用第三范式的设计规范设计社区食堂系统的数据表。通过第三范式的规范化过程，将数据拆分为更小的关系表，确保每个表中的数据都只与一个主键相关联，从而消除了传递依赖。这有助于保持数据的一致性，避免数据更新异常，如插入异常、更新异常和删除异常。消除了非主键属性对其他非主键属性的依赖，避免了数据的冗余存储。这意味着每个数据项只需存储一次，减少了存储空间的占用以及数据冗余可能导致的数据不一致性。通过将数据拆分成具有最小重复数据的表，第三范式可以提高数据的质量和一致性。数据在不同的表中更清晰地组织。

4.3 社区食堂系统数据库信息设计

在本研究中，社区食堂系统主要涉及的数据表包括：用户登录注册信息表、用户头像表、用户购物车表、用户订单表、系统公告表、菜品详情表。

用户登录注册信息表包括管理员和用户的账号和密码信息，登录系统前对用户信息进行验证，从而保护用户信息的安全性。

用户头像表保存用户的网络外貌信息，使系统对每个用户而言更具有独立性和特殊性。

用户购物车表包括用户的所有购物车信息，用户可以对感兴趣的菜品进行增删改查，从而匹配出符合用户需求的菜品内容，促进用户购买和支付的成功性。

用户订单表存储用户的所有购买信息，用户可以对所有购买记录进行回顾查看，也可以选择删除历史购买信息，提供了用户重复购买同一商品的理由。

系统公告表保存系统公告轮播图中的所有公告信息，可以让商家和管理员对公告进行修改，用户可以通过系统公告第一时间了解到社区食堂的最新动态信息。

菜品详情表包括系统中所有展示的菜品内容，商家可以对本商店菜品进行增删改查，管理员可以对所有商家的菜品信息进行管理，用户可以通过系统选择意向商家的菜品进行购买，促进了社区食堂菜品最新信息的管理力度。

4.3.1 用户登录注册信息表

用户登陆注册信息表中的实体是用户登录系统所需要的信息。用户登录系统信息的属性分别有用户名、密码和用户序号。用户名为实体的主键，唯一标识一名用户的登录系统信息。其属性关系见图 4.2 所示，属性信息见表 4.1 所示。

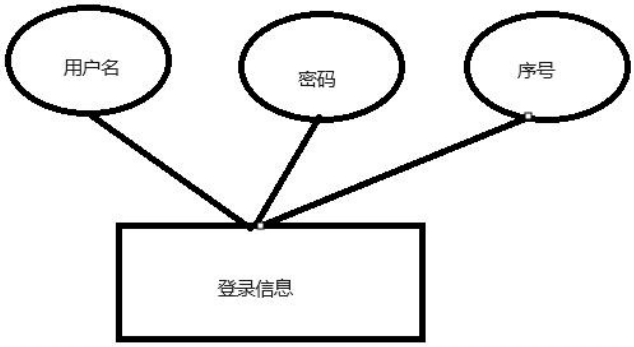

图 4.2 登录注册属性关系图

表 4.1 用户登陆注册信息表

序号	字段名称	类型	长度	主键	描述
1	username	varchar	10	是	用户名
2	pwd	varchar	20	否	密码
3	curid	int	/	否	序号

4.3.2 用户头像表

用户头像信息表中的实体是用户头像信息。用户头像信息的属性分别有用户名和用户头像链接地址。用户名为实体的主键，唯一标识一名用户的头像信息。其属性关系见图 4.3 所示，属性信息见表 4.2 所示。

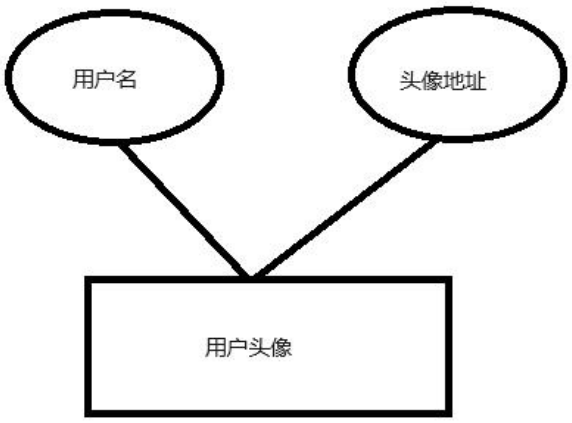

图 4.3 用户头像属性关系图

表 4.2 用户头像表

序号	字段名称	类型	长度	主键	描述
1	username	varchar	10	是	用户名
2	userIconPath	varchar	1024	否	头像路径

4.3.3 用户购物车表

用户购物车信息表中的实体是用户购物车信息。用户购物车信息的属性分别有菜品名称、菜品价格、购买数量、菜品图片链接地址和菜品序号。菜品名称为实体的主键，唯一标识购物车中一个菜品的选购信息。其属性关系见图 4.4 所示，属性信息见表 4.3 所示。

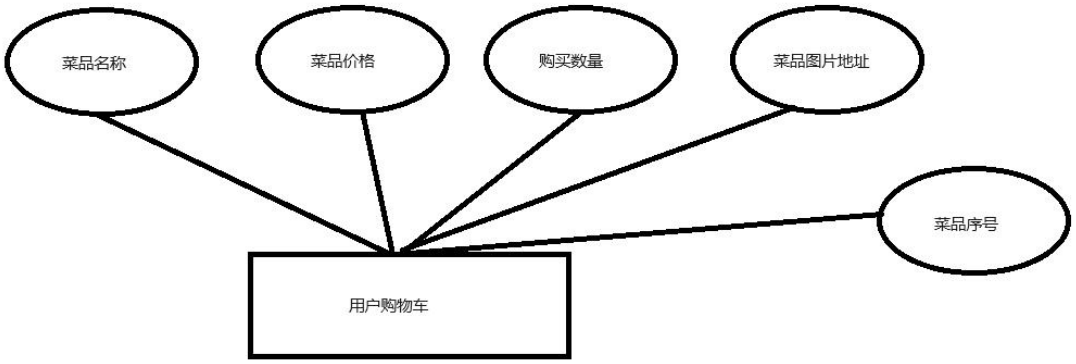

图 4.4 购物车属性关系图

表 4.3 用户购物车表

序号	字段名称	类型	长度	主键	描述
1	DishName	varchar	50	是	菜品名称
2	DishMoney	varchar	50	否	菜品价格
3	DishBuyNum	varchar	50	否	购买数量
4	DishImagePath	varchar	1024	否	图片路径
5	DishIndex	int	/	否	序号

4.3.4 用户订单表

用户订单信息表中的实体是用户订单信息。用户订单信息的属性分别有菜品名称、菜品价格、购买数量、菜品图片链接地址和订单产生时间。菜品名称为实体的主键，唯一标识用户订单中一个菜品的购买信息。其详细属性关系结果见图 4.5 所示，属性详细信息见表 4.4 所示。

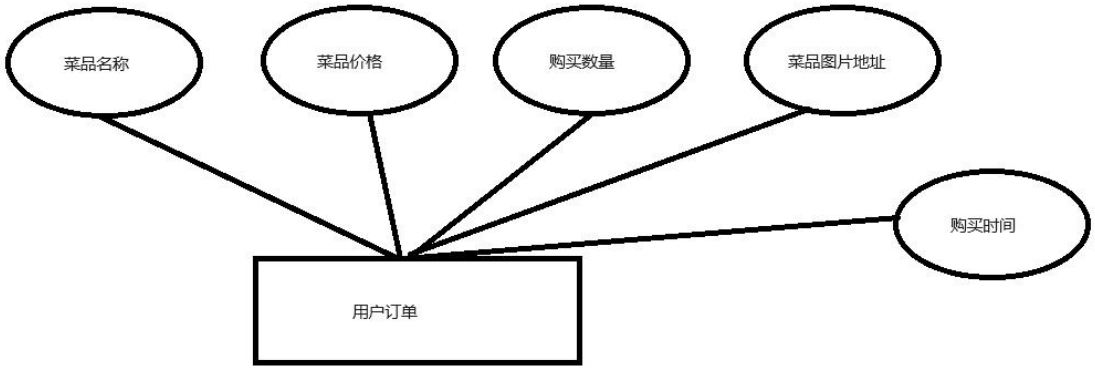

图 4.5 用户订单属性关系图

表 4.4 用户订单表

序号	字段名称	类型	长度	主键	描述
1	DishName	varchar	50	是	菜品名称
2	DishMoney	varchar	50	否	菜品价格
3	DishBuyNum	varchar	50	否	购买数量
4	DishImagePath	varchar	1024	否	图片路径
5	TakeOrderTime	datetime	/	否	购买时间

4.3.5 系统公告表

系统公告信息表中的实体是系统公告信息。系统公告信息的属性分别有公告信息图片链接地址和公告序号。公告信息链接地址为实体的主键，唯一标识系统公告信息中一个公告的信息。其属性关系见图 4.6 所示，属性信息见表 4.5 所示。

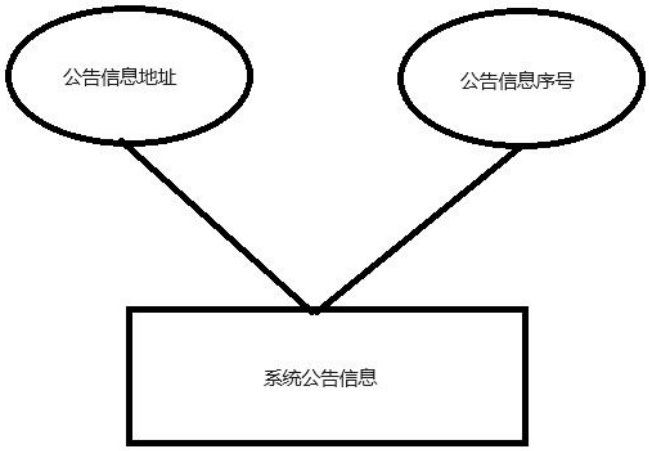

图 4.6 系统公告属性关系图

表 4.5 系统公告表

序号	字段名称	类型	长度	主键	描述
1	AnnoImagePath	varchar	1024	否	图片位置
2	AnnoIndex	int	/	是	序号

4.3.6 菜品详情表

菜品详情信息表中的实体是菜品信息。菜品信息的属性分别有菜品名称、菜品价格、菜品剩余数量、菜品图片链接地址和菜品序号。菜品名称为实体的主键，唯一标识菜品信息中一个菜品的信息。其属性关系见图 4.7 所示，属性信息见表 4.6 所示。

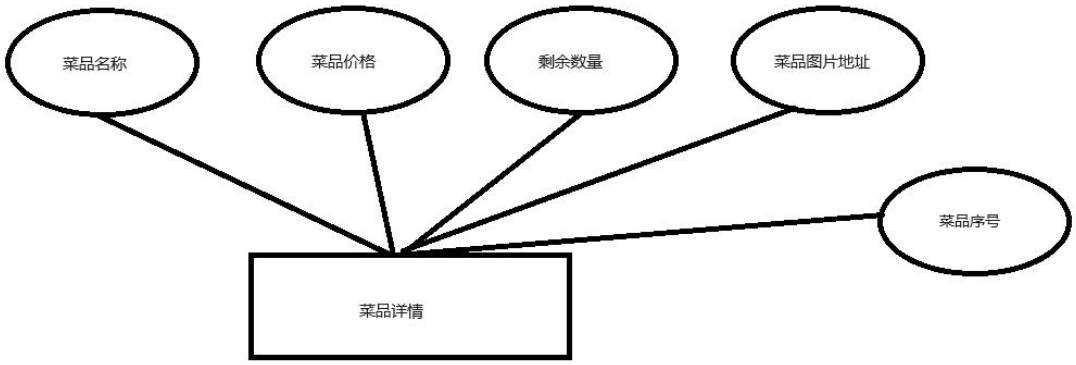

图 4.7 菜品详情信息实体关系图

表 4.6 菜品详情表

序号	字段名称	类型	长度	主键	描述
1	DishName	varchar	50	是	菜品名称
2	DishMoney	varchar	50	否	菜品价格
3	DishNum	varchar	50	否	剩余数量
4	DishImagePath	varchar	1024	否	图片路径
5	DishIndex	int	/	否	序号

4.4 系统 E-R 图

社区食堂系统的数据库中，每个数据库的功能都与当前登录系统的用户有关。其系统实体关系见图 4.8 所示。

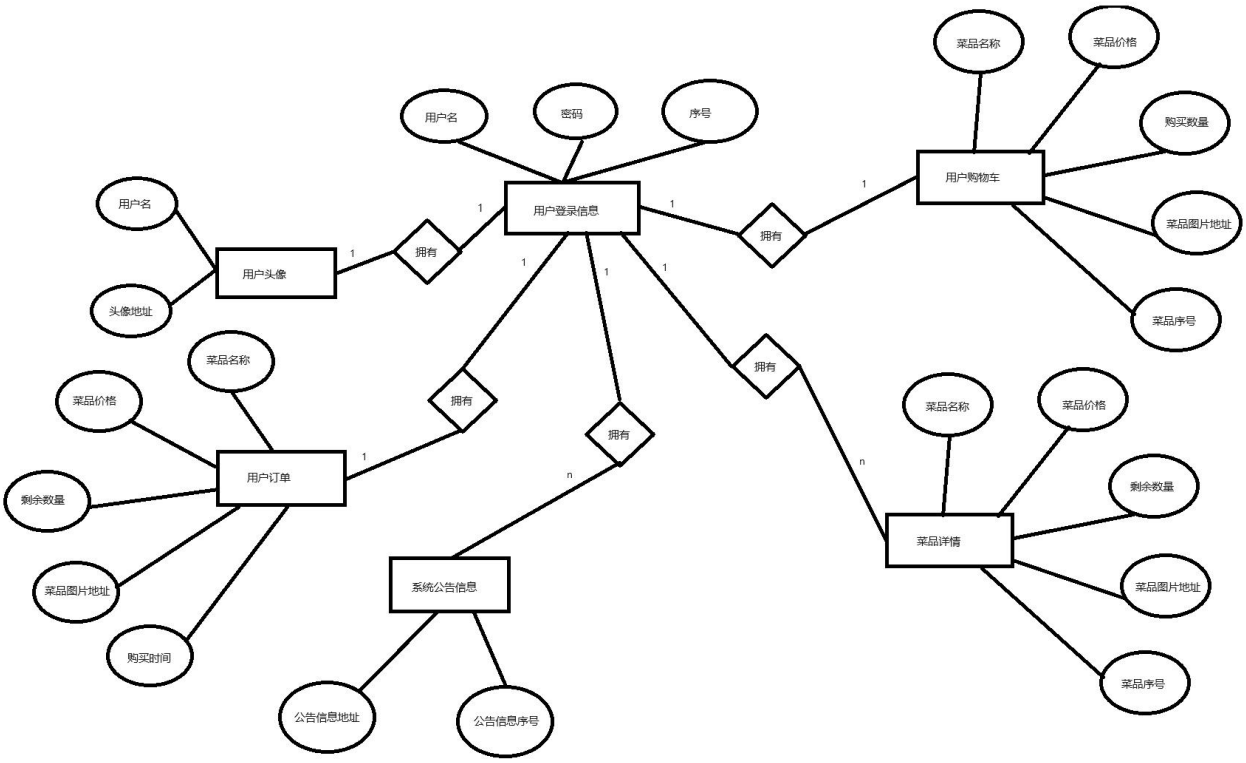

图 4.8 社区食堂系统实体关系图

4.5 本章小结

本章深入探讨了数据库设计规范化的重要性和系统数据库的实现过程。数据库是一种组织数据的结构化方式，数据库的设计结构直接影响到数据的获取和存储等性能问题。数据库设计和规范化是确保数据一致性和质量的关键步骤，其中规范化的第三范式（3NF）有助于消除数据冗余、提高数据一致性和可靠性。这些概念和技术对于数据库应用的合理运用和管理至关重要，为数据驱动的应用提供了可靠的基础。

5 系统功能介绍及实现

5.1 引言

系统设计与实现作为社区食堂系统建设的重要环节，涉及到系统架构、功能模块、用户界面设计、系统性能优化等方方面面。合理的系统设计能够确保系统具有良好的稳定性和扩展性，而高效的系统实现则能够将设计转化为现实，并为用户提供流畅的使用体验。本节将深入探讨社区食堂系统的系统设计与实现过程，详细介绍系统架构的设计思路和功能模块的划分。通过对系统设计与实现的深入分析，将能够全面了解社区食堂系统的构建过程，为类似系统的设计与实现提供借鉴与参考。

5.2 社区食堂系统功能流程介绍

社区食堂系统的主要功能流程见图 5.1，其主要实现功能流程如下：

用户开启系统后进入登录窗口，用户输入用户名和密码成功后则进入系统主窗口，否则用户需要先注册用户，注册成功则返回登录窗口重新登录，否则持续注册直到成功。

管理员通过管理员账号登录系统后进入管理员窗口，管理员可以对用户的账号信息通过账号管理窗口进行管理。管理员可以对公告信息进行修改和替换，控制系统中全局的公告信息，实时展示最新公告。管理员可以对每个菜品进行信息管理，实现对菜品的增删改查功能。

用户通过用户账号进入用户窗口，用户可以浏览食堂中当前上架的最新菜品信息和公告信息。用户可以将有意向的菜品加入购物车中，并在购物车中对菜品进行最终购买。购物车中可以对用户的菜品进行选中和删除，实现用户对购物车中菜品的自定义内容的修改。用户可以对食堂中的菜品直接进行购买，通过付款吗实现付款后，可以将订单锁死到订单详情中进行再一次查看。用户可以通过历史订单查看所有购买过的菜品记录，也可以对菜品记录进行删除，维护用户的个人隐私。用户可以通过返回登录窗口，切换新的用户身份重新登录食堂系统进行新一轮的购买活动。

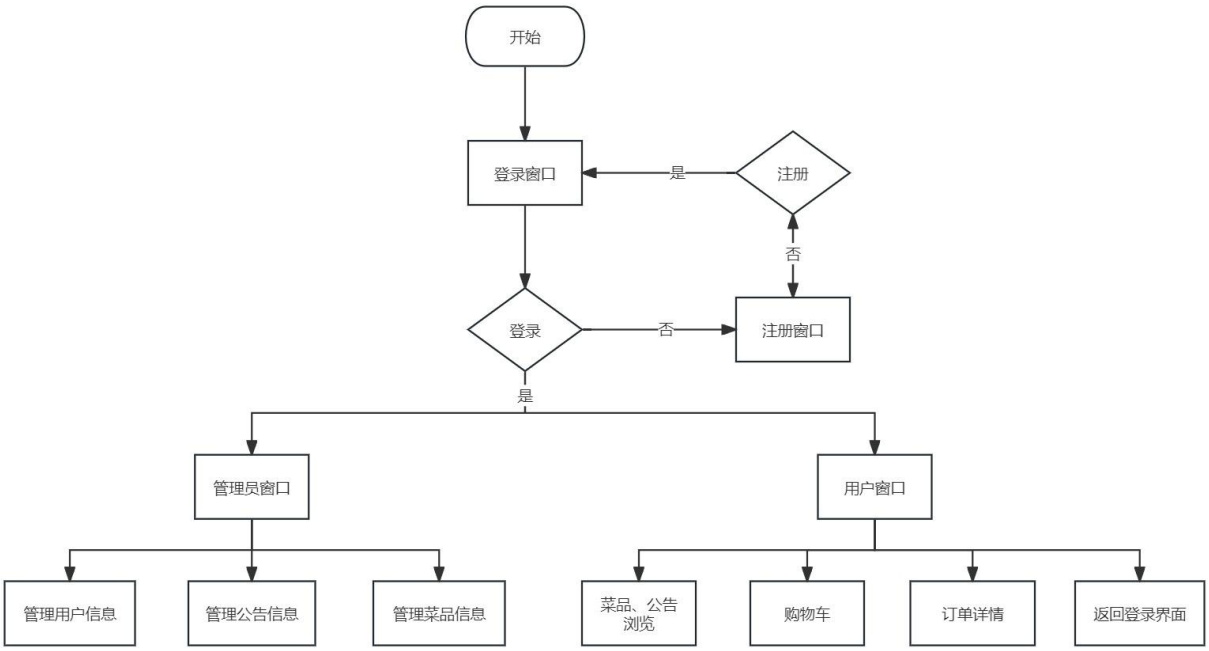

图 5.1 系统功能流程图

5.3 系统功能模块设计介绍

在本研究中，社区食堂系统主要涉及的功能模块包括：用户登录注册功能、用户头像功能、公告轮播图功能、菜品浏览功能、购物车功能、订单详情功能、用户管理功能。每个功能模块的界面设计都将先采用 Qt 的内置设计工具 Qt Designer 进行初步设计，再使用代码的方式进行二次设计。接下来将对这些功能的功能模块和用户界面设计进行详细的介绍。

