上海交通大学工程硕士学位论文

零售业 O2O 系统的性能测试与优化

硕 士 研 究 生：李俊楠

学　　号：1140379068

导　　师：吴刚副教授

申 请 学 位：工程硕士

学　　科：软件工程

所 在 单 位：电子信息与电气工程学院

答 辩 日 期：2017 年 1 月

授予学位单位：上海交通大学

Dissertation Submitted to Shanghai Jiao Tong University
for the Degree of Master

Benchmark and Optimization Implementation for O2O Commercial System

Candidate：	Li Junnan
Student ID:	1140379068
Supervisor：	Prof. Wu Gang
Academic Degree Applied for：	Master of Engineering
Speciality：	Software Engineering
Affiliation：	School of Electronic Information and Electrical Engineering
Date of Defence：	Jan, 2017
Degree-Conferring-Institution：	Shanghai Jiao Tong University

上海交通大学

学位论文原创性声明

本人郑重声明：所呈交的学位论文《零售业 O2O 系统的性能测试与优化》，是本人在导师的指导下，独立进行研究工作所取得的成果。除文中已经注明引用的内容外，本论文不包含任何其他个人或集体已经发表或撰写过的作品成果。对本文的研究做出重要贡献的个人和集体，均已在文中以明确方式标明。本人完全意识到本声明的法律结果由本人承担。

学位论文作者签名：

日期：　　年　　月　　日

上海交通大学

学位论文版权使用授权书

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

保密□，在___年解密后适用本授权书。

本学位论文属于

不保密□。

（请在以上方框内打“√”）

学位论文作者签名：　　指导教师签名：

日期：　　年　　月　　日　　日期：　　年　　月　　日

零售业 020 系统的性能测试与优化

摘 要

2015《互联网+流通行动计划》明确指出：支持大型实体零售企业利用电子商务平台开展网上订货实体店取货等业务。“互联网+零售”已经处于国家级战略高度。因此，许多传统零售企业纷纷开始布局进军电子商务领域，探索转型 O2O，实施“互联网+”行动计划，谋求转型升级。

探索转型 O2O，升级 O2O 交易系统需要实现以下三个目标：第一、需要满足复杂的业务场景：O2O 模式是线下零售与线上电商的全面融合，业务更为复杂。第二、需要支持高并发高可用：O2O 交易系统与传统线下零售系统相比，需要保持对线上 7*24 小时的业务支持以及对线上用户高并发的支持。第三、需要兼顾低成本易扩展：传统零售业转型 O2O 仍处于发展初期，缺少大量资金的投入以及对技术的积累，需要降低对昂贵硬件以及复杂技术的依赖，并在满足当前性能需求的基础上注意系统的扩展能力。

许多零售企业选择在原有的线下零售系统基础上进行业务升级成 O2O 交易系统，例如汇金百货。但是，这大多只是进行了业务上的升级而没有进行系统架构方面的改进。传统的线下零售系统架构设计方案，过于注重功能的完整性以及数据的强一致性，而缺少对高并发以及用户体验的考虑，当面对线上多用户的高并发请求时，存在许多的系统性能瓶颈。

本文针对以上所述问题，对汇金百货业务升级后的 O2O 系统进行了全面的性能测试，在测试结果的基础上，对当前系统架构的性能瓶颈进行了分析，并针对这些当前存在的系统瓶颈，进行了对应的优化与实现。第一、优化升级了系统的服务器架构，在单机上实现了负载

均衡模型，合理利用了系统服务器的全部硬件资源。第二、优化升级了系统的存储层架构，在此前存储层的基础上，加入了一层缓存层来对于高并发的读请求提供更好的性能。第三、优化升级了当前的订单库存系统，在原有实现的基础上，对订单流程和事务设计进行了改进与实现。

本文通过性能对比测试的方式，从服务器架构、存储层架构、订单库存系统三个方面对比测试传统系统和优化升级后系统的性能指标。实验结果表明，本文的优化设计与实现有效地提升了系统的并发性、可用性，并很好地实现了前文提到的三个目标。

关键词：O2O，性能测试，架构，系统优化，高并发

Benchmark and Optimization Implementation for O2O Commercial System

ABSTRACT

2015 “Internet+ circulation action plan” clearly pointed out that, support large-scale retail enterprises using e-commerce platform for shopping online and picking offline. “Internet plus retail” has been in the national strategic height. Therefore, many traditional retail enterprises have begun to layout into the field of e-commerce, to explore the transformation of O2O, the implementation of “Internet+” action plan, seeking transformation and upgrading.

Explore the transformation of O2O, upgrade O2O trading system needs to achieve the following three objectives: first, the need to meet the complex business scenarios: O2O model is a comprehensive integration of online and offline retail electricity supplier, the business is more complex. Second, the need to support high concurrency and high availability: O2O trading system compared with the traditional offline retail system, need to maintain the online 7*24 hours of business support and high concurrent support for online users. Third, the need to take into account the low-cost easy to expand: the traditional retail industry transition O2O is still in the early stages of development, the lack of large capital investment and the accumulation of technology makes the need to reduce dependence on expensive hardware and complex technology, and pay attention to the system expansion ability on the basis of the current performance requirements.

Many retailers choose to upgrade their O2O trading systems based on existing offline retail systems, such as Huijin Department Store. However, most of this is only a business upgrade without the improvement of the system architecture. The traditional offline system architecture design

scheme, too much emphasis on functional integrity and data consistency, but the lack of high concurrency and user experience considerations, when the face of online multiple users with high concurrent requests, there are many system performance bottlenecks.

In view of the above mentioned problems, this paper made a comprehensive benchmark test on the business upgraded O2O system, on the basis of the test results, analyzes the current system architecture performance bottlenecks, and aiming at the existence of the current system bottlenecks, has carried on the corresponding optimization and implementation. First, optimize and upgrade the server architecture of the system, in a single machine to achieve a load balancing model. Second, optimize and upgrade the system's storage architecture, based on the previous storage layer, adding a layer of cache for high concurrent read request. Third, optimize and upgrade the current inventory system, the order flow and transaction design are improved and realized on the basis of the original implementation.

In the end of this paper, we compare the performance of the traditional system and the optimized system from three aspects: the server architecture, the storage architecture and the order inventory system. The experimental results show that the optimized design and implementation of the system can effectively improve the concurrency and availability of the system, and achieve the three goals mentioned above.

KEY WORDS: O2O, benchmark, architecture, system optimization, high concurrency

目 录

图 录

表 录

第一章 绪论

1.1 研究背景及意义

从 80 年代中期开始，零售业开始进入信息化的时代，POS 机、条形码、基于 POS SERVER 的 MIS 系统，线下销售系统广泛进入零售行业，这些零售交易系统已经发展较为成熟，可以很好的满足线下交易管理的需求。然而，随着互联网的发展，电子商务增速惊人。与此同时，中国实体经济中社会消费品零售总额却呈下降趋势。2015《互联网+流通行动计划》明确指出：支持大型实体零售企业利用电子商务平台开展网上订货实体店取货等业务。“互联网+零售”已经处于国家级战略高度[1]。因此，许多传统零售企业纷纷开始布局进军电子商务领域，探索转型 O2O，实施“互联网+”行动计划，谋求转型升级[2]。然而，线下零售业转型 O2O 将会面临前所未有的困难，这些困难主要涉及到技术、业务两个方面：