5.3.1 登录功能介绍

a. 功能模块介绍

在登陆界面中，可以切入注册界面进行注册，注册界面也可以重新切回登录界面进行登录。在登陆界面中，空提交将默认报错内容不能为空。当输入正确的账号和密码后，将登录用户成功，进入系统用户界面。当输入的用户名正确，而密码错误时，则提示重新输入。当输入的用户名错误时，则提示需要先注册用户再进行登录。登录模块功能流程见图 5.2 所示。

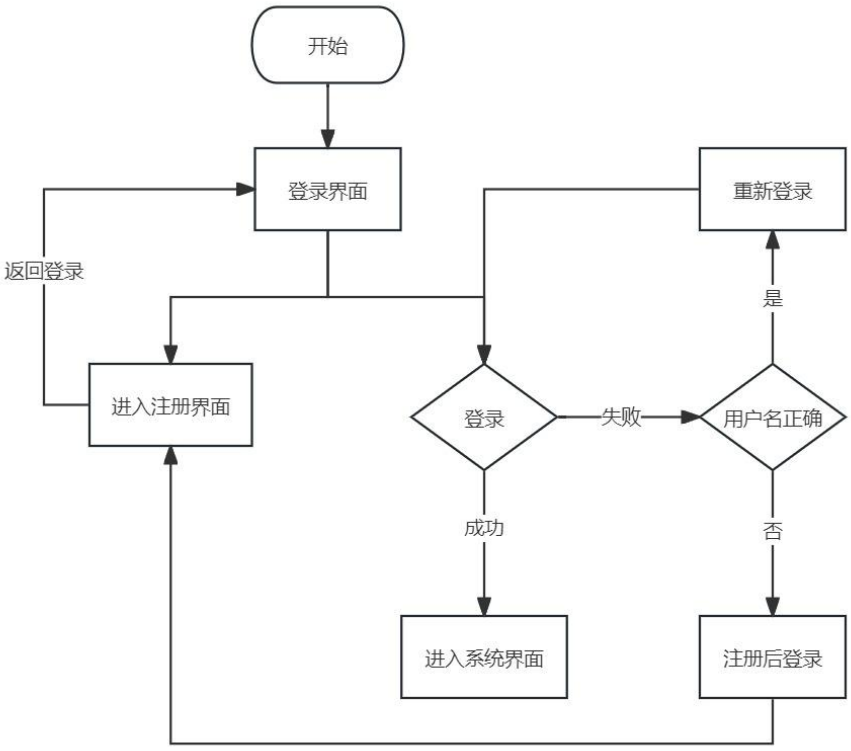

图 5.2 登录模块

b. 界面设计介绍

登录界面设计中，大标题使用 QLabel 默认设置社区食堂系统，登录入口使用 QGroup 将组件存储在容器中。登录输入内容主要使用两个 QLabel 和两个 QLineEdit 进行设计，并使用两个 QPushButton 作为登录和注册的触发按钮。最后，使用 QSS 将组件进行美化。登录界面设计见图 5.3 所示。

图 5.3 登录界面设计

5.3.2 注册功能介绍

a. 功能模块介绍

在注册界面中，可以切入登录界面进行登录，登录界面也可以重新切回注册界面进行注册。在注册界面中，空提交将默认报错内容不能为空。当输入非已注册用户的账号名并输入密码后，将注册用户成功，此时可以选择继续注册用户或者切换回登录界面进行登录。当输入的用户名为已注册用户时，提示重命名后重新输入。注册模块功能流程见图 5.4 所示。

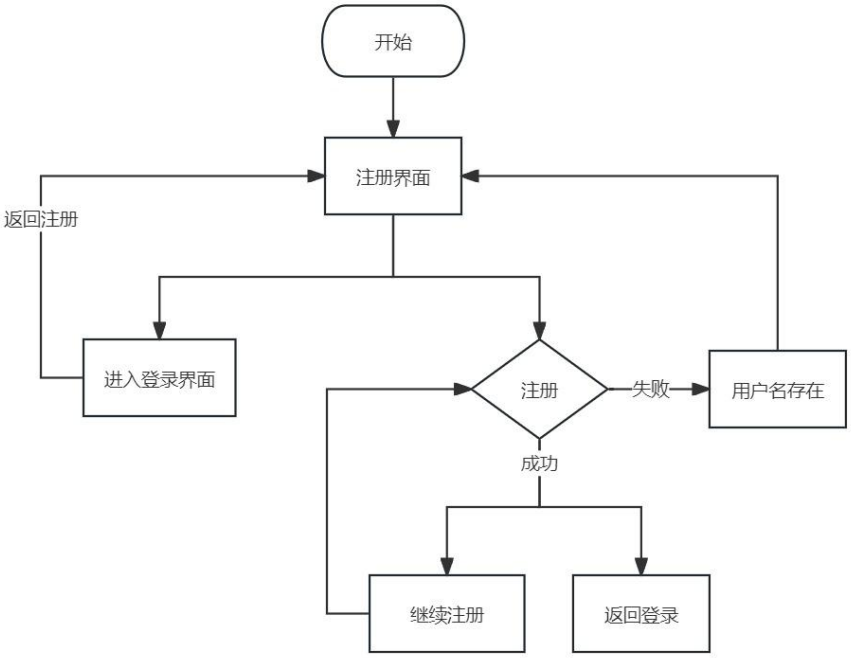

图 5.4 注册模块

b. 界面设计介绍

注册界面设计中，大标题使用 QLabel 默认设置社区食堂系统，注册使用 QGroup 将组件存储在容器中。注册输入内容主要使用两个 QLabel 和两个 QLineEdit 进行设计，并使用两个 QPushButton 作为注册和返回登录的触发按钮。最后，使用 QSS 将组件进行美化。注册界面设计见图 5.5 所示。

图 5.5 注册界面设计

5.3.3 用户头像功能介绍

a. 功能模块介绍

进入系统界面后，用户头像功能会将用户头像数据库中的该用户数据读出初始化自身。当用户点击头像时，可以选择是否需要重新设置网络外貌。如果取消，则默认使用旧头像。如果设置头像，则在文件选择界面中，选择需要设置的头像，确定后将新头像设置为自身头像。用户头像模块功能流程见图 5.6 所示。

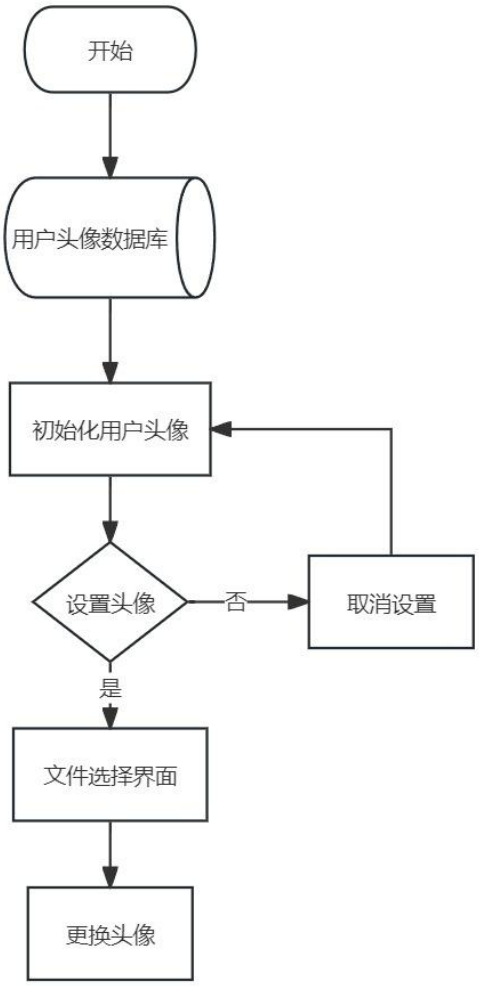

图 5.6 用户头像模块

b. 界面设计介绍

用户头像界面设计中，用户头像承载容器使用 QLabel 作为头像内容展示容器。当用户为初始用户时，头像内容为空，显示一段文字“请设置用户头像”对用户进行头像设置提示。当用户设置头像内容后，头像数据将被持久化到该用户的数据库中，再次登录时不需要再次设置头像内容。用户头像界面设计见图 5.7 所示。

图 5.7 用户头像界面设计

5.3.4 公告轮播图功能介绍

a. 功能模块介绍

进入系统界面后，系统公告功能将读取系统公告数据库中内容，使用数据初始化自身。无论任何用户进入系统公告界面对公告都可以进行左右切换查看公告内容，默认情况下系统公告会在 5s 后进行自动的内容切换，实现自动的系统公告内容轮播。当用户为普通用户时，用户点击公告不会出现任何交互界面。当用户为管理员用户时，用户点击公告时正常的公告轮播会暂时被停止，用户将对当前停止的界面公告进行修改，修改成功后，系统公告将重新进入默认轮播状态。系统公告模块功能流程见图 5.8 所示。

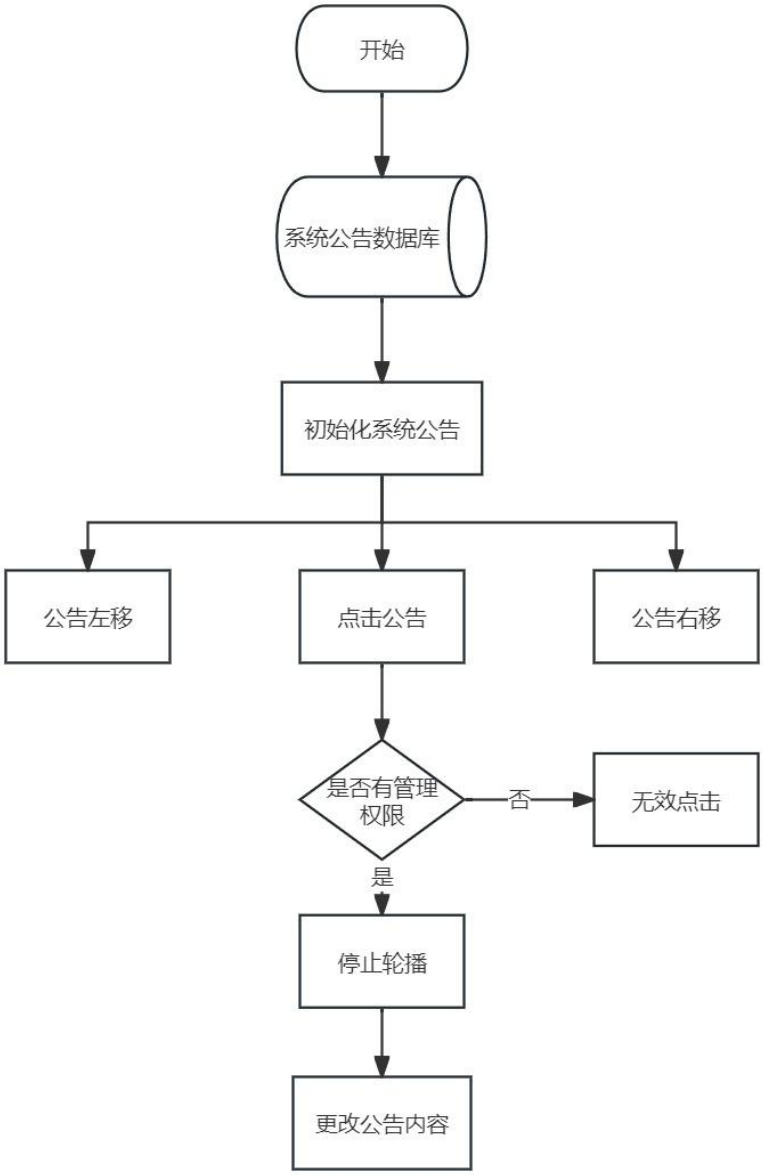

图 5.8 系统公告模块

b. 界面设计介绍

系统公告界面设计中，左右两个按钮使用 QPushButton 进行设计，中间的展

示公告使用 QLabel 作为公告内容的承载容器。最后，使用 QSS 进行界面优化。系统公告界面设计见图 5.9 所示。

图 5.9 系统公告界面设计

5.3.5 菜品浏览功能介绍

a. 功能模块介绍

进入系统界面后，菜品浏览功能将读取菜品详情数据库中的内容，并将数据初始化自身。除去菜品数据库中的被添加菜品外，分页器还会在队尾添加一个空菜品元素进入队列中。所有用户对有效菜品的分页查看都是全部展示的状态。当分页器查看到空菜品所在页时，将对用户是否为管理员进行判断，是否对空菜品进行展示，空菜品只对管理员展示，普通用户不做展示。菜品分页器模块功能流程见图 5.10 所示。

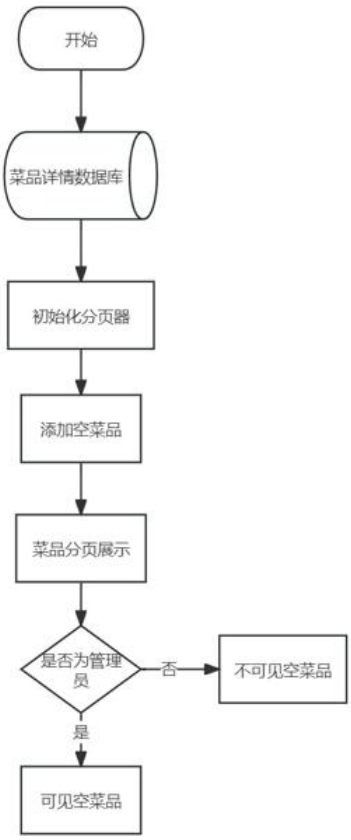

图 5.10 菜品分页器模块

b. 界面设计介绍

在菜品分页器界面设计中，界面只使用两个 QPushButton 和一个 QLineEdit 进行外观设计。菜品分页器的功能主要组合菜品展示器和底层逻辑代码进行数据控制和展示。菜品分页器界面设计见图 5.11 所示。

图 5.11 菜品分页器界面设计

5.3.6 菜品展示器功能介绍

a. 功能模块介绍

当菜品数据被传入菜品展示器功能后，将使用菜品数据初始化自身并生成该菜品的展示界面。当用户点击界面上的加入购物车按钮时，则将该菜品加入到购物车模块中。当用户点击立即购买功能时，将产生购买界面等待用户付款后生成购物订单。当普通用户点击菜品展示界面时，界面不产生任何交互。当管理员点击菜品展示界面时，将生成该菜品的菜品选项卡，管理员可以对该菜品的内容进行操作。当该菜品为空菜品时，默认该菜品为新增菜品，管理员只能对该菜品进行内容新增或者关闭新增。当该菜品为已添加菜品时，管理员可以对已添加菜品的菜品信息进行合理修改，也可以对该菜品进行删除或者关闭菜品选项卡什么都不做。菜品展示器模块功能流程见图 5.12 所示。

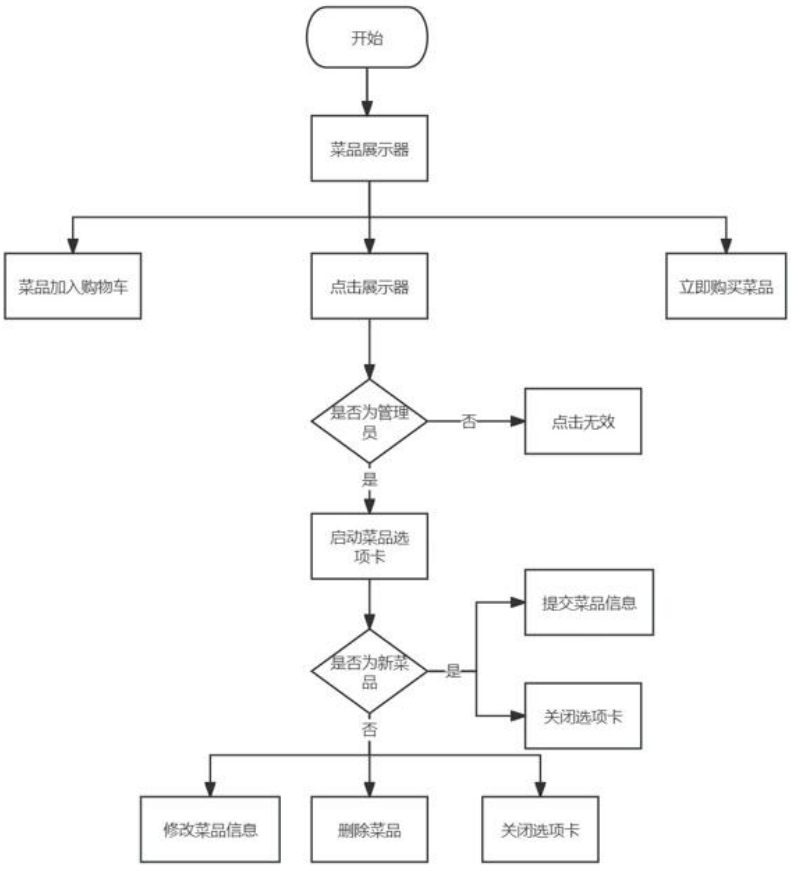

图 5.12 菜品展示器流程模块

b. 界面设计介绍

在菜品展示器界面设计中，顶部使用一个 QLabel 作为菜品展示的容器，使用 QVBoxLayout 作为菜品信息展示的布局容器，内部使用三个 QLabel 作为内容展示的容器，最后使用两个 QPushButton 作为加入购物车和立即购买功能的触发按钮。最后，使用 QSS 进行界面优化。菜品展示器界面设计见图 5.13 所示。

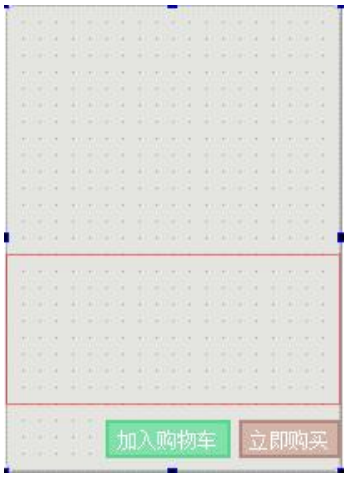

图 5.13 菜品展示器界面设计

5.3.7 购物车功能介绍

a. 功能模块介绍

当用户点击购物车功能后进入购物车界面时，购物车功能将读取用户购物车数据库中的内容，将数据初始化自身。用户通过选中和取消菜品的方式对所需菜品进行选中，用户可以选择对这些菜品进行删除或者购买。当用户选择删除时，这些菜品将会全部删除，留下未被选中的菜品在购物车中。当用户选择购买菜品时，这些菜品将被全部购买，其购买价格除了会通过消息窗口通知外，还将动态的出现在购物车界面中给予用户价格提示，当用户确认购买后，这些被购买的菜品将生成对应的菜品购买订单进入订单模块。购物车功能模块流程结果见图 5.14 所示。