1）从业务方面来说，O2O 不仅仅是简单的把商品放到线上买，而是要借助互联网这个平台，将线上和线下有机的结合起来。完整的 O2O 交易场景不应该再有单独的线上和线下概念，而是一个整合起来的平台[3]。对于浏览商品，可以选择任意时间任意地点在 APP 浏览，也可以选择到现场获得更直观的商品体验；下单可以选择通过 APP，也可以通过柜台营业员；结账可以通过 APP 自助，也可以去商场收银台选择现金或者刷卡。由此可见，通过 O2O 搭建起了一个完整的融合的线上线下购物平台，提供给了用户全新的购物体验，而不仅仅只是多提供了一个线上购物的选择。

2）从技术方面来说，现有的大多数线下零售系统大都属于传统软件，较为注重功能的完整性、一致性，而不考虑系统的并发性、拓展性与可用性。相对而言，当系统扩展到 O2O 提供线上平台之后，由于需要直接面对用户，因此应该关注用户体验的问题。随着 O2O 业务的不断拓展，线上用户数目会不断增长，这时对系统的并发性就会提出更高的要求。当前成熟的电商系统都提供了良好的用户体验，具有很好的并发性。但是，这些电商系统的核心架构都是公司的技术机密，并且这些架构都依赖于庞大的硬件，私有中间件[4]以及大量技术人员的巨资投入。这些对于初期探索转型 O2O 的传统企业都是不具备的。同时，由于传统线下零售系统的存在，转型 O2O 并不仅仅是重新开始设计实现一套新的系统，而是要在线下零售系统的基础上进行业务的升级改造。由此，如何在控制成本的基础上对传统

线下零售系统进行架构和业务上的升级，搭建出适用于线下零售业的 O2O 交易系统则是一个巨大的挑战。

综上所述，线下零售转型 O2O 并不只是简简单单的搭建出一套线上电商系统，无论从业务角度还是技术角度都有很大的不同。因此，本文将从传统的线下零售系统出发，研究出如何在控制成本的条件下优化升级成一个高并发，高可用，易拓展的 O2O 交易系统。

1.2 研究内容

本文以汇金百货的线下零售交易系统进行业务升级后的 O2O 交易系统为背景，研究了 O2O 交易系统高并发，高可用、易拓展的架构优化升级与实现。具体研究内容包括：

1）研究总结了线下零售系统所使用的 Node.js 编程语言的特点，包括：Node.js 的平台架构、Node.js 的 I/O 模型以及 Node.js 的单进程模型。接下来，从数据存储的角度，研究了当前使用的关系型数据库 MySQL 以及基于高并发、分布式、海量数据为目标设计的 NoSQL，包括 MySQL 的视图，NoSQL 的概念、特性和分类，为后续的研究实现做理论基础。

2）利用压力测试工具，对进行了业务升级后的 O2O 交易系统进行了全面的性能测试，通过对测试结果的分析，总结出了业务升级后的 O2O 交易系统面临的关键性的系统瓶颈，主要包括以下三个方面：服务器架构，存储层架构以及订单库存系统。

3）针对系统的问题所在，研究优化、升级系统的设计与实现。第一、优化升级了系统的服务器架构，在单机上实现了负载均衡模型，合理利用了系统服务器的全部硬件资源。第二、优化升级了系统的存储层架构，在此前存储层的基础上，加入了一层缓存层来对高并发的读请求提供更好的性能。第三、优化升级了当前的订单库存系统，在原有实现的基础上，对订单流程和事务设计进行了改进与实现。

4）对于改进后的系统，再次利用压力测试工具，对系统进行全面的性能测试，通过对比测试结果，验证了系统优化升级方案的有效性。

1.3 论文组织结构

本文总共分为六个章节，每章内容如下：

第一章：绪论。首先介绍了课题的研究背景及意义，接下来阐明了本文的研

究内容，并对本文的组织架构进行了介绍。

第二章：相关技术研究。研究了 Node.js 编程语言，包括其架构，I/O 模型以及单进程服务器模型，此外还研究了存储层技术，包括关系型数据库 MySQL 及其视图，以及当前流行的针对高并发分布式设计的 NoSQL 数据库概念、特性与分类。

第三章：系统分析与测试。针对业务升级后的 O2O 交易系统进行了全面的系统测试，通过对测试结果的分析找出了当前系统存在的性能瓶颈。

第四章：优化设计与实现。针对从测试中暴露的系统瓶颈，分别从服务器架构、存储层架构以及订单库存系统三个方面进行了优化升级实现。

第五章：实验验证。对于优化升级后的系统，再次进行了全面的性能测试，通过测试结果的对比，验证了优化方案的有效性。

第六章：总结与展望。总结了该课题的研究成果与结论，并且对未来的工作进行了展望。

第二章 相关技术研究

2.1 Node.js

2.1.1 Node.js 架构

Node.js 是一个基于 Google V8 JavaScript 引擎的 JavaScript 运行环境[5]。Node.js 使用了一个事件驱动[6]，非阻塞的 I/O 模型来达到轻量和高效的目标。作为一个异步事件驱动的 JavaScript 运行环境，Node.js 被设计用来搭建高并发可伸缩的网络应用。如图 2-1 是 Node.js 系统架构图。最上层是利用 JavaScript 实现的 Node.js 标准库，面向 Node 应用程序提供应用层接口。在标准库之下，Node 使用 Google V8 引擎作为 JavaScript 的解释器，并且还提供了 libuv 来作为抽象封装层，从而使得 Node.js 可以兼容 Windows 平台和*nix 平台。在*nix 平台下使用了 libev 来提供高性能的事件循环，用于事件驱动的网络编程，使用了 libeio 来提供异步的文件 I/O 操作。而在 Windows 平台下，使用了 Windows 提供的一种独有的内核异步 I/O 方案：IOCP[7]。

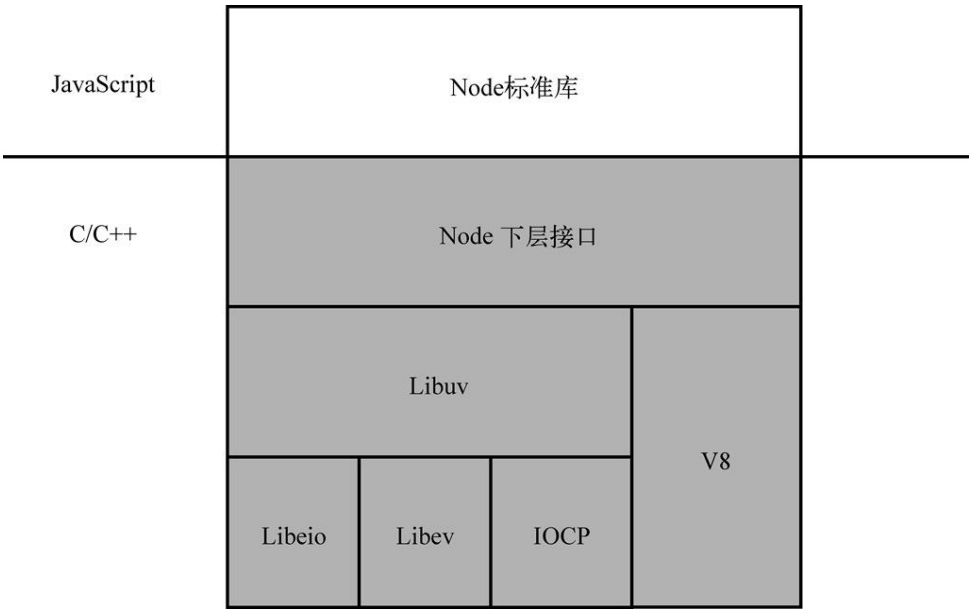

图 2-1 Node.js 系统架构图
Fig.2-1 Node.js system architecture

2.1.2 Node.js 的 I/O 模型

Node.js 创始人 Ryan Dahl 认为高性能 Web 应用服务器的关键就在于事件驱动和非阻塞 I/O，理论基础就在于 I/O 的代价是昂贵的，尤其是磁盘 I/O 和网络 I/O。Ryan Dahl 曾经在 JSConf 大会上对不同介质的 I/O 性能进行过对比[8]，见表 2-1，由此可见，磁盘和网络的 I/O 相比于 cache 和内存有着几十万倍的性能差距，和 CPU 的处理能力更是严重不匹配。然而，当前大量的互联网 Web 应用都是 I/O 密集型的应用，需要频繁的访问网络，磁盘，数据库等等。那么如何提升 I/O 的性能，如何在 I/O 的过程中提高 CPU 的使用率则是提升服务器性能，打造高并发 Web 应用的关键。目前随着技术的发展，出现了多种 I/O 模型：阻塞 I/O 模型，非阻塞 I/O 模型，I/O 多路复用模型和异步 I/O 模型。

表 2-1 不同介质的 I/O 性能对比
Table 2-1 Comparision of I/O performance of different media

I/O	CPU CYCLE
L1-cache	3
L2-cache	14
RAM	250
Disk	41000000
Network	240000000

1）阻塞 I/O 模型：阻塞 I/O 模型是一种直观、常用的 I/O 模型，在进行 I/O 操作的时候，调用者线程会处于阻塞状态，等待 I/O 操作完成。由表 2-1 可以看出，一次磁盘 I/O 会消耗掉 41000000 个 CPU 循环，网络 I/O 则更多，因此，如果 CPU 在此时等待调用线程则会浪费掉大量宝贵的 CPU 资源。为了提高资源利用率，阻塞 I/O 模型常常与多线程编程模型一起使用。在多线程编程模型中，每个线程负责处理一个或者多个 I/O 事件，当该线程执行 I/O 操作时，会被置为阻塞状态，等待 I/O 操作的完成，此时操作系统会去调度其它处于就绪状态的线程交由 CPU 执行。在 I/O 操作执行完成之后，调用线程会重新变成就绪等待状态，等待被操作系统调度重新执行。多线程模型简单，直观的解决了 I/O 并发的问题。而该模式最大的弊端在于资源开销过大。我们使用线程的时候首先是创建一个线程，线程创建好以后就是使用它，在线程执行完它的工作后就要销毁它。而在请求处理的场景中，我们会发现线程的创建过程和线程的销毁过程其实和处理请求的逻辑无关，而线程创建和销毁的时间也会统计到请求的处理过程中。在 64 位操作系统中，一个线程默认的堆栈大小为 8M，如果该服务器同时处理 10000 个并发请求，那么

创建出来的线程就要消耗掉 80G 的内存，这对系统资源是极大的消耗。同时，创建和销毁线程以及线程间的切换也会占用大量 CPU 的处理时间，使得业务处理单元无法获得足够的 CPU 执行时间，造成了很多无谓的系统资源消耗。为了对无谓的系统资源消耗进行优化，产生了线程池技术。在系统启动的时候，就会预先创建一部分线程在线程池中，当客户端的请求到来时，会从线程池中选出一个线程来负责处理，处理完成后再放回线程池中。通过线程池技术，虽然优化了线程创建和销毁的时间，但是当大量并发请求到来超过线程池中可以提供的线程数量时，工作线程无法处理这些请求，必须将无法处理的请求进行排队，或者消耗系统资源来创建更多线程。因此当并发量过大时，多线程模型仍然无能为力。同时，随着并发量的增大，多线程之间会对同一个临界资源进行访问，这时就需要一定的同步机制，例如锁或者信号。而多线程之间对于锁的竞争，又会导致系统并发能力的下降。

2）非阻塞 I/O 模型：在阻塞 I/O 模型中，虽然我们通过多线程提高了 CPU 的使用率，但是当线程等待 I/O 请求的结果时，线程会一直处于等待状态，此时线程处于闲置状态却无法被使用。而我们分析下网站请求的特点，可以发现大量的网站请求都包含两个步骤，分别是 I/O 操作和 CPU 操作，这时很慢的磁盘或者网络 I/O 就会大大影响到该线程的执行效率。如果我们可以把 I/O 操作和 CPU 操作分开，那么负责 CPU 操作的线程就可以获得很高的执行效率。由此我们得到了非阻塞 I/O 模型，在进行 I/O 操作时，调用者线程并不会阻塞等待 I/O 操作结果，而是会立即得到一个返回值，通过返回值可以得到当前 I/O 操作的状态[9]。通过这种方式，同一个线程可以处理更多的请求，不用再创建大量的工作线程同时也减少了因此带来的切换开销，这种方式极大的提高了系统的性能。然而，非阻塞的 I/O 需要自己不断去轮询检查 I/O 操作是否已经完成，那么就有可能会重复检查一个未完成的 I/O 多次，造成 CPU 的浪费，也可能出现一个 I/O 已经完成却迟迟没有检查完成状态，造成响应时间过大。

3）I/O 多路复用模型：为了解决非阻塞 I/O 模型中需要不断轮询检查 I/O 状态的问题，产生了 I/O 多路复用模型。多路复用是指使用一个线程来检查多个文件描述符（Socket）的就绪状态，传入多个文件描述符，如果其中有一个文件描述符处于就绪状态，则返回，通知用户线程读取数据[10]。I/O 多路复用模型解决了一个连接占用一个线程的问题，达到了在使用一个线程同时处理多个 I/O 请求的目的。它将等待 I/O 就绪的操作交给了操作系统去完成，而它只用等待操作系统返回 I/O 就绪的结果，然后自己或者启动一个新线程去读取数据。当然，I/O 多

路复用模型也有其缺点。它需要在编程时将业务逻辑进行分隔，把 I/O 事件划分出来并注册到 I/O 复用的监听集上。这样就导致原本的程序逻辑变得分散，大大增加了编程难度。

4）异步 I/O 模型：真正的异步 I/O 模型需要操作系统更多的支持。在 I/O 多路复用模型中，事件循环将文件句柄的状态事件通知给用户进程，然后由用户进程自己读取数据，对数据进行处理。而在异步 I/O 模型中，当用户线程收到事件处理完毕的通知时，用户线程所读取的数据已经被内核读取完毕并放在了用户线程指定的缓冲区内，用户线程可以直接使用数据[11]。