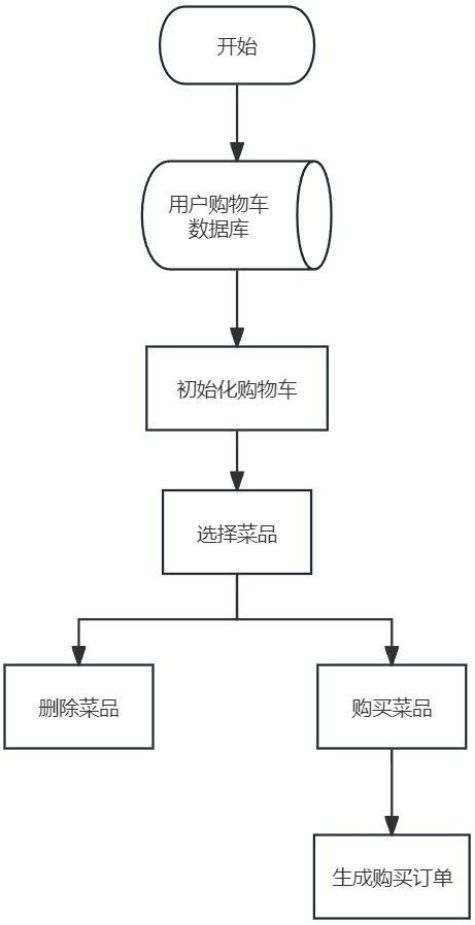

图 5.14 购物车模块

b. 界面设计介绍

在购物车设计界面中，顶部使用两个 QLabel 进行展示，一个作为购物车 Icon 承载容器，一个作为购物车名称承载容器。中间使用一个 QScrollArea 作为菜品的展示界面使用，所有菜品都将被展示在滚动区内。底部使用一个 QCheckBox 作为菜品内容的全选、半选和取消按钮的功能按钮。使用三个 QPushButton 分别作为删除选中、购买和取消的功能按钮使用，并使用一个 QLabel 作为选中菜品的价格展示，选中菜品的价格将进行动态的价格展示，增加了用户在使用购物车功能时的购物体验。最后，使用 QSS 进行界面优化。购物车界面设计见图 5.15 所示。

图 5.15 购物车界面设计

5.3.8 订单详情功能介绍

a. 功能模块介绍

当用户点击订单详情功能后，用户进入订单详情界面。订单详情功能将读取用户的订单数据库内容，将数据初始化订单界面。用户可以通过订单详情界面查看历史订单条，对不需要的历史订单可以进行永久删除，对于需要详细查看的订单条内容可以点击查看详情查看订单中购买的所有菜品内容。订单详情模块功能流程见图 5.16 所示。

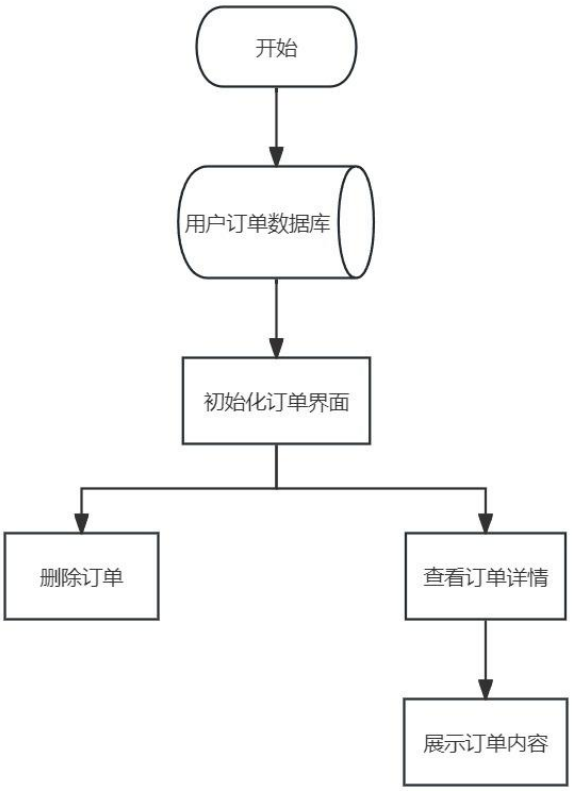

图 5.16 订单详情模块

b. 界面设计介绍

在订单详情界面设计中，顶部使用两个 QLabel 容器，一个承载图片，一个承载“订单详情”文字内容。订单详情设计中，主要使用中间的 QScrollArea 作为所有订单条的展示界面。订单详情界面设计结果图见图 5.17 所示。

图 5.17 订单详情界面设计

在订单详情界面中，组合了订单条作为缩略的订单内容展示。订单条界面主要使用一个 QLabel 和两个 QPushButton 作为主要控件，QLabel 承载订单的产生

时间内容，一个 QPushButton 作为删除订单的按键，另一个 QPushButton 作为详情查看的按键。最后，使用 QSS 进行界面优化。订单条界面设计见图 5.18 所示。

图 5.18 订单条界面设计

在订单条界面中，当点击查看订单详情按钮后，将打开订单菜品界面。订单菜品界面主要由一个 QLabel 承载订单产生时间，一个 QPushButton 作为关闭订单菜品界面的按钮。在订单菜品界面中，最重要的菜品展示界面为 QScrollArea 界面，其内部承载所有菜品内容进行展示。最后，使用 QSS 进行界面优化。订单菜品界面设计见图 5.19 所示。

图 5.19 订单菜品界面设计结果图

5.3.9 用户管理功能介绍

a. 功能模块介绍

当用户在系统控制界面中以管理员身份进入系统时，用户管理功能可以显现，如果该用户为普通用户，则该功能无法被触发。当管理员触发用户管理功能后将进入用户管理界面。在用户管理界面中，用户可以通过返回主界面功能重新返回系统界面。用户在用户管理界面中主要使用的三种功能池为持久化功能池、修改功能池、查看功能池。持久化功能池中分别有全部保存功能和保存当前行功能，这些功能将实际改变用户数据库中的数据，将界面中的变更内容持久化到用户数据库中。修改功能池中分别有插入条目功能、添加条目功能和删除条目功能，这些功能将在用户管理界面中将用户数据进行修改和删除。查看功能池中包含更新状态功能、清空界面功能和加载登录信息功能，其中更新状态功能包含了清空界面功能和加载登录信息功能，这些功能对用户管理界面的查看进行修改，将用户数据库中的数据从数据库中展示到界面中进行操作。用户管理模块功能流程见图 5.20 所示。

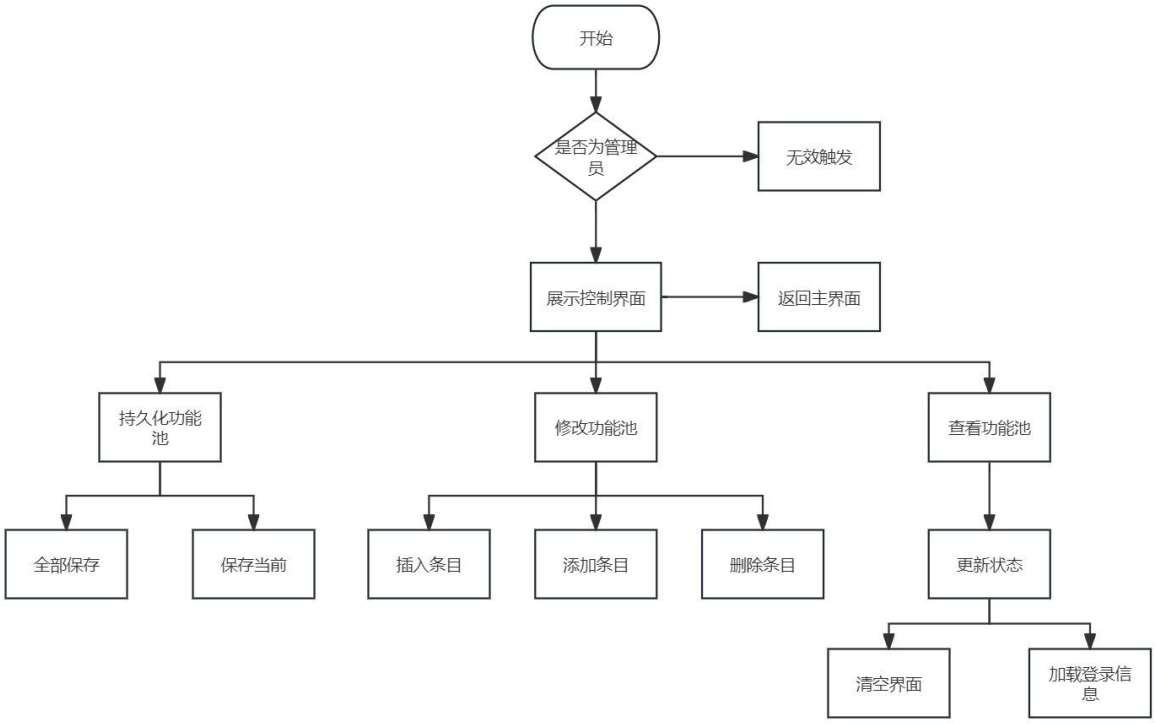

图 5.20 用户管理模块

b. 界面设计介绍

在用户管理界面设计中，使用到了一个 QTableWidget 作为数据的承载容器、九个 QPushButton 作为对数据控制的指令按钮、两个 QLabel 和一个 QProcessBar 作为展示任务执行的当前状态的容器。用户管理界面设计见图 5.21 所示。

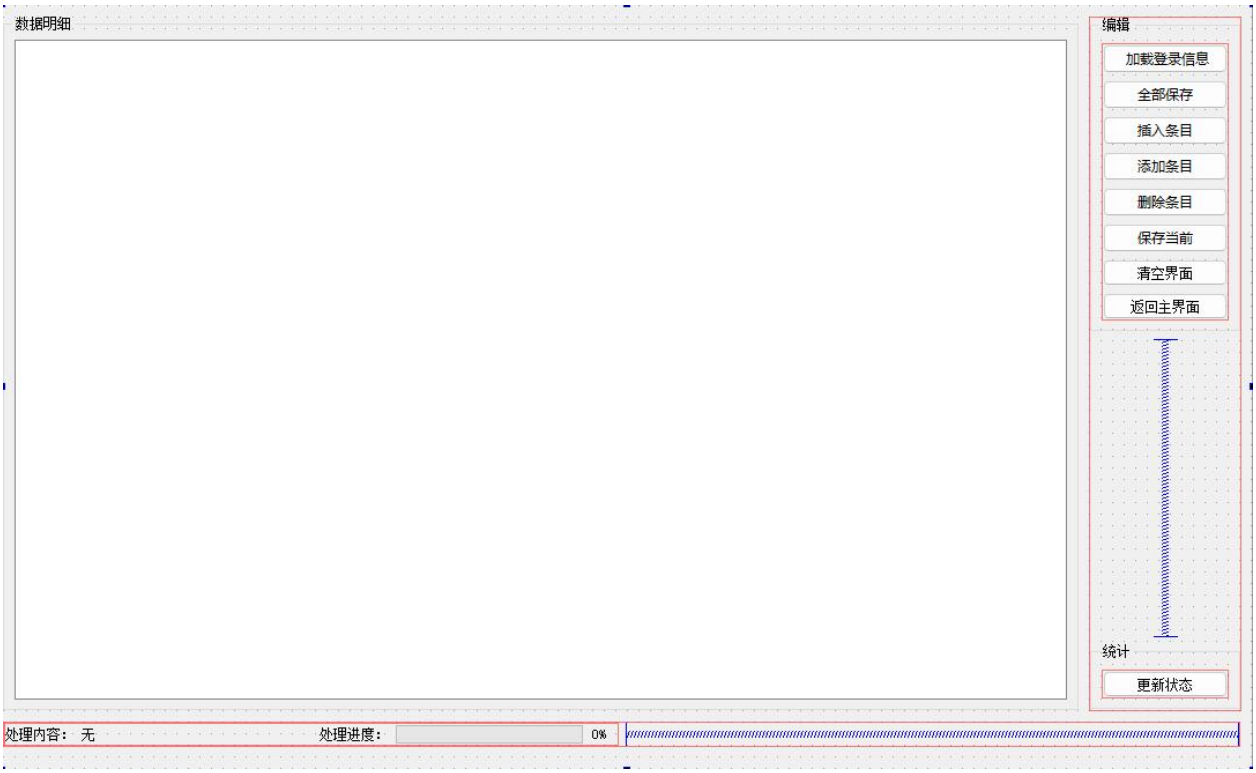

图 5.21 用户管理界面设计

5.4 系统功能实现

接下来将对这些功能的具体代码实现和实现结果进行讲解和展示。

5.4.1 登录功能实现

在登录界面设计的基础上，使用代码的方式对界面再次进行二次构造。将登录窗口固定大小为 512*527 的大小，并默认窗口名称为“用户登录”，最后对窗口的功能按键设置信号槽连接。

登录窗口的构造函数中的代码如下：

```
QPixmap pixmap("D:/MyDesktop/Graduation/ME/CommunityCanteenSys/Image/Background_PICS/login.jpg");//设定图片
    pixmap = MenuAlgorithm::PixmapToRound(pixmap,0,this->width(),this->height());
    QPalette palette;//创建一个调色板对象
    palette.setBrush(this->backgroundRole(),QBrush(pixmap));//用调色板的画笔把映射到 pixmap 上的图片画到 frame.backgroundRole()这个背景上
    this->setPalette(palette);//设置窗口调色板为 palette，窗口和画笔相关联

    watchEye = new HoverableLabel(this);
    QImage *image = new QImage("D:/MyDesktop/Graduation/ME/CommunityCanteenSys/Image/OrderDetail_ICONs/watchDetail.png");
    image = new QImage(image->scaled(30, 30, Qt::IgnoreAspectRatio, Qt::SmoothTransformation));
    watchEye->setPixmap(QPixmap::fromImage(*image));
    watchEye->move(414, 277);
    watchEye->setMinimumSize(30,30);
    watchEye->setStyleSheet(
                "QLabel{background-color: rgb(178,180,164,40%); border-radius: 15px;border: 1px;}"
                "QLabel:hover{background-color: gray; border-radius: 15px;border: 1px;}"
                );

    this->setFixedSize(512,527);
```

```
    this->setWindowTitle("用户登录");


connect(&re,&registerwin::register_complete,this,&loginwin::after_register_login);

connect(&control_win,&controlwin::back_login_paper,this,&loginwin::after_control_win);

connect(this->watchEye,&HoverableLabel::mouseEntered,this,&loginwin::on_enter_watchEye);

connect(this->watchEye,&HoverableLabel::mouseLeft,this,&loginwin::on_leave_watchEye);
```

当登录的账号密码输入结束后，可以手动点击登录按键登录系统，也可以使用按键 enter 的方式登录系统，这里使用事件处理的方式，将事件触发控制在 QLineEdit 内部触发时将事件拦截并响应，具体实现代码结果如下：

```
void loginwin::keyPressEvent(QKeyEvent *event)
{
    /*
     * 事件处理，优先于信号槽处理，且事件处理方式为，从被触发处开始进行链式向上父组件传递的处理方式进行处理
     * 这里由于 lineEdit 捕捉了字符的事件处理，因此无法被上层窗体的事件处理处理，而 enter 和 return 不被 lineEdit 捕捉
     * 传递到父组件这里被处理
     */
    if(event->key() == Qt::Key_Enter || event->key() == Qt::Key_Return)
        on_login_pbtn_clicked();
}
```

当用户点击登录、注册和 EyeWatch 查看时，登录窗口将做出对应的响应，其具体响应代码如下：

```
void loginwin::on_register_pbtn_clicked()
{
    this->hide();
    re.show();
}

void loginwin::after_register_login()
{
```

```
    re.hide();
    this->show();
}

void loginwin::after_control_win()
{
    control_win.hide();
    this->show();
}

void loginwin::on_enter_watchEye()
{
    // 显示密码
    ui->pwd_lineEdit->setEchoMode(QLineEdit::Normal);
}

void loginwin::on_leave_watchEye()
{
    // 隐藏密码（显示为星号或点）
    ui->pwd_lineEdit->setEchoMode(QLineEdit::Password);
}
```

当用户点击登录时，登录窗口将对输入内容与数据库中内容进行对比，只有当用户输入的账号和密码与数据库中的账号密码一致时，该用户才会登录成功。当登录出现失败时，系统将给出对应的登陆失败提示信息。具体实现登录功能的代码如下：

```
void loginwin::on_login_pbtn_clicked()
{
    /*
     * 连接数据库，登录正确和错误提示
     */
    QString username = ui->username_lineEdit->text();
    QString pwd = ui->pwd_lineEdit->text();

    int next_flage = 1;

    if(username == "" || pwd == "")
    {
        QMessageBox::warning(this,"警告","账号或密码不能为空!");
```

```
        next_flage = 0;
    }

    if(next_flage == 1)
    {
        QSqlQuery query(*DB);
        QString sql = QString("select * from logininfo "
                              "where username = '%1' and pwd = '%2';"
                              ).arg(username).arg(pwd);

        /*
         * query 执行 select 都返回 true，不存在 false【只有 update，insert 这些可能 false】
         */
        query.exec(sql);
        /*
         * while(query.next()) // query 的结果初始为 head 空数据，next 后才有第一个有效数据
         * qDebug() << "databse: " << query.value(0).toString() << " " << query.value(1).toString();
         * qDebug() << "size: " << query.size(); // query 查询没有数据时，数值为-1，否则为查询到的数据条目数
         */
        if(query.next())
        {
            QMessageBox::information(this,"提示","登录成功！");
            // 打开主用户窗体
            control_win.set_user_name(username);
            control_win.triggered();
            control_win.show();

            this->hide();
            next_flage = 0;
        }

        if(next_flage == 1)
        {
            sql = QString("select * from logininfo "
```

```
                                    "where username = '%1';"
                                    ).arg(username);
                query.exec(sql);
                if(query.next())
                    QMessageBox::warning(this,"警告","密码输入错误");
                else
                    QMessageBox::critical(this,"错误","登陆失败，请先注册后再登
录!");
            }
        }

        ui->username_lineEdit->clear();
        ui->pwd_lineEdit->clear();
}
```

该功能程序运行结果如图 5.22 所示。

图 5. 22 登录功能实现

当登录成功时，系统会给出对应登录成功的对话提示框。程序运行结果如图 5.23 所示。

图 5.23 登录成功

当登录失败时，系统会给出对应登录失败的对话框并给出对应的登录提示。程序运行结果如图 5.24 所示。

图 5.24 登录失败

5.4.2 注册功能实现

在注册界面设计的基础上，使用代码的方式对界面再次进行二次构造。将登录窗口固定大小为 512*527 的大小，并默认窗口名称为“用户注册”，最后对窗口的功能按键设置信号槽连接。