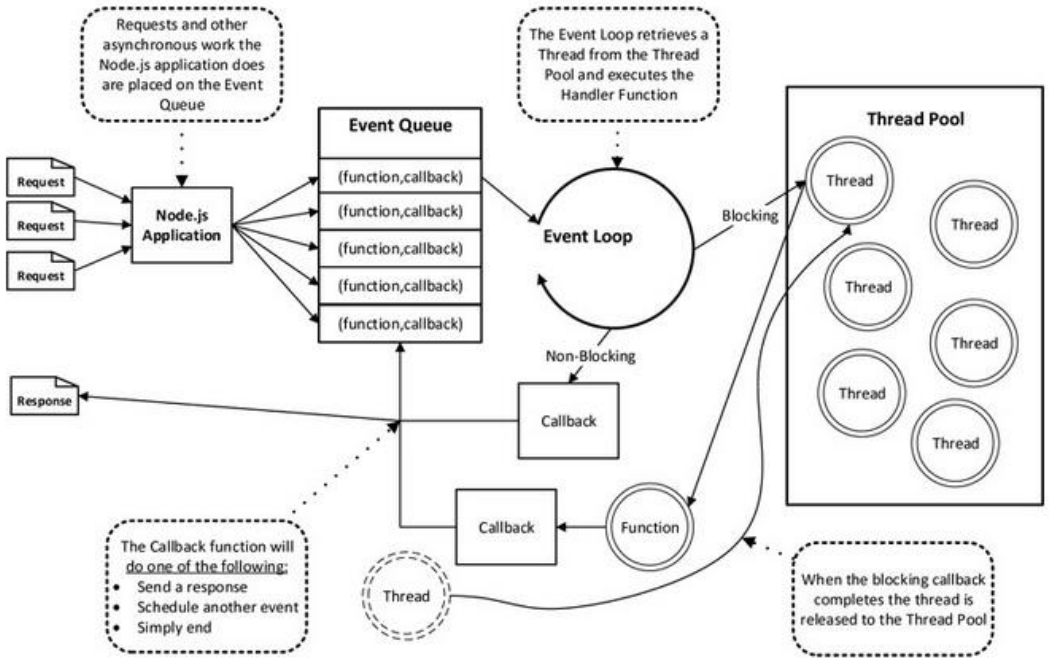

图 2-2 Node.js 事件驱动 I/O 模型[12]
Fig.2-2 Node.js event-driven I/O model

Node.js 高性能的关键就在于使用了基于事件驱动的异步 I/O 模型[13]，如图 2-2 所示。所谓事件驱动，就是指通过状态或事件的变化来对应用程序的流程进行控制[14]。一般通过监听事件来实现，一旦监听到事件的状态发生变化则调用注册的回调函数来处理。当一个新的请求到来时，请求会被加入到事件队列，然后一个循环会不断的来检测事件队列中的状态变化，如果检测到其中事件的状态变化，则执行该事件对应的处理代码，通常都是注册的回调函数。在 Node.js 的事件驱动模型中，每一个 I/O 请求被添加到事件队列（event-queue）中，Node.js 主线程通过一个事件循环（event-loop）来循环检查事件队列中是否有未处理的事件，当执

行过程中遇到 I/O 任务时，Node.js 会通过异步 I/O 库（libuv）交给操作系统处理，留下一个处理结果的回调函数，而不会阻塞等待结果，继续处理事件队列中下一个事件。当阻塞的 I/O 事件处理完毕后，会将回调函数添加到事件队列中，等待 Node.js 主线程在下一个事件循环中来进行处理。

2.1.3 Node.js 的单进程模型

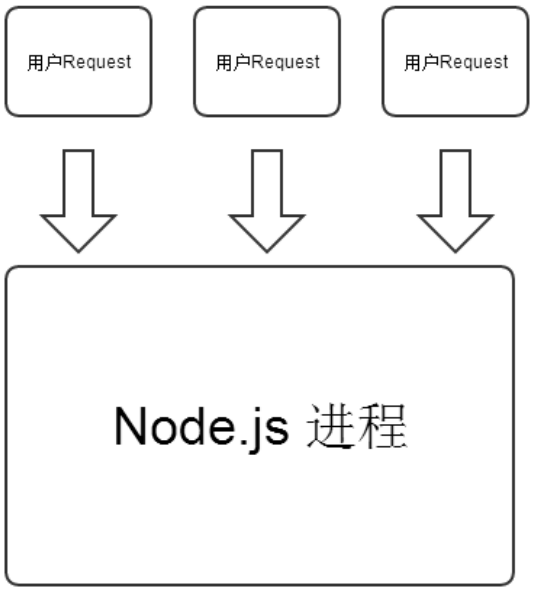

图 2-3 Node.js 单进程模型
Fig.2-3 Node.js single process model

Node.js 为了打造高性能的 web 服务器，采用了单进程单线程的服务器模型。如图 2-3 所示是 Node.js 单进程服务器模型。所有的用户请求达到 Node.js 服务器之后，全部由单进程 Node.js 的主线程接收并处理。如图 2-4 所示则是传统常见的 PHP + Apache 多线程模型。当每个用户请求到来之后，Apache 服务器会启动一个对应的线程来单独为一个用户请求服务。

通过单进程模型，Node.js 节省了创建大量线程的系统资源消耗，也节约了 CPU 在多线程间的的切换开销以及多线程对临界资源的竞争问题。但是 Node.js 的单进程模型并不是完美的，也会有相对于多线程模型的缺点：

- 一旦某个用户请求输入了某些非法参数，导致 Node.js 主进程出现异常退出，那么整个 Node.js 服务就会退出造成服务的不可用。而多线程模型中只会造成当前线程的异常退出，而不会影响到其它用户和整个系统服务。因此，Node.js

单进程模型对程序的健壮性有极高的要求。

- 只使用了单个 CPU，在当今的多核 CPU 时代是对 CPU 资源是极大的浪费。比如在一个 4 核 CPU 的服务器上，当 4 个用户请求到来的时候，多线程模型会启动 4 个线程分别运行 4 个 CPU 上，而 Node.js 的单进程模型则只能使用到其中一个 CPU 核心，通过事件循环来处理 4 个用户请求。
- CPU 密集运算会导致对 CPU 的长时间占用，从而影响到对其它请求的处理。多线程模型中，由于操作系统对于多线程之间的调度，多个线程可以近乎平均分配到 CPU 的使用权。而 Node.js 的单进程模型，会导致 CPU 资源被一个 CPU 密集型任务长时间占用，这时事件循环中后面的任务只能排队等待，哪怕异步 I/O 操作已经完成。因此，Node.js 的单线程模型也会出现由于前面请求的排队，而导致后面的请求响应时间越来越长的情况。而多线程模型由于操作系统对 CPU 使用权的分配，大量请求的响应时间会比较平均。

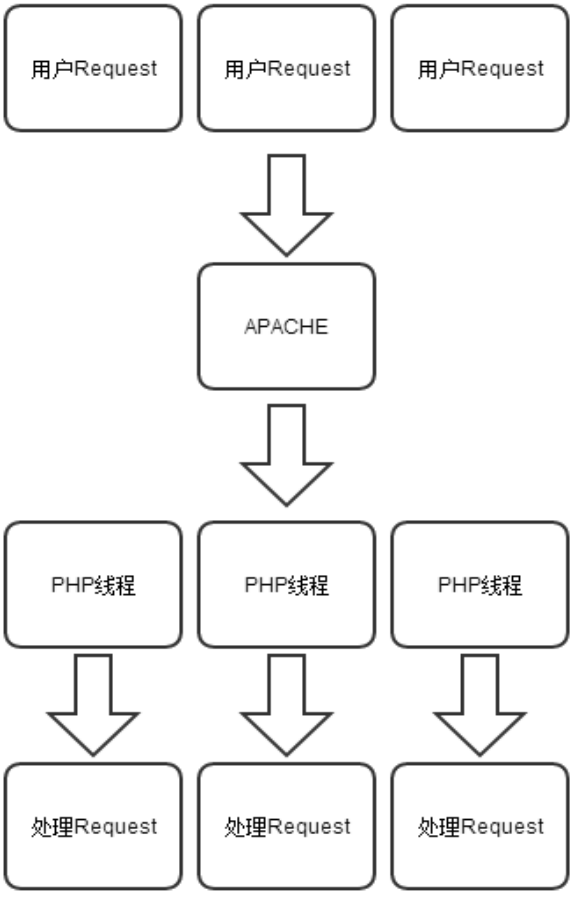

图 2-4 PHP + Apache 多线程模型
Fig.2-4 PHP + Apache multithreading model

2.2 关系型数据库与 NoSQL

2.2.1 MySQL 与视图

关系型数据库是指采用了关系模型来进行数据管理的数据库。关系模型是1970 年由 IBM 的研究员 E.F.Codd 博士提出的[15]，在之后的几十年中，关系模型的概念不断的发展并逐渐成为数据库结构中的主流模型。简单来说，关系模型可以理解成为二维表格模型，而大量二维表及其之间的联系所组成的一个数据组织就是关系型数据库。常见的主流数据库有 IBM DB2，ORACLE，Sybase，SQL server，MySQL 和 Access 等。

MySQL 是一个开源的关系型数据库管理系统，由瑞典 MySQL AB 公司开发。在 2008 年 1 月 16 号被 Sun 公司收购。由于其轻量级，速度快，成本低，尤其是开源的特性，受到了大部分互联网企业和中小型网站系统的青睐。

数据库视图是一个从一张或几张数据库表或视图中导出的虚拟表，它的作用类似于对数据表进行筛选，必须使用 SQL 语句中的 SELECT 语句实现构成。和真实的数据库表一样，视图包含一系列带有名称的列和行数据。但是，视图与真实表不同，并不在数据库中储存视图的数据。在定义视图时，只是把视图的定义存放在数据库，直到用户对视图进行查询时才会根据视图进行数据的查询并返回。视图主要有以下几个特点：

- 简化数据操作：用户可以将经常使用的连接，联合查询以及选择查询定义为视图，这样在每次查询的时候，只需要简单的查询视图即可。视图可以隐藏表表间的复杂关系，使用户将注意力集中在所关心的数据上，简化用户的数据查询操作，减少复杂的 SQL 语句，增强可读性。
- 权限控制：视图可以作为安全机制，通过创建视图，并将对应的用户权限与视图绑定，可以将数据库的权限控制限定到特定行和特定列，弥补了数据库功能上的不足。
- 保持程序逻辑独立性：数据的逻辑独立性是指用户的应用程序不会受到数据库结构或表重新构造的影响，如增加新的关系或者对原有关系增加新的字段。当开发中遇到灵活性的功能需求，需要改动数据库表导致大量的工作量时，可以通过视图来作为虚拟表对上层应用程序隐藏修改细节。或者当表中的数据量太大需要对表进行拆分时，可以通过使用视图来屏蔽实体表之间的逻辑关系，来构造上层应用程序所需要的原始表关系。

  MySQL 的视图有三种类型：MERGE,TEMPTABLE,UNDEFINED[16]。所谓

MERGE 类型，就是合并执行的方式，每当执行的时候，先将视图的定义语句和查询视图的 sql 语句合并成一条整体的 sql 语句，最后执行得到查询结果。而 TEMPTABLE 即临时表，每当查询的时候，会先根据视图的定义使用 select 语句生成一个结果的临时表，然后在临时表的基础上对查询语句进行查询。对于 UNDEFINED，是 MySQL 默认的视图处理方法。MySQL 会自己确定使用哪种方式来进行处理，MySQL 倾向于选择 MERGE 算法，因为 MERGE 的执行效率更高并且 TEMPTABLE 类型的视图不支持对视图数据的更新。然而 MERGE 算法要求视图中的行和基表中的行具有一一对应的关系，如果无法满足该关系，那么必须使用 TEMPTABLE 类型。因此，MERGE 算法无法支持类似于 DISTINCT，GROUP BY，HAVING，UNION 或 UNION ALL，SUM()，MAX()等聚合函数，当上述条件出现的时候，只能选择 TEMPTABLE 类型的视图。在 MySQL 用户手册中，说明了目前的视图实现是未经过优化的，只有在使用 MERGE 算法的视图可以在查询时用上基表的索引，而 TEMPTABLE 算法的视图是无法利用上基表索引的，只能在临时表创建过程中利用索引，然后会对临时表进行全表扫描。

2.2.2 NoSQL 概念与特性

NoSQL 一词最早是在 1998 年由 Carlo Strozzi 提出来的，指的是他开发的一个没有 SQL 功能，轻量级的，开源的关系型数据库[17]。这个定义跟我们现在对 NoSQL 的定义有很大的区别，随着 NoSQL 的发展，我们发现我们需要的不是“no sql”，而是“no relational”，也就是非关系型数据库。2009 年初，Johan Oskarsson 举办了一场关于开源分布式数据库的讨论，Eric Evans 在这次讨论中再次提出了 NoSQL 一词，用于代表那些采用分布式设计，不使用关系模式概念，并且一般不保证遵循 ACID 原则的数据存储系统[18]。NoSQL 可以被理解为“non-relational”，也可以被理解为“Not Only SQL”。

传统的关系型数据库最大的特点就是事务的一致性，也就是数据库的 ACID 特性：Atomic（原子性），Consistency（一致性），Isolation（隔离性），Durability（持久性）。关系型数据库所具有的另一个特点是固定的表结构，所有的数据都是根据固定的表结构组织的。关系型数据库的优点有数据模型简单，数据独立性高，理论基础严格等。这对于早期的银行系统，财务系统，信息管理系统等对数据有强一致性要求的系统来说十分匹配。然而，随着 web2.0 的快速发展，移动互联网以及 SNS（Social Network Service）社交网络的兴起和发展，一致性不再是至关重要的指标，取而代之的是系统的并发量，响应时间等用户体验相关的指标。

但是，关系型数据库为了维护一致性而牺牲了读写性能，难以应对高并发的读写请求，同时，关系型数据库固定的表结构导致其扩展性太差，难以满足需求快速变动，不断迭代的新功能需求，最后，海量数据也对关系型数据库带来了极大的挑战，单机性能难以应对，需要分库分表搭建数据库集群来应对，然而，关系型数据库起始并不是针对分布式设计，导致了系统架构的复杂性和难扩展性。

这时，基于分布式，高并发，海量数据而设计出来的各种类型的 NoSQL 应运而生。NoSQL 的主要理论依据建立在 CAP 法则，Base 模型和最终一致性上。在 2000 年，Eric A.Brewer 教授在 PODC 大会上正式提出了 CAP 法则[19]，该法则认为一个分布式系统不可能同时满足一致性（Consistency）、可用性（Availability）和分区容忍性（Tolerance to network Partitions），最多只能同时满足两个。后来，Seth Gilbert 和 Nancy Lynch 对该法则的正确性进行了证明[20]。一致性指分布式系统中所有的数据备份，在同一时刻都是同一值。可用性指每个操作总能在确定的时间内得到响应，即系统任何时候都处于可用状态。分区容忍性指分布式集群中的某些节点不可用了，但是集群整体还是可以对外提供服务。CAP 法则要求分布式系统必须在一致性和可用性之间做出权衡，从而导致了最终一致性和强一致性两种选择。而 BASE 模型则是分布式系统牺牲强一致性做出的折衷，即：基本可用性（Basically Available）、软状态（Soft state）和最终一致性（Eventually consistent）[21]。BASE 模型允许一段时间内的状态不一致，只需最终满足一致性即可。