注册窗口的构造函数中的代码如下：

```
QPixmap pixmap("D:/MyDesktop/Graduation/ME/CommunityCanteenSys/Image/Background_PICS/register.jpg");//设定图片
    pixmap = MenuAlgorithm::PixmapToRound(pixmap,0,this->width(),this->height());
    QPalette palette;//创建一个调色板对象
    palette.setBrush(this->backgroundRole(),QBrush(pixmap));//用调色板的画笔把映射到 pixmap 上的图片画到 frame.backgroundRole()这个背景上
    this->setPalette(palette);//设置窗口调色板为 palette，窗口和画笔相关联

    watchEye = new HoverableLabel(this);
    QImage *image = new QImage("D:/MyDesktop/Graduation/ME/CommunityCanteenSys/Image/OrderDetail_ICONs/watchDetail.png");
    image = new QImage(image->scaled(30, 30, Qt::IgnoreAspectRatio, Qt::SmoothTransformation));
    watchEye->setPixmap(QPixmap::fromImage(*image));
    watchEye->move(414, 277);
    watchEye->setMinimumSize(30,30);
    watchEye->setStyleSheet(
                "QLabel{background-color: rgb(178,180,164,40%); border-radius: 15px;border: 1px;}"
                "QLabel:hover{background-color: gray; border-radius: 15px;border: 1px;}"
                );

    this->setFixedSize(512,527);
    this->setWindowTitle("用户注册");

connect(this->watchEye,&HoverableLabel::mouseEntered,this,&registerwin::on_enter_watchEye);

connect(this->watchEye,&HoverableLabel::mouseLeft,this,&registerwin::on_leave_watchEye);
```

当注册的账号密码输入结束后，可以手动点击注册按键注册用户，也可以使用按键 enter 的方式注册用户，这里使用事件处理的方式，将事件触发控制在 QLineEdit 内部触发时将事件拦截并响应，具体实现代码结果如下：

```
void registerwin::keyPressEvent(QKeyEvent *event)
{
    if(event->key() == Qt::Key_Enter || event->key() == Qt::Key_Return)
        on_register_pbtn_clicked();
}
```

当用户点击注册、返回登录和 EyeWatch 查看时，注册窗口将做出对应的响应，其具体响应代码如下：

```
void registerwin::on_enter_watchEye()
{
    // 显示密码
    ui->pwd_lineEdit->setEchoMode(QLineEdit::Normal);
}

void registerwin::on_leave_watchEye()
{
    // 隐藏密码（显示为星号或点）
    ui->pwd_lineEdit->setEchoMode(QLineEdit::Password);
}
void registerwin::on_pushButton_clicked()
{
    this->hide();
    emit register_complete();
}
```

当用户点击注册时，注册窗口将对输入内容与数据库中内容进行对比，只有当用户输入的账号与数据库中的账号不一致时，该用户才会注册成功。当注册出现失败时，系统将给出对应的注册失败提示信息。具体实现注册功能的代码如下：

```
void registerwin::on_register_pbtn_clicked()
{
    // 判断注册是否成功
    QString username = ui->username_lineEdit->text();
    QString pwd = ui->pwd_lineEdit->text();

    int next_flage = 1;

    if(username == "" || pwd == "")
    {
        QMessageBox::warning(this,"警告","账号或密码不能为空!");
        next_flage = 0;
```

```
    }

    if(next_flage == 1)
    {
        QSqlQuery query(*DB);
        QString sql = QString("select * from logininfo "
                              "where username = '%1';"
                              ).arg(username);

        query.exec(sql);

        if(query.next())
        {
            QMessageBox::warning(this,"警告","注册失败，该用户已存在，请更换用户名");
        }
        else
        {
            sql = QString("select * from logininfo order by curid desc;");
            query.exec(sql);
            query.next();

            int curid = query.size() == 0 ? 0 : query.value("curid").toInt() + 1;

            sql = QString("insert into logininfo values "
                          "('%1', '%2', '%3');"
                          ).arg(username).arg(pwd).arg(curid);

            if(query.exec(sql))
            {
                if(QMessageBox::information(this,"提示","注册成功！","返回登陆","继续注册",0,1) == 0)
                    on_pushButton_clicked();
            }
            else
            {
                QMessageBox::critical(this,"警告","数据插入失败！");
            }
```

```
            }
        }
        ui->username_lineEdit->clear();
        ui->pwd_lineEdit->clear();
}
```

注册功能的程序运行结果如图 5.25 所示。

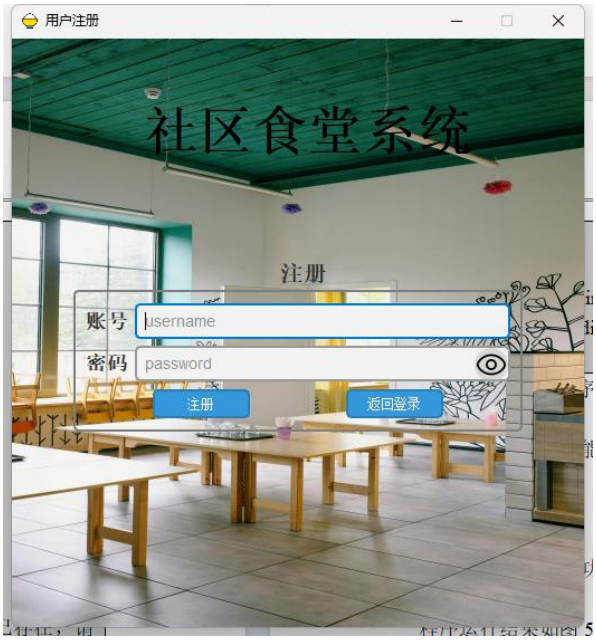

图 5.25 注册功能实现

当注册成功时，系统会给出对应注册成功的对话提示框。程序运行结果如图 5.26 所示。

图 5.26 注册成功

当注册失败时，系统会给出对应注册失败的对话框并给出对应的注册提示。程序运行结果如图 5.27 所示。

图 5.27 注册失败

5.4.3 用户头像功能实现

用户头像功能的主要界面设计在 Qt Designer 中完成。用户头像的逻辑功能是当用户点击头像时，该功能触发对应的鼠标点击事件并触发头像选择功能。具体实现头像功能的代码如下：

```
void UserIconLabel::mousePressEvent(QMouseEvent *ev)
{
    if(ev->button() == Qt::LeftButton)
    {
        QString userIconPath = QFileDialog::getOpenFileName(Q_NULLPTR,

QObject::tr("Select UserIcon"),
                                                             QObject::tr("../"),

QObject::tr("File(*.jpg);;File(*.png);;File(*.ico);;"

"All(*.*)"));
        if(userIconPath.size() > 0) emit back(userIconPath);
```

```
    }

}
```

用户头像功能的程序运行结果如图 5.28 所示。

图 5.28 用户头像展示

5.4.4 公告轮播图功能实现

公告轮播图的界面大小主要取决于其上层父窗体的窗体大小，当窗体产生后，公告栏将公告轮播图数据初始化自身。最后将自身功能与信号槽进行链接。具体代码实现如下：

```
    this->setGeometry(0,0,parent->size().width(),parent->size().height());

    carouseInitFromDB();
    init_btns();


connect(leftBtn,&QPushButton::clicked,this,&CarouselChart::on_leftBtn_clicked);

connect(rightBtn,&QPushButton::clicked,this,&CarouselChart::on_rightBtn_clicked);

connect(&animation_group,&QParallelAnimationGroup::stateChanged,this,&Carous
elChart::on_animationState);
```

公告栏默认在 5s 后自动触发公告移动，如果通过左右按钮触发公告移动切换时，则使用方向移动触发按钮。具体实现代码如下：

```
void CarouselChart::timerEvent(QTimerEvent *ev)
```

```
{
    if(animationIsRunning)
        return;
    imageOrderMove(DIRECTION_RIGHT);
}

void CarouselChart::on_leftBtn_clicked()
{
    if(animationIsRunning)
        return;
    resetTimer();
    imageOrderMove(DIRECTION_LEFT);
}

void CarouselChart::on_rightBtn_clicked()
{
    if(animationIsRunning)
        return;
    resetTimer();
    imageOrderMove(DIRECTION_RIGHT);
}
```

公告栏的移动动画产生的控制代码结果如下：

```
void CarouselChart::imageOrderMove(CarouselChart::DIRECTION direction)
{
    old_index = cur_index;

    int order = 0;
    if(direction == DIRECTION::DIRECTION_LEFT)
        order = -1;
    else if(direction == DIRECTION::DIRECTION_RIGHT)
        order = 1;
    cur_index = (cur_index + order + labels.size()) % labels.size();

    // 动画的效果有“pos” 、“opacity ”、“geometry”，“sacleFactor”，分别控制位置、透明度、形状，大小;
    QPropertyAnimation *animation_old = new QPropertyAnimation(labels[old_index], "geometry",this); // 给 label 添加动画【动画只是一个效果，附加在空间上】
```

```
    animation_old->setDuration(Pic_Smooth_time); // 动画的持续时间
    animation_old->setLoopCount(1);
    QPropertyAnimation *animation_new = new QPropertyAnimation(labels[cur_index], "geometry",this);
    animation_new->setDuration(Pic_Smooth_time);
    animation_new->setLoopCount(1);

    // 【点击左侧按钮，想看左侧的图片，图片动画应该是向右滑动】
    if(direction == DIRECTION::DIRECTION_LEFT)
    {
        // 设置动画的开始位置和结束位置
        // value:它是一个 QVariant 类型，QPoint 对应 pos，QRect 对应 geometry，单个数值对应 opacity 或者 sacleFactor

animation_old->setStartValue(QRect(0,0,this->size().width(),this->size().height()));

animation_old->setEndValue(QRect(this->size().width(),0,this->size().width(),this->size().height()));


animation_new->setStartValue(QRect(-this->size().width(),0,this->size().width(),this->size().height()));

animation_new->setEndValue(QRect(0,0,this->size().width(),this->size().height()));
    }
    else if(direction == DIRECTION::DIRECTION_RIGHT)
    {

animation_old->setStartValue(QRect(0,0,this->size().width(),this->size().height()));

animation_old->setEndValue(QRect(-this->size().width(),0,this->size().width(),this->size().height()));


animation_new->setStartValue(QRect(this->size().width(),0,this->size().width(),this->size().height()));

animation_new->setEndValue(QRect(0,0,this->size().width(),this->size().height()));
```

```
    }

    labels[cur_index]->show();

    // 将设置好的两个控件的动画加入并行动画组，让他们一起展示
    animation_group.addAnimation(animation_old);
    animation_group.addAnimation(animation_new);

    animation_group.setDirection(QAbstractAnimation::Forward);
    animation_group.start(); // 开始并行执行加载的动画
}
```

系统公告轮播图的程序运行结果见图 5.29 所示。

图 5.29 系统公告轮播图实现

5.4.5 菜品浏览功能实现

菜品浏览功能实现主要依赖底层逻辑代码对数据进行控制后结合菜品展示器将菜品结果展示，具体实现代码如下：

```
void DishTurnPageBar::dishInitfromDB()
{
    QSqlQuery query(*DB);
    QString sql = QString("select * from dishesInfo order by DishIndex asc;");
    query.exec(sql);

    DishShowBar *t = nullptr;
    while(query.next())
    {
        t = new DishShowBar(controlwin_username,query.value("DishImagePath").toString(),
                            query.value("DishName").toString(),
                            query.value("DishMoney").toString(),
                            query.value("DishNum").toString(),
```

```
query.value("DishIndex").toString(),this->scrollAreaContents);

        // 同一个类型的对象，发送相同的信号，但不同的对象，选择性的对信号进行链接

connect(t,&DishShowBar::deleteSuccess,this,&DishTurnPageBar::slot_deleteDishShowBarSuccess);

            dishManager.addDish(t);
        }
        t = new DishShowBar(controlwin_username,"","请添加菜品",
                                "请添加价格","请添加数量","-1",this->scrollAreaContents);

connect(t,&DishShowBar::addSuccess,this,&DishTurnPageBar::slot_addDishShowBarSuccess);

        dishManager.addDish(t);
        t = nullptr;
}

void DishTurnPageBar::dishShow()
{
        for(int i = 0;i < dishManager.sizeDish();i++) dishManager[i]->hide();
//        qDebug() << adminFlag;
        int validDishSize = **controlwin_username == "admin" ?
                            dishManager.sizeDish() : dishManager.sizeDish() - 1;

        int i = curIndex * pageShowNum;
        int margin = (curIndex + 1) * pageShowNum;
        int valid_margin = (validDishSize < margin ? validDishSize : margin);

        for(;i < valid_margin;i++)
        {
            dishManager[i]->move(x + ((i % pageShowNum) % 3) * 220, y + (((i % pageShowNum)) / 3) * 290);
            dishManager[i]->show();
```

```
    }

    this->move(170,y + ((((--i % pageShowNum)) / 3) + 1) * 290 + 20);
    scrollArea->verticalScrollBar()->setValue(0);

}
```

当控制菜品分页器进行左右分页展示菜品时，或者直接更改菜品当前页面数字时，将直接对菜品展示页面的内容进行更换。具体实现代码如下：

```
int DishTurnPageBar::getCurIndex()
{
    return curIndex;
}

void DishTurnPageBar::setCurIndex(int index)
{
    curIndex = index;
    ui->pageIndex_lineEdit->setText(QString::number(index));
}

void DishTurnPageBar::on_curIndex_lineEdit_textEdited(const QString &arg1)
{
    // 预判
    int t_index = arg1.toInt();
    if(t_index < 1 || t_index > (dishManager.sizeDish() + pageShowNum - 1) /
pageShowNum)
    {
        ui->pageIndex_lineEdit->setText(QString::number(curIndex + 1));
        return;
    }

    curIndex = t_index - 1;
    // 展示
    dishShow();
}

void DishTurnPageBar::on_left_ptn_clicked()
{
    // 预判
```

```
    curIndex = curIndex > 0 ? curIndex - 1 : 0;

    // 展示
    dishShow();

    ui->pageIndex_lineEdit->setText(QString::number(curIndex + 1));
}

void DishTurnPageBar::on_right_ptn_clicked()
{
    // 预判
    int valid_size = **controlwin_username == "admin" ?
                dishManager.sizeDish() : dishManager.sizeDish() - 1;

    curIndex = curIndex <   (valid_size + pageShowNum - 1) / pageShowNum - 1 ?
                curIndex + 1 : (valid_size + pageShowNum - 1) / pageShowNum
- 1;

    // 展示
    dishShow();

    ui->pageIndex_lineEdit->setText(QString::number(curIndex + 1));

}

void DishTurnPageBar::on_pageIndex_lineEdit_textChanged(const QString &arg1)
{
    // 预判
    int t_index = arg1.toInt();
    if(t_index < 1 || t_index > (dishManager.sizeDish() + pageShowNum - 1) /
pageShowNum)
    {
        ui->pageIndex_lineEdit->setText(QString::number(curIndex + 1));
        return;
    }

    curIndex = t_index - 1;
    // 展示
```

```
    dishShow();
}
```

当管理员对菜品进行删除或新增菜品时，菜品分页器内所管理的所有菜品展示器内容也需要发生对应的排列变化，从而影响菜品展示内容的顺序。其具体代码实现如下：

```
void DishTurnPageBar::slot_addDishShowBarSuccess(DishShowBar* self)
{

disconnect(self,&DishShowBar::addSuccess,this,&DishTurnPageBar::slot_addDishShowBarSuccess);

connect(self,&DishShowBar::deleteSuccess,this,&DishTurnPageBar::slot_deleteDishShowBarSuccess);

    DishShowBar *t = new DishShowBar(controlwin_username,"","请添加菜品",
                                     "请添加价格","请添加数量","-1",this->scrollAreaContents);

connect(t,&DishShowBar::addSuccess,this,&DishTurnPageBar::slot_addDishShowBarSuccess);

    dishManager.addDish(t);
    t = nullptr;
    dishShow();
}

void DishTurnPageBar::slot_deleteDishShowBarSuccess(DishShowBar* self)
{
    dishManager.deleteDish(*self->getDishName());
    dishShow();
}
```

菜品分页器的程序运行结果见图 5.30 所示。

图 5.30 菜品分页器实现

5.4.6 购物车功能实现

购物车功能在 Qt Designer 中实现初步设计后，在其构造时将再一步对其进行详细设计。具体实现代码如下：

```
    ui->setupUi(this);
    this->setFixedSize(this->size());

    QPixmap
pixmap("D:/MyDesktop/Graduation/ME/CommunityCanteenSys/Image/Background_
PICS/buycar.jpg");//设定图片

//      QPainter p1(&pixmap);
//      p1.setCompositionMode(QPainter::CompositionMode_Source);
//      p1.drawPixmap(0, 0, pixmap);
//      p1.setCompositionMode(QPainter::CompositionMode_DestinationIn);
```

```
//      //根据 QColor 中第四个参数设置透明度，0～255
//      p1.fillRect(pixmap.rect(), QColor(179,179,164, 220));
//      p1.end();

    pixmap = MenuAlgorithm::PixmapToRound(pixmap,0,this->width(),this->height());
    QPalette palette;//创建一个调色板对象
    palette.setBrush(this->backgroundRole(),QBrush(pixmap));//用调色板的画笔把映射到 pixmap 上的图片画到 frame.backgroundRole()这个背景上
    this->setPalette(palette);//设置窗口调色板为 palette，窗口和画笔相关联

    ui->scrollArea->setStyleSheet("QScrollArea {background-color:transparent;}");
    ui->scrollArea->viewport()->setStyleSheet("QScrollArea {background-color:transparent;}");

    // 设置垂直滚动条样式
    QString verticalScrollBarStyle =
        "QScrollBar:vertical {"
            "    border: none;"
            "    background: transparent;"
            "    width: 12px;"
            "    margin: 16px 0 16px 0;"
            "}"
            "QScrollBar:vertical:hover {"
            "    background: #b3d9ff;"
            "}"
            "QScrollBar::handle:vertical {"
            "    background: #5dade2;"
            "    border-radius: 6px;"
            "    min-height: 30px;"
            "}"
            "QScrollBar::handle:vertical:hover {"
            "    background: #2e86c1;"
            "}"
            "QScrollBar::add-line:vertical {"
            "    height: 16px;"
            "    subcontrol-position: bottom;"
```

```
            "      subcontrol-origin: margin;"
            "}"
            "QScrollBar::sub-line:vertical {"
            "      height: 16px;"
            "      subcontrol-position: top;"
            "      subcontrol-origin: margin;"
            "}";

ui->scrollArea->verticalScrollBar()->setStyleSheet(verticalScrollBarStyle);

// 设置水平滚动条样式
QString horizontalScrollBarStyle =
      "QScrollBar:horizontal {"
            "      border: none;"
            "      background: transparent;"
            "      height: 12px;"
            "      margin: 0 16px 0 16px;"
            "}"
            "QScrollBar:horizontal:hover {"
            "      background: #b3d9ff;"
            "}"
            "QScrollBar::handle:horizontal {"
            "      background: #5dade2;"
            "      border-radius: 6px;"
            "      min-width: 30px;"
            "}"
            "QScrollBar::handle:horizontal:hover {"
            "      background: #2e86c1;"
            "}"
            "QScrollBar::add-line:horizontal {"
            "      width: 16px;"
            "      subcontrol-position: right;"
            "      subcontrol-origin: margin;"
            "}"
            "QScrollBar::sub-line:horizontal {"
            "      width: 16;"
            "      subcontrol-position: left;"
            "      subcontrol-origin: margin;"
```

```
                "}";

    ui->scrollArea->horizontalScrollBar()->setStyleSheet(horizontalScrollBarStyle);

    this->setWindowTitle(control_username + "的购物车");

    QImage *image = new QImage("D:/MyDesktop/Graduation/ME/CommunityCanteenSys/Image/BuyCar_ICONs/BuyCarIcon.png");
    image = new QImage(image->scaled(81, 81, Qt::IgnoreAspectRatio, Qt::SmoothTransformation));
    ui->buyCarIcon_lab->setPixmap(QPixmap::fromImage(*image));

    ui->allPay_lab->setText("总计金额：0 元");
    ui->allSelect_ckbox->setTristate(true);
    ui->allSelect_ckbox->setCheckState(Qt::Unchecked);

    buyCarDishInitFromDB();
    showBuyCarDish();
```