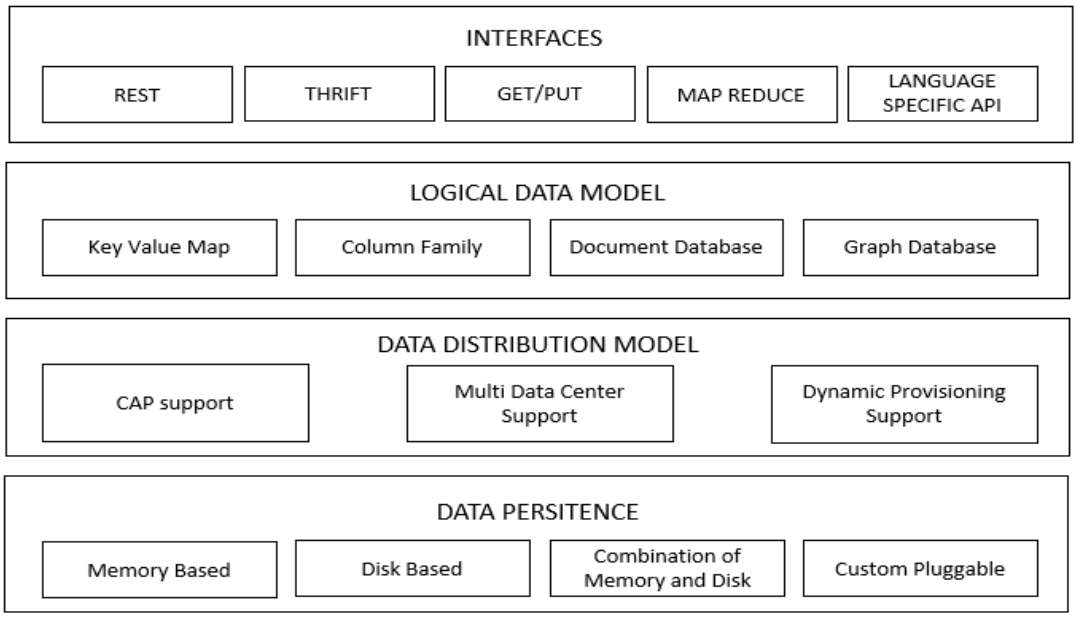

图 2-5 NoSQL 整体架构
Fig.2-5 NoSQL architecture

如图 2-5 所示是 Sourav Mazumder 提出的 NoSQL 整体架构[22]。Sourav Mazumder 将 NoSQL 分为四层：最上层是接口层，指 NoSQL 面向编程语言的接口，方式有 REST，Thrift，MAPREDUCE，GET/PUT 以及语言特定 API；在接口层下，是逻辑数据模型层，指的是数据库数据的逻辑模型，有键值对模型，列族模型，文档模型以及图模型；数据分布层在第三层，指的是数据在分布式系统中遵循的 CAP 理论，多数据中心支持和动态部署支持；最底层是数据持久层，有基于内存的，有基于硬盘的，还有硬盘内存结合持久化以及定制可插拔的选项。

2.2.3 NoSQL 分类

基于 NoSQL 体系架构中的 NoSQL 逻辑数据模型，可以对 NoSQL 分为四类：

1）键值对（Key Value）数据库：键值对模型类似于传统编程语言中哈希表数据结构。可以通过主键 Key 来添加，查询，删除数据。由于所有操作都是通过主键 Key 来进行，因此可以定位到需要操作的数据，有很好的性能表现。适用于关系型数据库的数据缓存，储存用户信息，例如配置文件，购物车等。由于键值对数据库只能通过主键来查询，因此不适用于需要复杂查询的场景，例如通过值来进行筛选查询。同样，由于键值对难以储存多数据间的关联关系，因此也不适用于数据间关联较多的场景。主要产品有：Amazon’s Dynamo、Memcached、Redis、Riak 等。

2）列族（Column-Family）数据库：列族数据库被设计用来处理海量数据，它提供了高性能的读取和写入操作以及服务的高可用性。谷歌为了满足其服务需求推出了 Bigtable[23]，Facebook 开发了 Cassandra 来支持其收件箱搜索服务[24]。列族数据库管理系统通常运行在由多个服务器组成的集群上。列族数据库适用于那些对数据库写操作有特殊要求的应用程序，数据在地理上分布于多个数据中心的应用程序以及可以容忍副本中存在短期不一致的应用程序。通常应用在一些需要这种大数据的处理能力的领域，比如对网络流量和日志数据进行安全性分析，使用交易数据进行股票市场分析，搜索引擎，社交网络服务等。主要产品有：HBase、Cassandra 等。

3）文档（Document）数据库：文档数据库以灵活性为标准设计，如果一个应用程序需要存储具有不同属性结构的大量数据，那么文档数据库可以很好的满足需求。文档数据库将数据以文档的形式存储，每个文档都是自包含的数据单元，是一系列数据项的集合。每个数据项都有一个键及其对应的值，值即可以是如字符串、数字等简单的数据类型，也可以是如有序列表和关联对象等的复杂数据类

型。文档数据库与键值对数据库的区别在于，文档数据库可以理解所存储数据中的文档内容，并且可以根据此来进行查询检索。通常用于具有大量读写操作，需要管理复杂数据类型和变量属性的系统。主要产品有：MongoDB、CouchDB、RavenDB 等。

4）图（Graph）数据库：图数据库以图的方式来存储管理数据。实体会被存储为图的顶点，而实体之间的联系则会被作为顶点间的边。图形数据库并不太关注数据的规模和可用性，而主要针对数据之间存在怎样的关系以及用户需要怎样执行计算任务。图数据库的常见用途包括地理空间计算、推荐引擎、网络分析等。主要产品有：Neo4J、Infinite Graph、OrientDB 等。

2.3 本章小结

本章主要介绍了课题研究中相关的概念和技术。首先介绍了基于 Google V8 Javascript 解释器的 Node.js 平台，包括：Node.js 的体系架构，Node.js 的 I/O 模型以及 Node.js 的单进程模型。接下来，介绍了数据库存储相关的技术，包括：关系型数据库 MySQL 及其视图，NoSQL 的概念和特性以及 NoSQL 的分类。

在研究学习了这些相关技术后，在下一章，我们将会首先介绍基于线下零售系统转型升级的 O2O 交易系统的系统架构，然后针对升级后的系统进行相关性能测试，找出系统当前存在的问题和性能瓶颈。

第三章 系统分析与测试

3.1 系统业务升级与现状

从 80 年代中期开始，零售业开始进入信息化的时代，POS 机、条形码、基于 POS SERVER 的 MIS 系统，线下销售系统广泛进入零售行业，这些零售交易系统已经发展较为成熟，可以很好的满足线下交易管理的需求。然而，随着互联网的发展，电子商务增速惊人。与此同时，中国实体经济中社会消费品零售总额却呈下降趋势。2015《互联网+流通行动计划》明确指出：支持大型实体零售企业利用电子商务平台开展网上订货实体店取货等业务。“互联网+零售”已经处于国家级战略高度。因此，许多传统零售企业纷纷开始布局进军电子商务领域，探索转型 O2O，实施“互联网+”行动计划，谋求转型升级。然而，O2O 的转型过程并不容易。由于正在使用的线下交易系统的历史包袱，同时兼顾到技术投入和人力成本等，并不能简单的重新搭建一套 O2O 系统，而是必须在保证线下系统正常良好运营的情况下来进行 O2O 的业务升级。因此，完成业务升级后的 O2O 交易系统，仍然沿用传统的系统架构，并没有考虑到 O2O 系统直接面对线上用户需要关注的用户体验问题，也没有考虑到线上互联网带来的大量用户并发对系统性能的挑战，存在很多的系统瓶颈。因此，我们需要在当前 O2O 系统的基础上，针对当前系统的性能瓶颈，提高系统的高并发高可用性，同时，兼顾系统的易扩展性，以应对后期系统用户日益增加所带来的系统性能压力，可以低成本的扩展系统的性能。

本文以徐家汇的汇金百货的交易系统为背景，展开课题研究。近两年，汇金百货积极探索转型 O2O，其线下零售交易系统（以下简称：RMS）是极具代表性的零售交易系统，每年支撑着汇金百货 30 亿的销售额。经过近两年的努力，目前已经完成了从线下交易系统到 O2O 交易系统（以下简称：MEC）的业务升级。在业务升级后，系统不仅覆盖了线下零售场景和线上电商交易场景，并且还将线下零售和线上电商场景相结合，打通了线上线下之间的阻隔，做到了完全的 O2O 融合。如图 3-1 所示是业务升级后的交易场景图，在购物流程中的选购、下单、支付、取货各个环节中，都可以进行任意结合。比如选购可以在任意地点通过 APP 浏览商品，也可以在商场现场浏览商品。选中心仪商品后，即可以直接 APP 直接下单，也可以由商场营业员帮助下单购买，同时在商场中如果营业员处人多需要排队，还可以通过手机 APP 自行扫码下单。在支付阶段，可以通过商场里的收银