购物车中当用户对菜品进行动态选择时，其菜品的总金额也将动态变化。具体实现代码如下：

```
void buycarwin::refreshAllSelect_Allpay()
{
    int allPay = 0;
    bool checkedFlag = false;
    bool uncheckedFlag = false;

    for(int i = 0;i < buyCarDishes.size();i++)
    {
        if(buyCarDishes[i]->getSelectCheckBox()->isChecked())
        {
            checkedFlag = true;
            allPay += MenuAlgorithm::GetQStringByFirstNum(buyCarDishes[i]->getAllMoney());
        }
        else uncheckedFlag = true;
    }
```

```
    // 设置 allPay
    ui->allPay_lab->setText("总计金额：" + QString::number(allPay) + "元");

    // 设置 allSelect
    ui->allSelect_ckbox->setTristate(true);
    if(checkedFlag && uncheckedFlag)
ui->allSelect_ckbox->setCheckState(Qt::PartiallyChecked);
    else if(checkedFlag && !uncheckedFlag)
ui->allSelect_ckbox->setCheckState(Qt::Checked);
    else ui->allSelect_ckbox->setCheckState(Qt::Unchecked);
}

void buycarwin::on_allSelect_ckbox_clicked()
{
    int allPay = 0;
    ui->allSelect_ckbox->setTristate(false);

    Qt::CheckState checkStatus = ui->allSelect_ckbox->checkState();

    for(int i = 0;i < buyCarDishes.size();i++)
    {
        buyCarDishes[i]->getSelectCheckBox()->setCheckState(checkStatus);
        if(checkStatus == Qt::Checked)
            allPay +=
MenuAlgorithm::GetQStringByFirstNum(buyCarDishes[i]->getAllMoney());
    }

    // 设置 allPay
    ui->allPay_lab->setText("总计金额：" + QString::number(allPay) + "元");
}
```

用户在购物车中当点击删除菜品、购买按钮和取消按钮时，将对选中菜品进行对应的操作。其具体实现代码如下：

```
void buycarwin::on_deleteSelect_ptn_clicked()
{
    bool flag = false;
    QString buyCarTable = control_username + "BuyCar";

    for(int i = 0;i < buyCarDishes.size();)
```

```
    {
        if(buyCarDishes[i]->getSelectCheckBox()->isChecked())
        {
            // DB 删除
            QSqlQuery query(*DB);
            QString sql = QString("delete from %1 where DishName = '%2';")
                    .arg(buyCarTable)
                    .arg(buyCarDishes[i]->getDishName());

            query.exec(sql);

            if(!flag) flag = true;
            delete buyCarDishes[i];
            buyCarDishes.remove(i);
        }
        else i++;
    }

    if(flag)
    {
        ui->allPay_lab->setText("总计金额：0 元");
        ui->allSelect_ckbox->setCheckState(Qt::Unchecked);

        showBuyCarDish();
    }
}

void buycarwin::on_cancel_ptn_clicked()
{
    this->close();
}

void buycarwin::on_buy_ptn_clicked()
{
    if(buyCarDishes.size() == 0)
    {
        QMessageBox::warning(nullptr,"购买警告","请先向购物车中添加菜品，
再进行购买！");
```

```
        return;
    }

    QString text = "确认购买，%1";
    text = text.arg(ui->allPay_lab->text());
    int check = QMessageBox::information(nullptr," 购 物 车 购 买 信 息 ",text,QMessageBox::Ok | QMessageBox::Close);

    if(check == QMessageBox::Ok)
    {
        // 付款界面
        QImage *paypal = new QImage("D:/MyDesktop/Graduation/ME/CommunityCanteenSys/Image/PayPal/Paying.jpg");
        QDialog *w = new QDialog;
        QVBoxLayout *layout = new QVBoxLayout; // 在布局中添加控件，最后将布局添加到窗体中
        QLabel *label = new QLabel();
        label->setPixmap(QPixmap::fromImage(*paypal));
        layout->addWidget(label);
        w->setLayout(layout);
        w->move(0,0);
        w->exec();

        QString orderDetailTable = control_username + "OrderDetail";

        QSqlQuery query(*DB);
        // 获取订单创建时间
        QString sql = QString("select now();");
        query.exec(sql);
        query.next();
        QString nowTime = query.value(0).toString();

        for(int i = 0;i < buyCarDishes.size();i++)
        {
            if(buyCarDishes[i]->getSelectCheckBox()->isChecked())
            {
                sql = QString("insert into %1 values"
```

```
                                        "('%2','%3','%4','%5','%6')")
                                .arg(orderDetailTable)
                                .arg(buyCarDishes[i]->getDishName())
                                .arg(buyCarDishes[i]->getDishMoney())
                                .arg(buyCarDishes[i]->getDishBuyNum())
                                .arg(buyCarDishes[i]->getImagePath())
                                .arg(nowTime);
                        query.exec(sql);
                    }
                }

                on_deleteSelect_ptn_clicked();
                QMessageBox::information(nullptr,"购买成功","购买成功！ ");
            }
}
```

用户购物车程序运行结果见图 5.31 所示。

图 5. 31 用户购物车实现

用户购物车实现删除功能运行过程见图 5.32 和 5.33 所示。

图 5.32 选中删除菜品

图 5.33 删除菜品实现

用户购物车实现菜品购买功能的程序运行结果见图 5.34 所示。

图 5.34 菜品购买实现

5.4.7 订单详情功能实现

用户订单详情功能在 Qt Designer 中实现初步设计后，在订单详情产生时将读取用户订单详情数据并再进一步进行详细设计。具体代码实现结果如下：

```
ui->setupUi(this);
    this->setFixedSize(this->size());

    QPixmap pixmap("D:/MyDesktop/Graduation/ME/CommunityCanteenSys/Image/Background_PICS/orderDetail.png");//设定图片

//    QPainter p1(&pixmap);
//    p1.setCompositionMode(QPainter::CompositionMode_Source);
//    p1.drawPixmap(0, 0, pixmap);
//    p1.setCompositionMode(QPainter::CompositionMode_DestinationIn);

//    //根据 QColor 中第四个参数设置透明度，0～255
//    p1.fillRect(pixmap.rect(), QColor(179,179,164, 220));
//    p1.end();
```

```
    pixmap = MenuAlgorithm::PixmapToRound(pixmap,0,this->width(),this->height());
    QPalette palette;//创建一个调色板对象
    palette.setBrush(this->backgroundRole(),QBrush(pixmap));//用调色板的画笔把映射到 pixmap 上的图片画到 frame.backgroundRole()这个背景上
    this->setPalette(palette);//设置窗口调色板为 palette，窗口和画笔相关联

    ui->scrollArea->setStyleSheet("QScrollArea {background-color:transparent;}");
    ui->scrollArea->viewport()->setStyleSheet("QScrollArea {background-color:transparent;}");

    // 设置垂直滚动条样式
    QString verticalScrollBarStyle =
        "QScrollBar:vertical {"
            "    border: none;"
            "    background: transparent;"
            "    width: 12px;"
            "    margin: 16px 0 16px 0;"
            "}"
            "QScrollBar:vertical:hover {"
            "    background: #b3d9ff;"
            "}"
            "QScrollBar::handle:vertical {"
            "    background: #5dade2;"
            "    border-radius: 6px;"
            "    min-height: 30px;"
            "}"
            "QScrollBar::handle:vertical:hover {"
            "    background: #2e86c1;"
            "}"
            "QScrollBar::add-line:vertical {"
            "    height: 16px;"
            "    subcontrol-position: bottom;"
            "    subcontrol-origin: margin;"
            "}"
            "QScrollBar::sub-line:vertical {"
            "    height: 16px;"
```

```
            "      subcontrol-position: top;"
            "      subcontrol-origin: margin;"
            "}";

ui->scrollArea->verticalScrollBar()->setStyleSheet(verticalScrollBarStyle);

// 设置水平滚动条样式
QString horizontalScrollBarStyle =
      "QScrollBar:horizontal {"
            "      border: none;"
            "      background: transparent;"
            "      height: 12px;"
            "      margin: 0 16px 0 16px;"
            "}"
            "QScrollBar:horizontal:hover {"
            "      background: #b3d9ff;"
            "}"
            "QScrollBar::handle:horizontal {"
            "      background: #5dade2;"
            "      border-radius: 6px;"
            "      min-width: 30px;"
            "}"
            "QScrollBar::handle:horizontal:hover {"
            "      background: #2e86c1;"
            "}"
            "QScrollBar::add-line:horizontal {"
            "      width: 16px;"
            "      subcontrol-position: right;"
            "      subcontrol-origin: margin;"
            "}"
            "QScrollBar::sub-line:horizontal {"
            "      width: 16;"
            "      subcontrol-position: left;"
            "      subcontrol-origin: margin;"
            "}";

ui->scrollArea->horizontalScrollBar()->setStyleSheet(horizontalScrollBarStyle);
```

```
    this->setWindowTitle(control_username + "的订单详情");

    QImage *image = new QImage("D:/MyDesktop/Graduation/ME/CommunityCanteenSys/Image/OrderDetail_ICONs/oderDetailIcon.png");
    image = new QImage(image->scaled(61, 61, Qt::IgnoreAspectRatio, Qt::SmoothTransformation));
    ui->orderIcon_lab->setPixmap(QPixmap::fromImage(*image));

    dishDetailInitFromDB();
    showDishDetail();
```

当用户在订单详情中进行删除订单时，历史订单将被永久删除。具体实现代码如下：

```
void orderdetailwin::deleteDishDetail(OderDetailBar *t)
{
    QString orderDetailTable = control_username + "OrderDetail";

    for(int i = 0;i < orderDetails.size();i++)
    {
        if(orderDetails[i] == t)
        {
            QSqlQuery query(*DB);
            QString sql = QString("delete from %1 "
                                  "where TakeOrderTime = '%2';")
                    .arg(orderDetailTable)
                    .arg(t->getTakeOrderTime());

            if(!query.exec(sql)) // 重复删除不会报错！【不能产生多个窗口同时控制】
            {
                QMessageBox::critical(nullptr,"删除失败","删除订单详情失败");
                return;
            }

            if(orderDetails[i]) delete orderDetails[i];
            orderDetails.remove(i);
            showDishDetail();
```

```
                break;
            }
        }
}
```

订单详情程序运行结果见图 5.35 所示。

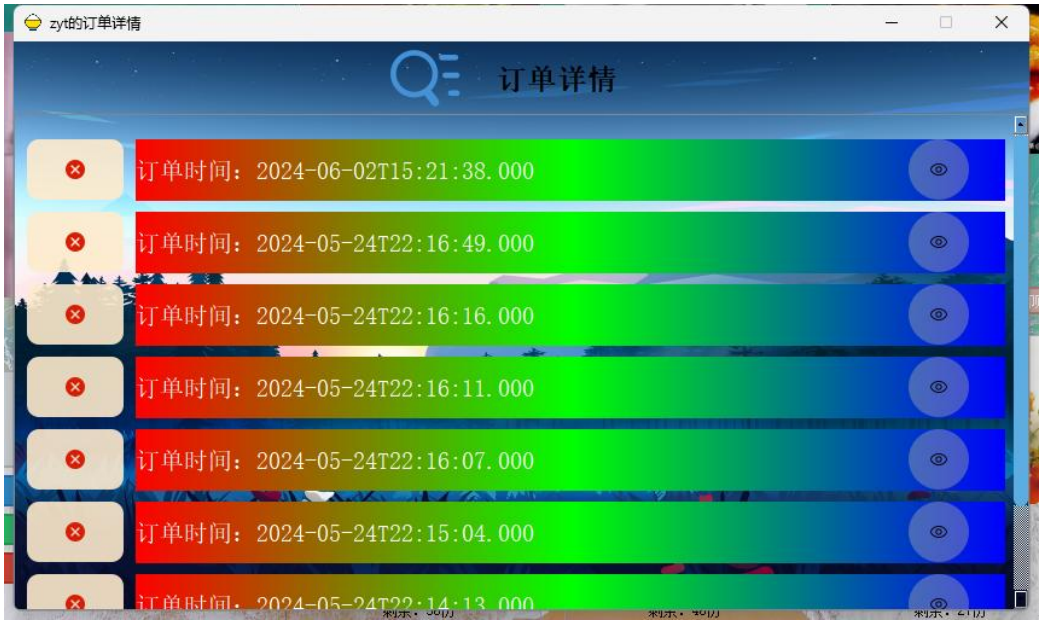

图 5.35 订单详情实现

当用户在订单详情页面中对订单进行进一步查看，点击查看订单详情时，将触发生成订单菜品页面。具体实现代码如下：

```
void OderDetailBar::on_detail_ptn_clicked()
{
    DetailBar *detailBar = new DetailBar(ui->orderTime_lab->text()
                                         ,controlwin_username,takeOrderTi
me);
    detailBar->setAttribute(Qt::WA_DeleteOnClose);
    detailBar->setWindowModality(Qt::ApplicationModal);
    detailBar->show();

}
```

订单菜品程序运行结果见图 5.36 所示。

图 5.36 订单菜品实现

5.4.8 用户管理功能实现

用户管理功能主要对社区食堂系统中的所有用户进行账户管理，其主要功能实现依赖上层用户操作与用户数据库之间的信息交互。

持久化池功能主要实现上层用户对账户信息的修改内容后，将内容同步到数据库中永久保存。其具体实现代码如下

```
void adminwin::on_save_btn_clicked()
{
    ui->progress_Bar->setMaximum(3);
    ui->progress_Bar->setValue(0);
    ui->process_Details->setText(QString("任务执行中..."));
    int curRow = 0;
    /*
     * curRow = valid[0,...] invalid[-1]
    */
    curRow = ui->detailsTable->currentRow();
    //qDebug() << curRow;

    if(curRow != -1)
    {
```

```
        if(ui->detailsTable->item(curRow,0)->text() == "" ||
ui->detailsTable->item(curRow,1)->text() == "")
        {
            ui->progress_Bar->setValue(0);
            ui->process_Details->setText(QString("Error:添加失败，当前行存在
空数据..."));
            return;
        }

        QSqlQuery query(*DB);
        ui->progress_Bar->setValue(1);

        QString order = QString("update logininfo set username = '%1', "
                                "pwd = '%2' "
                                "where curid = '%3';")
                .arg(ui->detailsTable->item(curRow,0)->text())
                .arg(ui->detailsTable->item(curRow,1)->text())
                .arg(curRow);
        ui->progress_Bar->setValue(2);

        if(query.exec(order))
        {
            ui->progress_Bar->setValue(3);
            ui->process_Details->setText(QString("更新成功..."));
        }
        else
        {
            ui->progress_Bar->setValue(3);
            ui->process_Details->setText(QString("更新失败..."));
        }

    }
    else
    {
        ui->progress_Bar->setValue(0);
        if(loadDataFlag)  ui->process_Details->setText(QString("Warning: 请 选 中
需要保存的数据行，或先添加数据..."));
        else ui->process_Details->setText(QString("Warning:请先加载数据..."));
```

```
    }

}
void adminwin::on_save_all_btn_clicked()
{
    int row = ui->detailsTable->rowCount();
    QString order = "";
    QSqlQuery query(*DB);

    if(row > 0)
    {
        ui->progress_Bar->setMaximum(row);
        ui->progress_Bar->setValue(0);
        ui->process_Details->setText(QString("任务执行中..."));

        for (int i = 0; i < row; i++)
        {
            ui->progress_Bar->setValue(i + 1);
            order = QString("update logininfo set username = '%1', "
                                        "pwd = '%2' "
                                        "where curid = '%3';")
                    .arg(ui->detailsTable->item(i,0)->text())
                    .arg(ui->detailsTable->item(i,1)->text())
                    .arg(i);
            query.exec(order);
        }

        ui->process_Details->setText(QString("完成"));
    }
    else
    {
        ui->progress_Bar->setValue(0);
        if(loadDataFlag)  ui->process_Details->setText(QString("Warning: 请 先 添
加数据再保存..."));
        else ui->process_Details->setText(QString("Warning:请先加载数据..."));
    }
}
```

修改功能池主要实现用户对读取到上层界面中数据的修改，用户层对数据的

修改结果不会同步到数据库中持久化。具体实现代码如下：

```
void adminwin::on_add_btn_clicked()
{
    ui->progress_Bar->setMaximum(3);
    ui->progress_Bar->setValue(0);
    ui->process_Details->setText(QString("任务执行中..."));
    int rowCnt = ui->detailsTable->rowCount();

    if(rowCnt != 0 || (loadDataFlag && rowCnt == 0))
    {
        QSqlQuery query(*DB);
        QString order = QString("insert into logininfo(username,"
                                "pwd,curid) "
                                "values('%1', '%2', '%3');")
                            .arg("")
                            .arg("")
                            .arg(rowCnt);
        ui->progress_Bar->setValue(1);

        bool ok = query.exec(order);
        ui->progress_Bar->setValue(2);
        ui->detailsTable->insertRow(rowCnt);
        ui->detailsTable->setItem(rowCnt,0,new QTableWidgetItem(""));
        ui->detailsTable->setItem(rowCnt,1,new QTableWidgetItem(""));

        if(ok)
        {
            ui->progress_Bar->setValue(3);
            ui->process_Details->setText(QString("加入新行成功..."));
        }
        else
        {
            ui->progress_Bar->setValue(0);
            ui->process_Details->setText(QString("加入新行失败..."));
        }
    }
    else
    {
```

```
            ui->progress_Bar->setValue(0);
            ui->process_Details->setText(QString("Warning:请先加载数据..."));
        }

}

void adminwin::on_insert_btn_clicked()
{
        ui->progress_Bar->setMaximum(3);
        ui->progress_Bar->setValue(0);
        ui->process_Details->setText(QString("任务执行中..."));
        int curRow = ui->detailsTable->currentRow();

        if(curRow != -1)
        {
            move_curRow_base(curRow - 1, 1);
            ui->progress_Bar->setValue(1);

            QSqlQuery query(*DB);
            QString order = QString("insert into logininfo(username,"
                                    "pwd,curid) "
                                    "values('%1', '%2', '%3');")
                                .arg("")
                                .arg("")
                                .arg(curRow);

            bool ok = query.exec(order);
            ui->progress_Bar->setValue(2);
            ui->detailsTable->insertRow(curRow);
            ui->detailsTable->setItem(curRow,0,new QTableWidgetItem(""));
            ui->detailsTable->setItem(curRow,1,new QTableWidgetItem(""));

            if(ok)
            {
                ui->progress_Bar->setValue(3);
                ui->process_Details->setText(QString("插入成功..."));
            }
            else
```

```
            {
                ui->progress_Bar->setValue(0);
                ui->process_Details->setText(QString("插入失败..."));
            }
        }
        else
        {
            ui->progress_Bar->setValue(0);
            if(loadDataFlag)  ui->process_Details->setText(QString("Warning: 请选中需要插入的数据行，或先添加数据..."));
            else ui->process_Details->setText(QString("Warning:请先加载数据..."));
        }
}

void adminwin::on_delete_btn_clicked()
{
    // insert 行和 blank 的插入数据
    ui->progress_Bar->setMaximum(5);
    ui->progress_Bar->setValue(0);
    ui->process_Details->setText(QString("任务执行中..."));

    int curRow = ui->detailsTable->currentRow();
    ui->progress_Bar->setValue(1);

    if(curRow != -1)
    {
        QString username = ui->detailsTable->item(curRow,0)->text();
        QString userpwd = ui->detailsTable->item(curRow,1)->text();
        if(username == "admin")
        {
            ui->progress_Bar->setValue(0);
            ui->process_Details->setText(QString("Error: 禁止删除管理员用户..."));
            return;
        }
        QSqlQuery query(*DB);
        QString sql = QString("delete from logininfo "
                              "where curid = '%1';")
```

```
                    .arg(curRow);
        bool del_ok = query.exec(sql);

        ui->progress_Bar->setValue(2);

        if(del_ok)
        {
            ui->progress_Bar->setValue(3);
            move_curRow_base(curRow,-1);
            ui->progress_Bar->setValue(4);

            ui->detailsTable->removeRow(curRow);

            // 同步数据删除
            // 删表
            sql = QString("drop table if exists %1BuyCar;")
                    .arg(username);
            query.exec(sql);
            sql = QString("drop table if exists %1OrderDetail;")
                    .arg(username);
            query.exec(sql);
            // 删除数据
            sql = QString("delete from username_usericonpath where username =
'%1';")
                    .arg(username);
            query.exec(sql);

            ui->progress_Bar->setValue(5);
            ui->process_Details->setText(QString("删除成功..."));
        }
        else
        {
            ui->progress_Bar->setValue(0);
            ui->process_Details->setText(QString("删除失败..."));
        }
    }
    else
    {
```

```
        ui->progress_Bar->setValue(0);
        if(loadDataFlag)  ui->process_Details->setText(QString("Warning: 请 选 中
需要删除的数据行，或先添加数据..."));
        else ui->process_Details->setText(QString("Warning:请先加载数据..."));

    }
}
```

查看功能池主要负责将数据库中的数据上传并展示到上层用户界面中。具体实现代码如下：

```
void adminwin::on_load_login_btn_clicked()
{
    if(loadDataFlag) on_clear_btn_clicked();
    ui->detailsTable->setColumnCount(2);
    ui->detailsTable->setHorizontalHeaderLabels({"username","pwd"});

    QSqlQuery query(*DB);

    query.prepare("select username,"
                  "pwd "
                  "from logininfo order by curid asc;");
    query.exec();

    int row = query.size();
    int column = ui->detailsTable->columnCount();

    /*
     * Bar 进度条的进度条规则为{0 起始}[1,len]为有效进度长度
    */
    ui->progress_Bar->setMaximum(row);
    ui->progress_Bar->setValue(0);
    ui->process_Details->setText(QString("任务执行中..."));

    for (int i = 0;i < row; i++)
    {
        ui->progress_Bar->setValue(i + 1);
        query.next();
        ui->detailsTable->insertRow(i);
        for (int j = 0; j < column; j++)
```

```
            {
                ui->detailsTable->setItem(i,j,new
QTableWidgetItem(query.value(j).toString()));

ui->detailsTable->item(i,j)->setTextAlignment(Qt::AlignVCenter|Qt::AlignHCenter);
            }
        }


ui->detailsTable->horizontalHeader()->setSectionResizeMode(QHeaderView::Resize
ToContents);

ui->detailsTable->verticalHeader()->setSectionResizeMode(QHeaderView::ResizeTo
Contents);
        loadDataFlag = true;

        ui->progress_Bar->setMaximum(1);
        ui->progress_Bar->setValue(1);
        ui->process_Details->setText(QString("数据加载成功..."));
}

void adminwin::on_clear_btn_clicked()
{
        ui->detailsTable->clearContents();
        ui->detailsTable->setRowCount(0);
        ui->detailsTable->setColumnCount(0);
        loadDataFlag = false;

        ui->progress_Bar->setMaximum(1);
        ui->progress_Bar->setValue(1);
        ui->process_Details->setText(QString("数据清理成功..."));
}
void adminwin::on_clear_btn_clicked()
{
        ui->detailsTable->clearContents();
        ui->detailsTable->setRowCount(0);
        ui->detailsTable->setColumnCount(0);
        loadDataFlag = false;
```

```
        ui->progress_Bar->setMaximum(1);
        ui->progress_Bar->setValue(1);
        ui->process_Details->setText(QString("数据清理成功..."));
}
void adminwin::on_update_status_btn_clicked()
{
        on_clear_btn_clicked();
        on_load_login_btn_clicked();
}
```

用户管理功能程序运行结果见图 5.37 所示。

图 5.37 用户管理功能实现

5.5 本章小结

本章深入探讨了社区食堂系统的功能和界面实现过程。首先对社区食堂系统的整体功能流程做了简短的阐述，其后对社区食堂系统中的主要功能从功能模块流程和界面设计的角度进行了深入介绍。最后，对社区食堂系统的主要功能代码进行了深入讲解并实际展示了代码的实现结果。

6 系统测试

6.1 引言

本章节将对社区食堂系统的系统测试进行阐述。首先说明系统的测试内容和测试环境，随后介绍系统功能测试的测试项目和测试结果。最后，对系统的非功能性测试，如安全性、可靠性和性能等方面进行全面的测试结果展示。

6.2 测试内容

社区食堂系统的测试采用了主要以黑盒测试为策略的方法，对所有模块进行全面、规范、系统的测试，包括购物车模块、历史订单模块等。本轮测试根据前述的功能性需求和非功能性需求来制定详细的测试环节和计划。在功能性测试过程中，针对各功能服务模块的内容设计不同的测试案例进行验证。测试案例执行完毕后，会发现其中的缺陷并根据其优先级进行修复，以最大程度地避免系统漏洞可能带来的损失。在非功能性测试过程中，利用相关工具和统计数据对整个软件系统进行性能测试、可靠性测试等环节，并进行统计分析。

6.3 测试环境

为模拟一般用户在实际使用社区食堂系统时的情景，在系统测试的软件环境层面，使用了市面上较为常用的操作系统 windows11，数据库使用 Mysql。同时硬件环境使用八核 CPU、AMD Ryzen 7 3750H 处理器、8G 内存以及 500G SSD 硬盘的配置。

6.4 功能性测试

功能性测试涉及社区食堂系统的登录功能、注册功能、用户头像功能、用户菜品浏览功能、用户公告浏览功能、购物车功能、立即购买功能、订单详情功能等。

6.4.1 登录功能

测试用例：使用用户 zyt 进行登录。
测试结果：成功登录用户名为 zyt 的用户信息。

6.4.2 注册功能

测试用例：注册用户名为 ABC 的用户。
测试结果：成功注册用户信息并实现用户登录。

6.4.3 用户头像功能

测试用例：使用新用户 ABC 的登录界面更换用户头像。
测试结果：成功更换用户头像信息，并在重启程序重新登陆后头像信息没有改变，成功验证头像信息持久化成功。

6.4.4 用户菜品浏览功能

测试用例：用户使用菜品分页功能进行菜品浏览。
测试结果：成功浏览用户可浏览信息，高级管理员可浏览信息成功对用户屏蔽。

6.4.5 用户公告浏览功能

测试用例：用户使用公告轮播图实时查看公告信息，通过公告按钮实现公告的前后切换功能。
测试结果：成功实现用户的公告浏览功能，并将高级管理员控制公告信息功能对用户成功进行屏蔽。

6.4.6 购物车功能

测试用例：用户通过菜品个数选择按钮选择菜品数量添加至购物车，并对不需要菜品进行删除后，进行购买。
测试结果：成功实现多种菜品添加购物车，并对购物车中菜品进行删除后进行购买。

6.4.7 立即购买功能

测试用例：用户通过菜品个数按钮选择菜品数量后立即购买该菜品。
测试结果：成功实现一种菜品多个数量的立即购买功能。

6.4.8 订单详情功能

测试用例：用户使用购物车功能和立即购买功能后生成对应时间的菜品购买订单，并在菜品详情功能中查看历史订单，对不需要的历史订单进行删除，对需要详细了解的订单内容进行详细内容查看。
测试结果：成功实现购买菜品后生成菜品订单功能，并实现对菜品订单的删除和详情查看功能。

6.4.9 管理员用户信息管理功能

测试用例：使用管理员账号登录社区食堂系统，进入用户管理界面，查看所有系统中的用户信息，并依次执行以下操作：添加用户 123 密码 123，删除用户 ABC，修改用户 zyt 密码为 123，使用新用户 123、zyt 和 ABC 登录系统。
测试结果：成功实现用户管理界面的增删改查功能，并使用用户名为 123 和 zyt 的新修改信息成功登录系统且用户购物车和订单等信息依然存在，使用用户 ABC 成功实现登录系统失败，并使用注册功能注册用户 ABC 后成功登录系统，用户 ABC 的用户信息实现初始化，以往用户信息成功得到删除。

6.4.10 管理员菜品管理功能

测试用例：使用管理员账号登录社区食堂系统，修改已有菜品信息，并添加新的菜品信息。
测试结果：成功实现已有菜品信息的修改和删除功能，成功实现新菜品信息的添加功能。

6.4.11 管理员系统给公告管理功能

测试用例：使用管理员账号登录社区食堂系统，查看并修改系统公告信息。
测试结果：成功实现系统公告的查看和修改功能。

6.5 非功能测试

非功能性测试涉及社区食堂系统的安全性测试、可靠性测试、性能测试、可用性测试和兼容性测试。

6.5.1 安全性测试

（1）用户身份认证：成功验证系统对用户身份的识别和认证机制安全可靠。不同用户登录，系统会对用户区分为管理员还是普通用户。
（2）数据加密：成功验证系统对重要数据的加密和解密过程，保障数据安全性。系统对用户密码采用隐码的方式展现，非目标用户不可见隐藏信息。
（3）访问控制：成功验证系统对不同用户角色和权限的管理及访问控制情况。只有当用户以管理员身份登录系统时，才具有管理员的能力权限。

6.5.2 可靠性测试

（1）系统稳定性：成功验证系统在长时间运行和高负荷情况下保持稳定性。
（2）数据完整性：成功验证系统在数据传输、存储和处理过程中不会丢失或损坏数据。

6.5.3 性能测试

（1）响应时间：测试系统在接收请求后的响应速度为 1s 内。

6.5.4 可用性测试

（1）易用性：系统的用户界面和交互设计符合用户习惯、易于操作。
（2）可理解性：系统的信息和提示能够清晰易懂地传达给用户。
（3）可靠性：系统能够始终可用，对用户的每个请求都有可靠的响应。

6.6 本章小结

本章首先介绍了社区食堂系统的测试内容和测试环境，其后详细讨论了社区食堂系统的功能性测试和非功能性测试，其中包括安全性测试、可靠性测试、性能测试、可用性测试等。通过这些测试，系统的各项质量特征得到了全面评估，确保系统在安全、可靠、高性能的同时，也能提供良好的用户体验。这一系列测试将对系统的全面上线提供有力保障，并在后续的系统优化和改进中提供参考依据。

7 总结与展望

7.1 总结

本毕业设计项目致力于开发基于 Qt 的社区食堂系统，旨在为社区居民提供便捷、有效的社区食堂点餐功能，从而提升社区内居民营养均衡发展，餐饮统一管理的目的。在项目实施过程中，深入研究和应用了 Qt 平台，以及 Mysql 数据库。通过此系统，用户可以选择适合自己的菜品进行点餐，系统会将用户的菜品进行后台处理，并将用户的菜品准时送到用户面前。同时，结合管理员权限功能，进一步对食堂系统的安全性和健康性进行统一管理。此外，通过 Qt Designer 和 QSS 技术，实现了友好的前端设计，使得用户可以以相对简单的方式与界面进行交互。

7.2 展望

未来，我们将重点关注社区食堂系统实施和推广阶段的具体实践，包括系统部署实施、用户培训和支持，以及后续的维护和优化工作。同时，我们也将密切关注社区食堂系统在实际应用中的效果和反馈，不断改进和优化系统，使其能够更好地满足社区居民的实际需求，提升社区食堂的服务水平和现代化管理水平，为社区食堂系统的发展添砖加瓦。

通过对社区食堂系统的全面需求分析和系统测试，本文为社区食堂系统的实施和推广提供了有力的理论和实践支持，促进社区食堂服务的提升和现代化。希望本论文的研究成果能够为相关领域的学者和从业者提供有益的参考和借鉴，推动社区食堂系统的发展和应用。

参考文献

[1] Dierinck P.Community-Based Mental Healthcare for Psychosis:From Homelessness to Recovery and Continued In-home Support[M]. Taylor and Francis: 2023-04-25. DOI:10.4324/9781003220015.

[2] 刘佳影. 基于MINA框架的社区食堂智慧餐饮系统 [J]. 信息与电脑(理论版), 2023, 35 (02): 144-146.

[3] 崔娜,宋珂欣. 智慧社区管理系统建设概述 [J]. 建筑与文化, 2024, (05): 123-125. DOI:10.19875/j.cnki.jzywh.2024.05.040.

[4] 程嘉萱,邝慧仪. 智慧社区垃圾分类系统设计 [J]. 包装工程, 2024, 45 (06): 458.

[5] 黄李垚,何妙婧,赖彦蓉等.高校食堂智慧化点餐与营养分析系统设计与探讨[J].农产品加工,2023(05):116-120

[6] 刘婧莉,常贤发.智慧食堂用餐管理系统的设计与实现[J].电脑编程技巧与维护,2021(04):90-92

[7] 刘佳影.基于 MINA 框架的社区食堂智慧餐饮系统[J].信息与电脑(理论版),2023,35(2):144-146

[8] 任硕果.智慧食堂管理系统分析与设计[J].数字技术与应用,2016(06):177

[9] 曹婷.高校食堂订餐系统的研究与分析[D].云南大学,2016(04)

[10] 陈爽.高校食堂网上订餐系统[J].电脑知识与技术,2010,6(24):6755-6756

[11] 熊群毓.大数据时代 MySQL 数据库的应用分析[J].信息与电脑(理论版),2023,35(14):209-212

[12] 侯健明,静国刚,吴松洋等.基于 QT 的网络设备拓扑管理平台设计与实现[J].工业控制计算机,2022,35(01):87-88

[13] 潘志安,高知林,秦华旺等.基于Qt的地铁站智能照明系统软件设计与实现[J].工业控制计算机,2020,33(10):113-115

[14] 杨芬,宋晓燕.MySQL 数据库应用的课程教学分析[J].电子技术,2023,52(10):180-181

[15] 王二飞.基于 Qt 的智能家居管理软件设计[J].无线互联科技,2023,20(04):19-22.

[16] 马宁宁.网页中实现轮播图的简易方法探讨[J].电脑知识与技术,2021,17(05):22-25

[17] Wu Daiwen.The Application and Management System of Scientific Research Projects Based on PHP and MySQL[J]. Journal of Interconnection Networks, 2022, 1

[18] Michal Kvet,Lucia Fidesova,Experimental comparison of syntax and semantics

of DBS Oracle and MySQL[J]. Proceedings of the XXth Conference of Open Innovations Association FRUCT,2016,128

[19] I K G Sudiartha,Sudiartha I K G,Indrayana I N E,Suasnawa I W,Asri S A;Sunu Putu Wijaya.Data Structure Comparison Between MySql Relational Database and Firebase Database NoSql on Mobile Based Tourist Tracking Application[J].Journal of Physics: Conference Series,2020,032092

致谢

感谢陈芳导师的关心、指导和教诲。陈芳导师追求真理、献身科学、严以律己、宽已待人的崇高品质对学生将是永远的鞭策。

作者在攻读学士学位期间的工作自始至终都是在陈芳导师全面、具体的指导下进行的。陈芳老师渊博的学识、敏锐的思维、民主而严谨的作风，使学生收益匪浅，终生难忘。

毕业设计（论文）知识产权声明

本人完全了解西安工业大学有关保护知识产权的规定，即：本科学生在校攻读学士学位期间毕业设计（论文）工作的知识产权属于西安工业大学。本人保证毕业离校后，使用毕业设计（论文）工作成果或用毕业设计（论文）工作成果发表论文时署名单位仍然为西安工业大学。学校有权保留送交的毕业设计（论文）的原文或复印件，允许毕业设计（论文）被查阅和借阅；学校可以公布毕业设计（论文）的全部或部分内容，可以采用影印、缩印或其他复制手段保存毕业设计（论文）。

（保密的毕业设计（论文）在解密后应遵守此规定）

毕业设计（论文）作者签名：

指导教师签名：

日期：

毕业设计（论文）独创性声明

秉承学校严谨的学风与优良的科学道德，本人声明所呈交的毕业设计（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。尽我所知，除了文中特别加以标注和致谢的地方外，毕业设计（论文）中不包含其他人已经发表或撰写过的成果，不包含他人已申请学位或其他用途使用过的成果。与我一同工作的同志对本研究所做的任何贡献均已在论文中作了明确的说明并表示了致谢。
毕业设计（论文）与资料若有不实之处，本人承担一切相关责任。

毕业设计（论文）作者签名：

指导教师签名：

日期：

附录

西安工业大学
XI'AN TECHNOLOGICAL UNIVERSITY

毕业论文外文资料

院 （系）：计算机科学与工程学院

专　　业：计算机科学与技术

班　　级：20060213

姓　　名：张宇涛

学　　号：2020032951

指导教师：陈芳

外文出处：SIGMOD 2014

附　　件：1.译文；2.原文

2024 年 6 月

附件 1 译文

小块数据驱动并行： 面向多核时代的 NUMA 有感知的查询评估框架

摘要

随着现代计算机体系结构的发展，有两个问题阻碍了最先进的并行查询执行方法：（1）为了利用多核的优势，所有的查询工作必须平均分配到（很快）数百个线程中，以达到良好的速度，然而（2）由于现代失序核的复杂性，即使有准确的数据统计，平均分配工作也很困难。因此，现有的 ″计划驱动 ″的并行方法遇到了负载平衡和上下文切换的瓶颈，因此不再有规模。多核架构面临的第三个问题是内存控制器的分散化，这导致了非统一内存访问（NUMA）。

作为回应，我们提出了 ″Morsel 驱动 ″的查询执行框架，其中调度成为一个细粒度的运行时任务，是 NUMA 感知的。Morsel 驱动的查询处理采用小块的输入数据（″小块″），并将这些数据调度给工人线程，这些线程运行整个运算器管道，直到下一个管道中断。并行性的程度并没有在计划中体现出来，而是可以在查询执行过程中弹性地改变，因此，调度员可以根据不同小块的执行速度重新行动，也可以根据工作负载中新到来的查询动态地调整资源。此外，调度器知道 NUMA 本地模块的数据位置和操作者的状态，这样，绝大多数的执行都是在 NUMA 本地内存上进行的。我们对 TPC-H 和 SSB 基准的评估显示了极高的绝对性能，在 32 个核心的情况下，平均速度提高了 30 多倍。

1 引言

如今，硬件性能提升的主要动力来自于多核并行性的提高，而不是单线程性能的加速[2]。到 SIGMOD 2014，英特尔即将推出的主流服务器 Ivy Bridge EX，可以运行 120 个并发线程。我们用多核这个词来形容这种具有几十或几百个核的架构。