POS 结算，也可以直接通过 APP 进行线上支付，亦可以在家等待货到付款。支付完成后，顾客可以选择商品柜台自提，也可以选择物流快递送货上门。

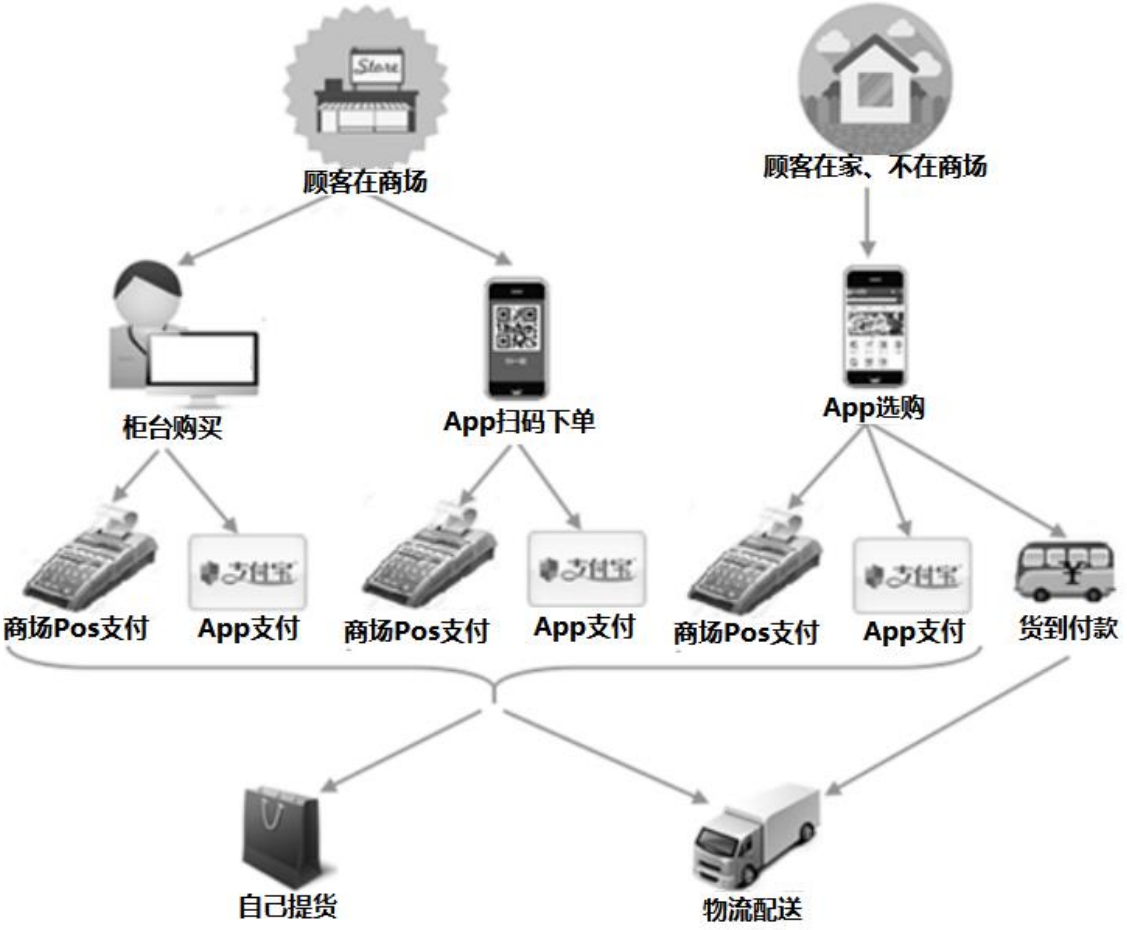

图 3-1 O2O 交易场景图
Fig.3-1 O2O Trading scenario

3.2 系统架构介绍

如图 3-2 所示为业务升级前的线下零售系统架构，在用户层提供了营业员和管理员的入口。在用户层之下，是系统的服务层。RMS 零售管理系统作为系统的主入口，与用户层进行交互。提供了商品管理系统，库存管理系统，订单系统，支付系统，账户管理系统，促销管理系统和售后管理系统。其中，搭建了 Drools 规则引擎作为系统的促销引擎，通过促销管理系统里设置的商城促销活动进行商品价格的计算。在存储层，采用了关系型数据库 MySQL 作为持久化方案。同时，由于 MIS 管理信息系统的存在，RMS 零售系统提供了与 MIS 系统的交互和同步接口，需要与 MIS 系统交互包括供应商，财务结算，会员折扣等信息。RMS 系统

需要同步 MIS 系统里的供应商信息来进行销售柜的管理，在促销计算的过程中，需要根据用户在 MIS 系统中的 VIP 信息来获取用户可以得到的优惠折扣进行促销计算。在每日运营结束后，需要将订单付款财务信息与 MIS 系统中的财务结算系统进行同步，在 MIS 中进行财务系统的管理。