同时，每台服务器的主内存容量增加到几 TB，导致了主内存数据库系统的发展。在这些系统中，查询处理不再受 I/O 限制，多核的巨大并行计算资源可以得到真正的利用。不幸的是，将内存控制器移到芯片中的趋势，以及因此而导致的内存访问的分散化--这是扩展巨大内存吞吐量所需要的--导致了非均匀内存访问（NUMA）。从本质上讲，计算机本身已经成为一个网络，因为数据项的访问成本根据数据和访问线程所在的芯片而不同。因此，多核并行化需要考虑到 RAM 和高速缓存的层次结构。特别是，必须仔细考虑 RAM 的 NUMA 划分，以确保线程在 NUMA 本地的数据上工作（大部分）。

20 世纪 90 年代对并行处理的大量研究导致大多数数据库系统采用了火山[12]模型中的一种并行形式，其中操作者基本上不知道并行。并行性被所谓的 ″交换 ″操作符所封装，这些操作符在多个线程之间路由元组流，每个线程执行查询计划中相同的流水线段。Volcano 模型的这种实现可以称为计划驱动：优化器在查询编译时静态地确定应该运行多少个线程，为每个线程实例化一个查询操作计划，并将其与交换操作符连接起来。

在本文中，我们提出了自适应的小鼠驱动的查询执行框架，这是我们为我们的主内存数据库系统 HyPer[16]设计的。我们的方法在图 1 中对三向连接查询 R IA S IB T 进行了描述。并行化是通过在不同的核心上并行处理每个管道来实现的，如图中的两个（上/红和下/蓝）管道所示。其核心思想是一种调度机制（″调度器″），允许灵活地并行执行运算器管道，甚至在查询执行过程中也能改变并行程度。一个查询被划分为若干段，每个执行段取一个输入图元（如 100,000）并执行这些图元，在下一个管道断路器中对结果进行匹配。如图中的颜色编码所示，Morsel 框架实现了 NUMA 本地处理： 一个线程对 NUMA 本地输入进行操作，并将其结果写入 NUMA 本地的存储区域中。我们的调度器运行固定的、与机器相关的线程数量，这样即使有新的查询到来，也不会出现资源超额占用的情况，而且这些线程被钉在核心上，这样就不会因为操作系统将线程移到不同的核心上而发生 NUMA 本地性的意外损失。

蚕食驱动调度的关键特征是，任务分配是在运行时完成的，因此是完全弹性的。这允许实现完美的负载平衡，即使面对不确定的中间结果的大小分布，以及难以预测的现代 CPU 内核的性能，即使它们得到的工作量是相同的，也会发生变化。它是有弹性的，因为它可以处理在运行时发生变化的工作负载（通过减少或增加已经在飞行中执行的查询的并行性），并且可以很容易地整合一个机制，以不同的优先级运行查询。

蛀虫驱动的想法从单纯的调度延伸到一个完整的查询执行框架，即所有的物理查询操作者必须能够在其所有的执行阶段（例如，哈希构建和探测）以蛀虫方式并行执行，根据阿姆达尔定律，这是实现多核可伸缩性的关键需求。morsel-wise 框架的一个重要部分是对数据位置性的认识。这从输入莫西尔和莫西尔化输出缓冲区的位置开始，但延伸到操作者可能创建和访问的状态（数据结构，如哈希表）。这种状态是共享数据，有可能被任何内核访问，但确实有高度的 NUMA 定位。因此，小块的调度是灵活的，但强烈倾向于最大化 NUMA 本地执行的调度选择。这意味着，只有在处理每个查询的几个摩西时，才会发生重新的 NUMA 访问，以实现负载平衡。通过主要访问本地 RAM，内存延迟得到了优化，跨插槽的内存流量也降到了最低，这可能会拖累其他线程。

在一个纯粹的基于 Volcano 的并行框架中，并行性被隐藏在运算符中，共享状态被避免，这导致计划在交换运算符中进行即时数据分割。我们认为，这并不总是导致最佳的计划（因为分割的努力并不总是有回报的），而通过即时分割实现的局部性可以由我们的局部感知的调度器来实现。其他的系统提倡每个操作者

的并行化[21]，以实现执行的灵活性，但这导致在一个管道段的操作者之间需要较少的同步。尽管如此，我们相信 morsel-wise 框架可以被整合到许多现有的系统中，例如，通过改变交换操作符的实现来封装 morsel-wise 调度，并引入例如哈希表共享。我们的框架也适用于使用及时（JIT）代码编译的系统[19, 25]，因为计划中出现的每个流水线的生成代码，随后可以进行分食式调度。事实上，我们的 HyPer 系统使用了这种 JIT 方法[25]。

在本文中，我们提出了一些相关的想法，以实现高效、可扩展和弹性的并行处理。主要的贡献是一个包含以下内容的查询引擎的架构蓝图：

- Morsel 驱动的查询执行是一个新的并行查询评估框架，它与传统的 Volcano 模型有很大的不同，因为它使用工作窃取的方式在线程之间动态地分配工作。这可以防止由于负载不平衡而导致的 CPU 资源未被使用，并允许弹性，即 CPU 资源可以在任何时候在不同的查询之间重新分配。

- 一套针对最重要的重理性化运算符的快速并行算法。

- 将 NUMA 意识整合到数据库系统中的系统性方案。

本文的其余部分组织如下。第 2 节专门详细讨论了管道并行化和数据碎片化的问题。在第 3 节中，我们讨论了调度器，它将任务（流水线作业）和小块（数据片段）分配给工作线程。调度器实现了完全的弹性，允许在任何时候改变工作在特定查询上的并行线程的数量。第 4 节讨论了并行连接、聚合和排序操作者的算法和同步细节。第 5 节通过整个 TPC-H 查询套件来评估该查询引擎的优点。在讨论了相关工作以指出我们的并行查询引擎架构的新颖性之后，我们在第 6 节中总结了本文。

2 小数据块驱动执行

改编自介绍中的激励性查询，我们将在以下查询计划的例子中展示我们的并行管道查询执行：

σ...(R) IA σ...(S) IB σ...(T)

假设 R 是最大的表（过滤后），优化器将选择 R 作为探测输入，并建立（团队）其他两个 S 和 T 的哈希表。由此产生的代数查询计划（由基于成本的优化器获得）由图 2 左侧所示的三个管道组成：

1. 扫描、过滤和建立基础关系 T 的哈希表 HT(T)、

2. 2. 扫描、过滤和建立论据 S 的哈希表 HT(S)、

3. 扫描、过滤 R 和探测 S 的哈希表 HT(S)，探测 T 的哈希表 HT(T)并存储结果图元。

HyPer 使用即时编译（JIT）来生成高效的机器代码。每个流水线段，包括所有操作器，都被编译成一个代码片段。这实现了非常高的原始性能，因为传统查询评估器的解释开销被消除了。此外，管道中的操作者甚至没有实现他们的中间结果，这仍然是由 Vectorwise[34]的已经非常高效的矢量一次评估引擎完成的。

代数计划中的小数据块驱动的执行是由一个所谓的 QEPobject 控制的，它将可执行的管道传输给一个调度器--参见第 3 节。QEPobject 的责任是观察数据的

依赖性。在我们的查询例子中，第三条（探测）管道只能在两个哈希表建立后执行，也就是在前两条管道完全执行后。对于每个管道，QEPobject 都会分配临时存储区域，执行该管道的并行线程会将其结果写入其中。在整个流水线完成后，临时存储区域被逻辑地重新分割成同等大小的小块；这样一来，后续的流水线就从新的相同大小的小块开始，而不是在流水线之间保留小块的边界，这很容易造成小块大小的偏差。任何时候在任何管道上工作的并行线程的数量都受处理器的硬件线程数量的限制。为了在本地写入 NUMA，并在写入中间结果时避免同步，QEPobject 为每个可执行管道的每个这样的线程/核分配了一个存储区域。

图 3 显示了过滤 T 和建立哈希表 H T（T）的流水线的并行处理。让我们专注于管道的第一阶段的处理，即过滤输入的 T 并将 ″幸存的 ″图元存储在临时存储区。

在我们的图中，显示了三个并行线程，每个线程一次在一个莫西干上操作。由于我们的基础关系 T 是在 NUMA 组织的内存中 ″以摩西为单位 ″存储的，调度器尽可能地分配一个位于线程执行的同一套接字上的摩西。这在图中用颜色表示： 在红色套接字的核心上运行的红色线程被分配到处理一个红色的碎片，也就是位于红色套接字上的基础关系 T 的一个小片段的任务。一旦，线程完成了对所分配的摩西的处理，它可以被委托（派发）到一个不同的任务，或者它获得另一个摩西（相同的颜色）作为其下一个任务。由于线程一次处理一个果子，系统是完全弹性的。在处理查询的过程中，可以在任何时候（更确切地说，在果子的边界）减少或增加并行度（MPL）。

(1)扫描/过滤输入 T 和(2)建立哈希表的逻辑代数管道实际上被分解成两个物理处理管道，在图的左侧标记为阶段。在第一阶段，过滤后的图元被插入到 NUMA 本地存储区，也就是说，为了避免同步，每个内核都有一个独立的存储区。为了在进一步的处理阶段保持 NUMA-locality，一个特定内核的存储区域被本地分配在同一个套接字上。

在所有的基表碎片被扫描和过滤后，在第二阶段，这些存储区域被扫描--同样是由循环在相应内核上的线程进行扫描--并将指针插入哈希表中。将逻辑哈希表的构建管道分为两个阶段，可以完美地确定全局哈希表的大小，因为在第一阶段完成后，″存活 ″对象的确切数量是已知的。这个（完美大小的）全局哈希表将被位于 NUMA 系统不同套接字上的线程探测；因此，为了避免竞争，它不应该驻留在一个特定的 NUMA 区域，因此被交错（分散）在所有套接字上。由于许多并行线程争相向该哈希表插入数据，因此无锁实现是必不可少的。哈希表的实现细节将在第 4.2 节描述。

在这两个哈希表构建完毕后，可以安排探测管道。探测管道的详细处理过程如图 4 所示。同样，一个线程向打补丁器请求工作，打补丁器在相应的 NUMA 分区中分配一个摩尔。也就是说，一个位于红色 NUMA 分区的核心上的线程被分配了一个位于核心上的基础关系 R 的摩西。

响应的 ″红色 ″NUMA 套接字。探究管道的结果再次存储在 NUMA 本地存储区

域，以便为进一步的处理保留 NUMA 定位（在我们的样本查询计划中没有出现）。

总的来说，Morsel 驱动的并行性是并行地执行多个管道，这与 Vol- cano 模型的典型实现相似。然而，与 Volcano 不同的是，这些管道并不独立。也就是说，它们共享数据结构，操作者意识到平行执行，必须执行同步（通过有效的无锁机制--见下文）。另一个区别是，执行计划的线程数量是完全弹性的。也就是说，如图 2 所示，线程数量不仅在不同的流水线段之间可能不同，而且在查询执行过程中，在同一流水线段内也可能不同--如下面所述。

3 调度器——并行 Pipeline 任务调度

调度器是控制和分配计算再源到并行管道的。这是通过将任务分配给工人线程来完成的。我们为机器提供的每个硬设备线程（pre-）创建一个工人线程，并将每个工人线程永久地绑定到它。因此，一个特定查询的并行程度不是通过创建或终止线程来控制的，而是通过给它们签署可能不同的查询的特定任务来控制的。分配给这样一个工人线程的任务包括一个管道作业和一个特定的磨盘，该管道必须在其上执行。任务的抢占发生在果壳的边界，从而消除了潜在的昂贵的中断机制。我们通过实验确定，在即时弹性调整、负载平衡和低维护开销之间，约有 100,000 个图元的摩西尔大小可以产生良好的权衡。

将任务分配给在特定核心上运行的线程有三个主要目标：

1. 通过将数据元组分配给分配元组的内核来保持(NUMA-)位置性
2. 关于特定查询的并行性水平的充分弹性
3. 负载平衡要求所有参与查询管道的核心在同一时间完成它们的工作，以防止（快）核等待其他（慢）核 1。

图 5 是调度器的结构简图。它维护一个待处理的管道作业的列表。这个列表只包含前提条件已经被处理的管道作业。例如，对于我们正在运行的示例查询，构建输入管道首先被插入到待处理作业列表中。只有在这两个构建管道完成后，才会插入探测管道。如前所述，每个活动查询都由一个 QEPob- ject 控制，该 QEPob- ject 负责将可执行管线传输给调度器。因此，调度器只维护所有从属管道已经被处理的管道工作的列表。一般来说，调度器队列将包含不同查询的待决管道作业，这些作业是平行执行的，以适应查询间的并行性。

3.1 弹性

通过 ″一次一粒 ″的工作调度实现的完全弹性并行，允许根据服务质量模型对这些查询间的并行管道工作进行智能调度。它可以优雅地降低长期运行的查询 Q1 在任何处理阶段的并行程度，以便优先处理可能更重要的交互查询 Q+。一旦优先级较高的查询 Q+完成，钟摆就会摆回到长期运行的查询上，将所有或大多数核心分配给长期运行的查询 Q1 的任务。在第 5.4 节中，我们通过实验来证明这种动态弹性。在我们目前的实现中，所有的查询都有相同的优先级，所以线程被平均分配到所有活动的查询上。基于优先级的调度组件正在开发中，但超出了本文的范围。

对于每一个流水线作业，调度器都会维护该流水线作业仍需执行的待处理小块的列表。对于每个核来说，都有一个单独的列表，以确保例如核 0 的工作请求返回一个与核 0 分配在同一套接字上的摩西。一旦核心 0 处理完所分配的食物，它就会请求一个新的任务，这个任务可能来自同一个管道工作，也可能不是。这取决于不同管道作业的优先级，这些作业来自于正在执行的不同查询。如果一个高优先级的查询进入系统，可能会导致当前查询的并行度降低。碎片化处理允许重新将核心分配给不同的流水线工作，而不需要任何激烈的中断机制。

3.2 实现综述

为了说明问题，我们在图 5 中为每个内核展示了一个（长的）链式列表。在现实中（即在我们的实现中），我们为每个核/套接字维护存储区域边界，并根据需要将这些大的存储区域分割成小块；也就是说，当一个核向调度器请求一个任务时，特定套接字上的管道参数存储区域的下一个小块被 ″切出来″。此外，在图 5 中，Dispatcher 看起来像是一个独立的线程。然而，这将招致两个缺点：（1）调度器本身需要一个核心来运行，或者可能预留查询评估线程；（2）它可能成为争论的来源，特别是如果小数据块量被配置得相当小。因此，调度器仅作为一个无锁数据结构来实现。然后，调度器的代码由请求工作的查询评估线程本身执行。因此，调度器被自动地在这个工作线程的（否则未使用的）核心上执行。依靠无锁数据结构（即流水线作业队列以及相关的小数据块队列），即使多个查询评估线程同时请求新的任务，也能减少争论。类似地，通过观察数据依赖关系（例如，在执行探测管道之前建立哈希表）来触发特定查询进展的 QEPobject 被实现为一个无源状态机。每当一个管道工作被完全执行时，该代码就会被调度员调用，因为在工作请求时无法找到新的小数据块。同样，这个状态机是在最初向调度器请求新任务的工人线程的其他未使用的核心上执行。

除了能够在任何时候将一个核心分配给不同的查询--称为弹性--外，″小数据块″式处理还保证了负载平衡和抗偏斜。所有在同一流水线上工作的线程都以 ″拍照完成 ″的方式运行：它们被保证在处理一个小数据块的时间段内到达终点。如果由于某种原因，一个核心在其特定的套接字上完成了所有的小数据块，调度器将从另一个核心 ″偷工″，也就是说，它将把小数据块分配到不同的套接字。在一些 NUMA 系统中，并不是所有的套接字都是相互直接连接的；在这种情况下，先从较近的套接字上偷工是值得的。在正常情况下，从远程套接字中窃取工作的情况很少发生；然而，有必要避免空闲线程。而且，无论如何，写入临时存储区的工作将在 NUMA 本地存储区完成（也就是说，如果在从红色套接字上的核心窃取工作的过程中，一个红色的小数据块被一个蓝色的核心处理过，那么它就会变成蓝色）。

到目前为止，我们已经讨论了管线内的并行性。我们的并行化方案也可以支持总线并行，例如，我们的例子中 ″过滤和建立 T 的哈希表 ″和 ″过滤和建立 S 的哈希表 ″这两条管道是独立的，因此可以并行执行。然而，这种形式的并行性的作用是有限的。独立管线的数量通常比内核的数量小得多，而且每个管线的工

作量通常是不同的。此外，丛式并行会因为减少缓存定位而降低性能。因此，我们目前避免从一个查询中并行执行多个管道；在我们的例子中，我们首先执行管道 T，只有在 T 完成后，管道 S 的工作才被添加到管道工作列表中。

除了弹性，小数据块驱动处理还允许简单而优雅地实现查询的取消。一个用户可能放弃了她的查询请求，查询中发生了异常（例如，数字溢出），或者系统的 RAM 耗尽。如果这些事件中的任何一个发生了，涉及的查询就会在调度器中被标记。每当该查询的一个片段完成时，就会检查该标记，因此，很快所有的工作线程就会停止在该查询上工作。与强迫操作系统杀死线程相比，这种方法允许每个线程进行清理（例如，释放分配的内存）。

3.3 数据块大小

与 Vectorwise[9]和 IBM 的 BLU[31]等系统相比，这些系统使用向量/步长在操作者之间传递数据，如果一个小数据块不适合进入缓存，则不会有性能上的损失。Morsels 被用来将一个大的任务分解成小的、大小不变的工作单元，以方便工作的窃取和抢占。因此，小数据块的大小对性能不是很关键，它只需要大到足以摊销调度开销，同时提供良好的再响应时间。为了显示食物大小对查询性能的影响，我们测量了在 Nehalem EX 系统上使用 64 个线程从 R 中选择 min(a)的查询的性能，这在第 5 节有描述。这个查询非常简单，所以它尽可能地强调了偷工减料的数据结构。图 6 显示，在开销可以忽略不计的情况下，小数据块大小应该被设置为最小的可行值，在这种情况下，应该设置为高于 10,000 的值。最佳设置取决于硬件，但可以很容易地通过实验来确定。

在多核系统中，任何共享数据结构，即使是无锁的，最终也会成为一个瓶颈。然而，在我们的偷工减料数据结构的情况下，有许多方面可以防止它成为一个可扩展性问题。首先，在我们的实现中，总的工作最初在所有线程之间分配，这样，每个线程暂时拥有一个本地范围。因为我们对每个范围进行了缓存线的调整，所以在缓存线层面上的冲突是不可能的。只有当这个本地范围用尽时，一个线程才会试图从另一个范围偷取工作。其次，如果有多个查询被同时执行，数据结构的压力就会进一步减少。最后，总是有可能增加小数据块的大小。这将导致对偷工减料数据结构的访问减少。在最坏的情况下，过大的小数据块大小会导致线程利用不足，但如果有足够的并发查询正在执行，则不会影响系统的吞吐量。

附件 2 原文

Morsel-Driven Parallelism: A NUMA-Aware Query Evaluation Framework for the Many-Core Age

ABSTRACT

With modern computer architecture evolving, two problems con- spire against the state-of-the-art approaches in parallel query exe- cution: (i) to take advantage of many-cores, all query work must be distributed evenly among (soon) hundreds of threads in order to achieve good speedup, yet (ii) dividing the work evenly is difficult even with accurate data statistics due to the complexity of modern out-of-order cores. As a result, the existing approaches for “plan- driven” parallelism run into load balancing and context-switching bottlenecks, and therefore no longer scale. A third problem faced by many-core architectures is the decentralization of memory con- trollers, which leads to Non-Uniform Memory Access (NUMA).

In response, we present the “morsel-driven” query execution framework, where scheduling becomes a fine-grained run-time task that is NUMA-aware. Morsel-driven query processing takes small fragments of input data (“morsels”) and schedules these to worker threads that run entire operator pipelines until the next pipeline breaker. The degree of parallelism is not baked into the plan but can elastically change during query execution, so the dispatcher can re- act to execution speed of different morsels but also adjust resources dynamically in response to newly arriving queries in the workload. Further, the dispatcher is aware of data locality of the NUMA-local morsels and operator state, such that the great majority of execu- tions takes place on NUMA-local memory. Our evaluation on the TPC-H and SSB benchmarks shows extremely high absolute per- formance and an average speedup of over 30 with 32 cores.

1. INTRODUCTION

The main impetus of hardware performance improvement nowa- days comes from increasing multi-core parallelism rather than from speeding up single-threaded performance [2]. By SIGMOD 2014 Intel’s forthcoming mainstream server Ivy Bridge EX, which can run 120 concurrent threads, will be available. We use the term many-core for such architectures with tens or hundreds of cores.

At the same time, increasing main memory capacities of up to several TB per server have led to the development of main-memory database systems. In these systems query processing is no longer I/O bound, and the huge parallel compute resources of many-cores can be truly exploited. Unfortunately, the trend to move memory controllers into the chip and hence the decentralization of mem- ory access, which was needed to scale throughput to huge mem- ories, leads to non-uniform memory access (NUMA). In essence, the computer has become a network in itself as the access costs of data items varies depending on which chip the data and the access- ing thread are located. Therefore, many-core parallelization needs to take RAM and cache hierarchies into account. In particular, the NUMA division of the RAM has to be considered carefully to en- sure that threads work (mostly) on NUMA-local data.

Abundant research in the 1990s into parallel processing led the majority of database systems to adopt a form of parallelism in- spired by the Volcano [12] model, where operators are kept largely unaware of parallelism. Parallelism is encapsulated by so-called “exchange” operators that route tuple streams between multiple threads each executing identical pipelined segments of the query plan. Such implementations of the Volcano model can be called plan-driven: the optimizer statically determines at query compile- time how many threads should run, instantiates one query operator plan for each thread, and connects these with exchange operators.

In this paper we present the adaptive morsel-driven query execu- tion framework, which we designed for our main-memory database system HyPer [16]. Our approach is sketched in Figure 1 for the three-way-join query R IA S IB T. Parallelism is achieved by processing each pipeline on different cores in parallel, as indi- cated by the two (upper/red and lower/blue) pipelines in the fig- ure. The core idea is a scheduling mechanism (the “dispatcher”) that allows flexible parallel execution of an operator pipeline, that can change the parallelism degree even during query execution. A query is divided into segments, and each executing segment takes a morsel (e.g, 100,000) of input tuples and executes these, mate- rializing results in the next pipeline breaker. The morsel frame- work enables NUMA local processing as indicated by the color coding in the figure: A thread operates on NUMA-local input and writes its result into a NUMA-local storage area. Our dispatcher runs a fixed, machine-dependent number of threads, such that even if new queries arrive there is no resource over-subscription, and these threads are pinned to the cores, such that no unexpected loss of NUMA locality can occur due to the OS moving a thread to a different core.

The crucial feature of morsel-driven scheduling is that task dis- tribution is done at run-time and is thus fully elastic. This allows to achieve perfect load balancing, even in the face of uncertain size distributions of intermediate results, as well as the hard-to-predict performance of modern CPU cores that varies even if the amount of work they get is the same. It is elastic in the sense that it can handle workloads that change at run-time (by reducing or increas- ing the parallelism of already executing queries in-flight) and can easily integrate a mechanism to run queries at different priorities.

The morsel-driven idea extends from just scheduling into a com- plete query execution framework in that all physical query opera- tors must be able to execute morsel-wise in parallel in all their ex- ecution stages (e.g., both hash-build and probe), a crucial need for achieving many-core scalability in the light of Amdahl’ s law. An important part of the morsel-wise framework is awareness of data locality. This starts from the locality of the input morsels and ma- terialized output buffers, but extends to the state (data structures, such as hash tables) possibly created and accessed by the opera- tors. This state is shared data that can potentially be accessed by any core, but does have a high degree of NUMA locality. Thus morsel-wise scheduling is flexible, but strongly favors scheduling choices that maximize NUMA-local execution. This means that re- mote NUMA access only happens when processing a few morsels per query, in order to achieve load balance. By accessing local RAM mainly, memory latency is optimized and cross-socket mem- ory traffic, which can slow other threads down, is minimized.

In a pure Volcano-based parallel framework, parallelism is hid- den from operators and shared state is avoided, which leads to plans doing on-the-fly data partitioning in the exchange operators. We argue that this does not always lead to the optimal plan (as parti- tioning effort does not always pay off), while the locality achieved by on-the-fly partitioning can be achieved by our locality-aware dispatcher. Other systems have advocated per-operator paralleliza- tion [21] to achieve flexibility in execution, but this leads to need- less synchronization between operators in one pipeline segment. Nevertheless, we are convinced that the morsel-wise framework can be integrated in many existing systems, e.g., by changing the implementation of exchange operators to encapsulate morsel-wise scheduling, and introduce e.g., hash-table sharing. Our framework also fits systems using Just-In-Time (JIT) code compilation [19, 25] as the generated code for each pipeline occurring in the plan, can subsequently be scheduled morsel-wise. In fact, our HyPer system uses this JIT approach [25].

In this paper we present a number of related ideas that enable efficient, scalable, and elastic parallel processing. The main contri- bution is an architectural blueprint for a query engine incorporating the following:

- Morsel-driven query execution is a new parallel query eval- uation framework that fundamentally differs from the tra- ditional Volcano model in that it distributes work between threads dynamically using work-stealing. This prevents un- used CPU resources due to load imbalances, and allows for elasticity, i.e., CPU resources can be reassigned between dif- ferent queries at any time.
- A set of fast parallel algorithms for the most important re- lational operators.
- AsystematicapproachtointegratingNUMA-awarenessinto database systems.

The remainder of this paper is organized as follows. Section 2 is devoted to a detailed discussion of pipeline parallelization and the fragmentation of the data into morsels. In Section 3 we discuss the dispatcher, which assigns tasks (pipeline jobs) and morsels (data fragments) to the worker threads. The dispatcher enables the full elasticity which allows to vary the number of parallel threads work- ing on a particular query at any time. Section 4 discusses algorith- mic and synchronization

details of the parallel join, aggregation, and sort operators. The virtues of the query engine are evaluated in Section 5 by way of the entire TPC-H query suite. After discussing related work in order to point out the novelty of our parallel query engine architecture in Section 6, we conclude the paper.

2. MORSEL-DRIVEN EXECUTION

Adapted from the motivating query of the introduction, we will demonstrate our parallel pipeline query execution on the following example query plan:

σ...(R) IA σ...(S) IB σ...(T)

Assuming that R is the largest table (after filtering) the optimizer would choose R as probe input and build (team) hash tables of the other two, S and T . The resulting algebraic query plan (as obtained by a cost-based optimizer) consists of the three pipelines illustrated on the left-hand side of Figure 2:

1. Scanning, filtering and building the hash table HT(T) of base relation T ,

2. Scanning, filtering and building the hash table HT(S) of ar- gument S,

3. Scanning, filtering R and probing the hash table HT(S) of S and probing the hash table HT(T) of T and storing the result tuples.

HyPer uses Just-In-Time (JIT) compilation to generate highly efficient machine code. Each pipeline segment, including all oper- ators, is compiled into one code fragment. This achieves very high raw performance, since interpretation overhead as experienced by traditional query evaluators, is eliminated. Further, the operators in the pipelines do not even materialize their intermediate results, which is still done by the already much more efficient vector-at-a- time evaluation engine of Vectorwise [34].

The morsel-driven execution of the algebraic plan is controlled by a so called QEPobject which transfers executable pipelines to a dispatcher – cf. Section 3. It is the QEPobject’s responsibility to observe data dependencies. In our example query, the third (probe) pipeline can only be executed after the two hash tables have been built, i.e., after the first two pipelines have been fully executed. For each pipeline the QEPobject allocates the temporary storage areas into which the parallel threads executing the pipeline write their results. After completion of the entire pipeline the tempo- rary storage areas are logically re-fragmented into equally sized morsels; this way the succeeding pipelines start with new homoge- neously sized morsels instead of retaining morsel boundaries across pipelines which could easily result in skewed morsel sizes. The number of parallel threads working on any pipeline at any time is bounded by the number of hardware threads of the processor. In order to write NUMA-locally and to avoid synchronization while writing intermediate results the QEPobject allocates a storage area for each such thread/core for each executable pipeline.

The parallel processing of the pipeline for filtering T and build- ing the hash table H T (T) is shown in Figure 3. Let us concentrate on the processing of the first phase of the pipeline that filters in- put T and stores the “surviving” tuples in temporary storage areas.

In our figure three parallel threads are shown, each of which op- erates on one morsel at a time. As our base relation T is stored “morsel-wise” across a

NUMA-organized memory, the scheduler assigns, whenever possible, a morsel located on the same socket where the thread is executed. This is indicated by the coloring in the figure: The red thread that runs on a core of the red socket is assigned the task to process a red-colored morsel, i.e., a small frag- ment of the base relation T that is located on the red socket. Once, the thread has finished processing the assigned morsel it can either be delegated (dispatched) to a different task or it obtains another morsel (of the same color) as its next task. As the threads pro- cess one morsel at a time the system is fully elastic. The degree of parallelism (MPL) can be reduced or increased at any point (more precisely, at morsel boundaries) while processing a query.

The logical algebraic pipeline of (1) scanning/filtering the input T and (2) building the hash table is actually broken up into two physical processing pipelines marked as phases on the left-hand side of the figure. In the first phase the filtered tuples are inserted into NUMA-local storage areas, i.e., for each core there is a sep- arate storage area in order to avoid synchronization. To preserve NUMA-locality in further processing stages, the storage area of a particular core is locally allocated on the same socket.

After all base table morsels have been scanned and filtered, in the second phase these storage areas are scanned – again by threads lo- cated on the corresponding cores – and pointers are inserted into the hash table. Segmenting the logical hash table building pipeline into two phases enables perfect sizing of the global hash table be- cause after the first phase is complete, the exact number of “surviv- ing” objects is known. This (perfectly sized) global hash table will be probed by threads located on various sockets of a NUMA sys- tem; thus, to avoid contention, it should not reside in a particular NUMA-area and is therefore is interleaved (spread) across all sock- ets. As many parallel threads compete to insert data into this hash table, a lock-free implementation is essential. The implementation details of the hash table are described in Section 4.2.

After both hash tables have been constructed, the probing pipeline can be scheduled. The detailed processing of the probe pipeline is shown in Figure 4. Again, a thread requests work from the dis- patcher which assigns a morsel in the corresponding NUMA parti- tion. That is, a thread located on a core in the red NUMA partition is assigned a morsel of the base relation R that is located on the cor

responding “red” NUMA socket. The result of the probe pipeline is again stored in NUMA local storage areas in order to preserve NUMA locality for further processing (not present in our sample query plan).

In all, morsel-driven parallelism executes multiple pipelines in parallel, which is similar to typical implementations of the Vol- cano model. Different from Volcano, however, is the fact that the pipelines are not independent. That is, they share data struc- tures and the operators are aware of parallel execution and must perform synchronization (through efficient lock-free mechanisms – see later). A further difference is that the number of threads exe- cuting the plan is fully elastic. That is, the number may differ not only between different pipeline segments, as shown in Figure 2, but also inside the same pipeline segment during query execution – as

described in the following.

3. DISPATCHER: SCHEDULING PARALLEL PIPELINE TASKS

The dispatcher is controlling and assigning the compute re- sources to the parallel pipelines. This is done by assigning tasks to worker threads. We (pre-)create one worker thread for each hard- ware thread that the machine provides and permanently bind each worker to it. Thus, the level of parallelism of a particular query is not controlled by creating or terminating threads, but rather by as- signing them particular tasks of possibly different queries. A task that is assigned to such a worker thread consists of a pipeline job and a particular morsel on which the pipeline has to be executed. Preemption of a task occurs at morsel boundaries – thereby elimi- nating potentially costly interrupt mechanisms. We experimentally determined that a morsel size of about 100,000 tuples yields good tradeoff between instant elasticity adjustment, load balancing and low maintenance overhead.

There are three main goals for assigning tasks to threads that run on particular cores:

1. Preserving (NUMA-)locality by assigning data morsels to cores on which the morsels are allocated

2. Full elasticity concerning the level of parallelism of a partic- ular query

3. Loadbalancingrequiresthatallcoresparticipatinginaquery pipeline finish their work at the same time in order to prevent (fast) cores from waiting for other (slow) cores1.

In Figure 5 the architecture of the dispatcher is sketched. It maintains a list of pending pipeline jobs. This list only contains pipeline jobs whose prerequisites have already been processed. E.g., for our running example query the build input pipelines are first inserted into the list of pending jobs. The probe pipeline is only inserted after these two build pipelines have been finished. As de- scribed before, each of the active queries is controlled by a QEPob- ject which is responsible for transferring executable pipelines to the dispatcher. Thus, the dispatcher maintains only lists of pipeline jobs for which all dependent pipelines were already processed. In general, the dispatcher queue will contain pending pipeline jobs of different queries that are executed in parallel to accommodate inter-query parallelism.

3.1 Elasticity

The fully elastic parallelism, which is achieved by dispatching jobs “a morsel at a time”, allows for intelligent scheduling of these inter-query parallel pipeline jobs depending on a quality of service model. It enables to gracefully decrease the degree of parallelism of, say a long-running query Ql at any stage of processing in order to prioritize a possibly more important interactive query Q+. Once the higher prioritized query Q+ is finished, the pendulum swings back to the long running query by dispatching all or most cores to tasks of the long running query Ql. In Section 5.4 we demonstrate this dynamic elasticity experimentally. In our current implemen- tation all queries have the same priority, so threads are distributed equally over all active queries. A priority-based scheduling com- ponent is under development but beyond

the scope of this paper.

For each pipeline job the dispatcher maintains lists of pending morsels on which the pipeline job has still to be executed. For each core a separate list exists to ensure that a work request of, say, Core 0 returns a morsel that is allocated on the same socket as Core 0. This is indicated by different colors in our architectural sketch. As soon as Core 0 finishes processing the assigned morsel, it requests a new task, which may or may not stem from the same pipeline job. This depends on the prioritization of the different pipeline jobs that originate from different queries being executed. If a high-priority query enters the system it may lead to a decreased parallelism de- gree for the current query. Morsel-wise processing allows to re- assign cores to different pipeline jobs without any drastic interrupt mechanism.

3.2 Implementation Overview

For illustration purposes we showed a (long) linked list of morsels for each core in Figure 5. In reality (i.e., in our implementation) we maintain storage area boundaries for each core/socket and segment these large storage areas into morsels on demand; that is, when a core requests a task from the dispatcher the next morsel of the pipeline argument’s storage area on the particular socket is “cut out”. Furthermore, in Figure 5 the Dispatcher appears like a sep- arate thread. This, however, would incur two disadvantages: (1) the dispatcher itself would need a core to run on or might pre- empt query evaluation threads and (2) it could become a source of contention, in particular if the morsel size was configured quite small. Therefore, the dispatcher is implemented as a lock-free data structure only. The dispatcher’s code is then executed by the work- requesting query evaluation thread itself. Thus, the dispatcher is au- tomatically executed on the (otherwise unused) core of this worker thread. Relying on lock-free data structures (i.e., the pipeline job queue as well as the associated morsel queues) reduces contention even if multiple query evaluation threads request new tasks at the same time. Analogously, the QEPobject that triggers the progress of a particular query by observing data dependencies (e.g., build- ing hash tables before executing the probe pipeline) is implemented as a passive state machine. The code is invoked by the dispatcher whenever a pipeline job is fully executed as observed by not being able to find a new morsel upon a work request. Again, this state machine is executed on the otherwise unused core of the worker thread that originally requested a new task from the dispatcher.

Besides the ability to assign a core to a different query at any time – called elasticity – the morsel-wise processing also guaran- tees load balancing and skew resistance. All threads working on the same pipeline job run to completion in a “photo finish”: they are guaranteed to reach the finish line within the time period it takes to process a single morsel. If, for some reason, a core finishes processing all morsels on its particular socket, the dispatcher will “steal work” from another core, i.e., it will assign morsels on a dif- ferent socket. On some NUMA systems, not all sockets are directly connected with each other; here it pays off to steal from closer sock- ets first. Under normal circumstances, work-stealing from remote sockets happens very infrequently; nevertheless it is necessary to avoid idle threads. And the

writing into temporary storage will be done into NUMA local storage areas anyway (that is, a red morsel turns blue if it was processed by a blue core in the process of steal- ing work from the core(s) on the red socket).

So far, we have discussed intra-pipeline parallelism. Our par- allelization scheme can also support bushy parallelism, e.g., the pipelines “filtering and building the hash table of T ” and “filtering and building the hash table of S” of our example are independent and could therefore be executed in parallel. However, the useful- ness of this form of parallelism is limited. The number of indepen- dent pipelines is usually much smaller than the number of cores, and the amount of work in each pipeline generally differs. Fur- thermore, bushy parallelism can decrease performance by reducing cache locality. Therefore, we currently avoid to execute multiple pipelines from one query in parallel; in our example, we first exe- cute pipeline T , and only after T is finished, the job for pipeline S is added to the list of pipeline jobs.

Besides elasticity, morsel-driven processing also allows for a simple and elegant implementation of query canceling. A user may have aborted her query request, an exception happened in a query (e.g., a numeric overflow), or the system is running out of RAM. If any of these events happen, the involved query is marked in the dispatcher. The marker is checked whenever a morsel of that query is finished, therefore, very soon all worker threads will stop work- ing on this query. In contrast to forcing the operating system to kill threads, this approach allows each thread to clean up (e.g., free allocated memory).

3.3 Morsel Size

In contrast to systems like Vectorwise [9] and IBM’ s BLU [31], which use vectors/strides to pass data between operators, there is no performance penalty if a morsel does not fit into cache. Morsels are used to break a large task into small, constant-sized work units to facilitate work-stealing and preemption. Consequently, the morsel size is not very critical for performance, it only needs to be large enough to amortize scheduling overhead while providing good re- sponse times. To show the effect of morsel size on query per- formance we measured the performance for the query select min(a) from R using 64 threads on a Nehalem EX system, which is described in Section 5. This query is very simple, so it stresses the work-stealing data structure as much as possible. Fig- ure 6 shows that the morsel size should be set to the smallest pos- sible value where the overhead is negligible, in this case to a value above 10,000. The optimal setting depends on the hardware, but can easily be determined experimentally.

On many-core systems, any shared data structure, even if lock- free, can eventually become a bottleneck. In the case of our work- stealing data structure, however, there are a number of aspects that prevent it from becoming a scalability problem. First, in our imple- mentation the total work is initially split between all threads, such that each thread temporarily owns a local range. Because we cache line align each range, conflicts at the cache line level are unlikely. Only when this local range is exhausted, a thread tries to steal work from another range. Second, if more than one query is executed concurrently, the pressure on the data structure is further

reduced. Finally, it is always possible to increase the morsel size. This re- sults in fewer accesses to the work-stealing data structure. In the worst case a too large morsel size results in underutilized threads but does not affect throughput of the system if enough concurrent queries are being executed.