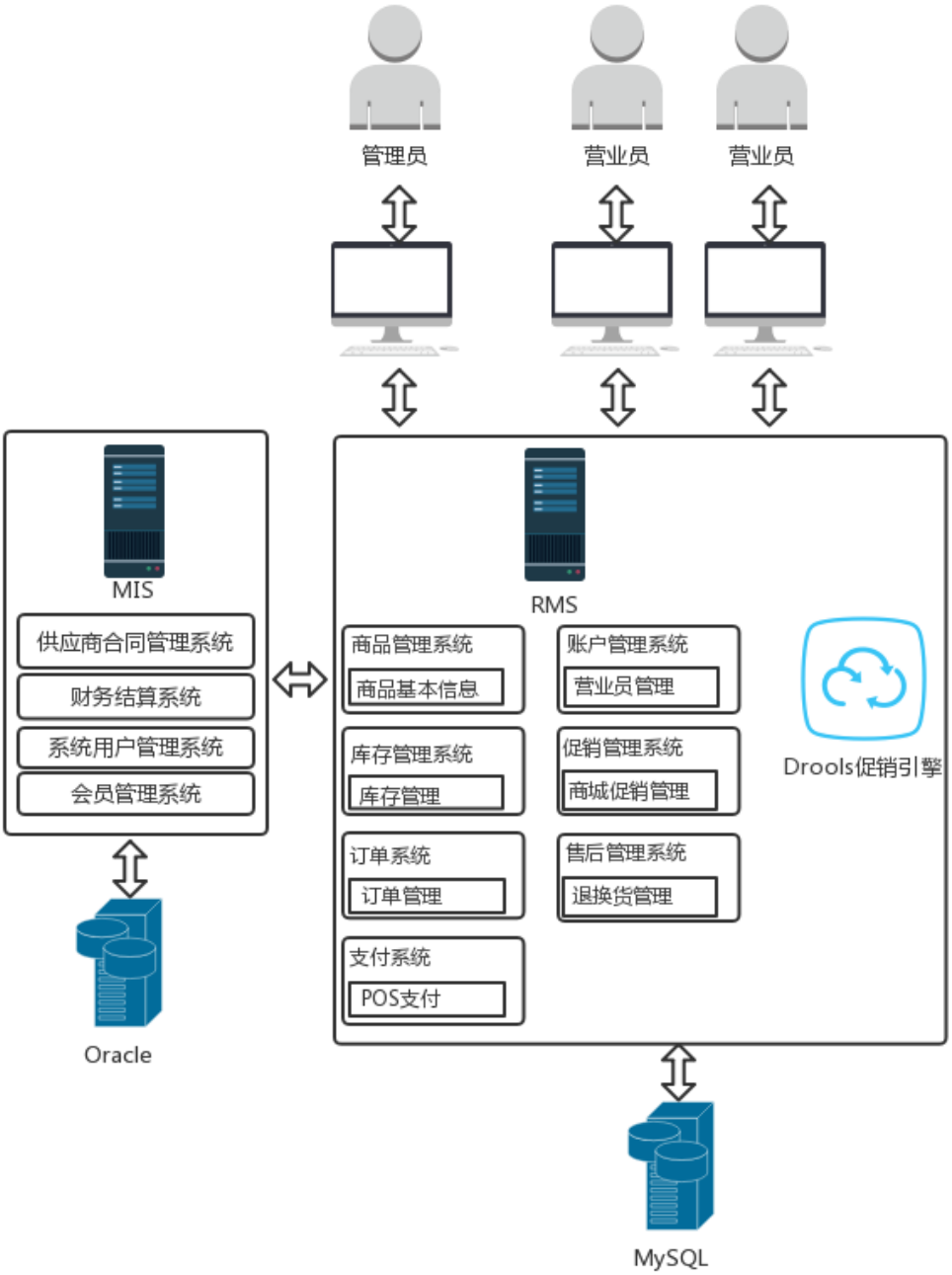

图 3-2 业务升级前系统架构

Fig.3-2 System architecture before business upgrade

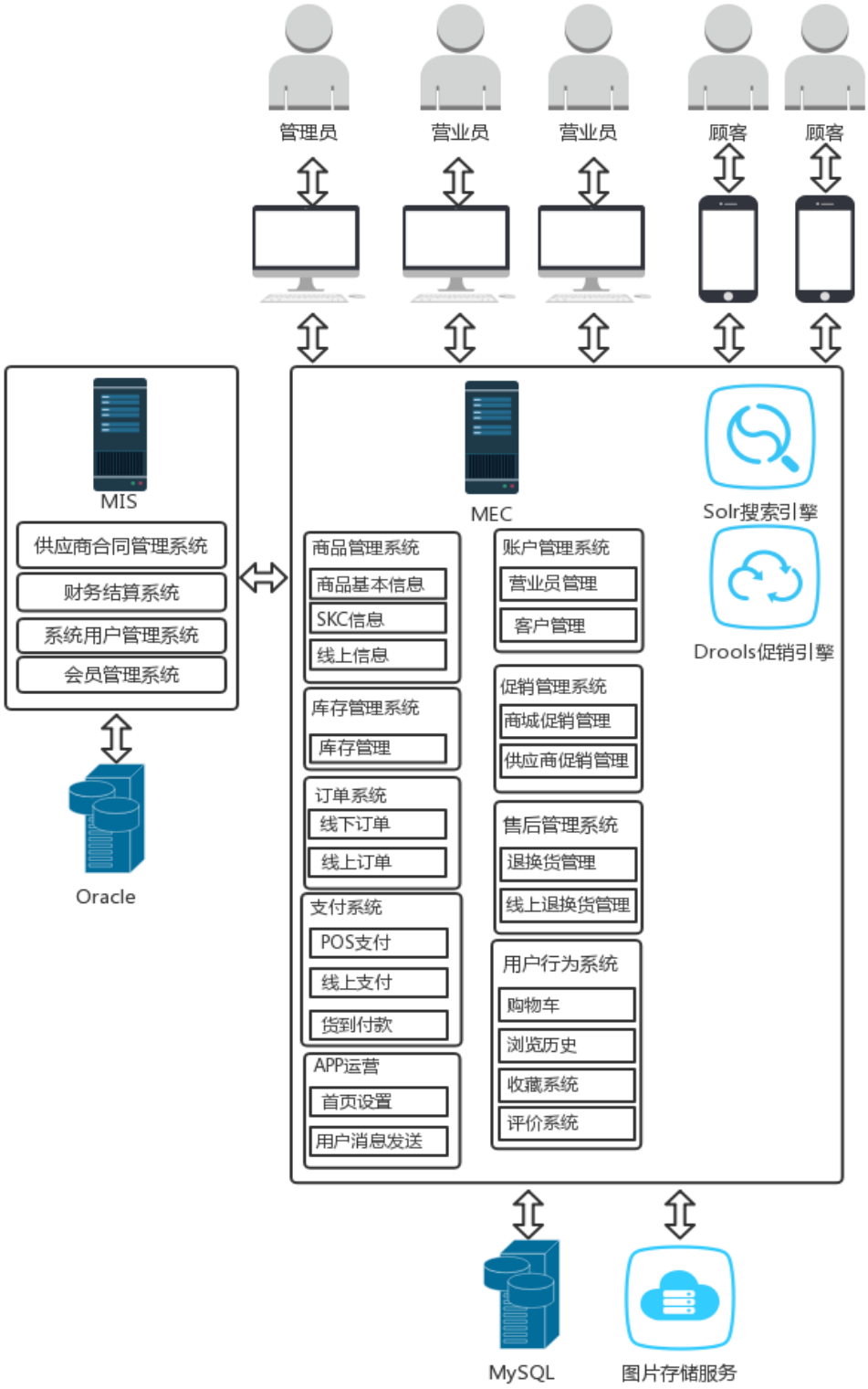

图 3-3 业务升级后系统架构

Fig.3-3 System architecture after business upgrade

如图 3-3 所示为业务升级后的系统架构，在不改变系统基本架构的情况下。在服务层中进行了业务相关的升级。在用户层，加入了 Android 和 IOS 的 APP，给顾客提供了购物入口。在服务层，在商品管理系统中添加了 SKC 信息，线上商品信息用于 APP 上商品信息的展示。在订单系统中，加入了线上订单的管理，并且把线上线下订单打通，达到 O2O 的目标。支付系统中加入了线上支付，货到付款两种支付方式的支持。账户管理系统中新加入了客户账户的管理。促销管理系统中，新加入了对供应商促销活动的支持。售后管理系统里，完善了线上退换货的管理。并且加入了线上相关的两个全新模块，APP 运营和用户行为系统。用来支持 APP 运营中包括首页活动设置，用户消息发送以及用户购物行为相关功能，例如购物车、浏览历史、商品品牌收藏以及购物评价等。同时，新搭建了 Solr 全文检索引擎来满足 APP 用户对于商品搜索的需求。在存储层，仍然沿用了 MySQL 的持久化方案，为了满足线上商品图片展示的需求，引入了新的图片云存储服务。

系统在业务功能上进行了相关的升级，并没有对系统整体架构进行升级调整。然而，传统的架构只是基于线下运行环境，基于功能性设计，并没有考虑到在升级转型到 O2O 系统后，线上大量用户一起涌入所带来的高并发压力。同时，并没有考虑到系统性能、响应时间对用户体验的影响。在系统升级上线之后，不断可以收到用户关于 APP 响应慢，用户体验差的抱怨和投诉。因此，我们迫切的需要优化系统架构，提高服务性能，改善用户体验差的问题。下面，首先会对系统进行性能基准测试。在测试的过程中了解系统目前的性能水平，根据测试结果找到目前系统主要的性能瓶颈和架构问题。

3.3 系统性能测试

3.3.1 测试计划

本次测试旨在通过对系统的 APP 接口进行全面的基准测试，了解系统目前的性能水平。从业务角度来说，用户使用 APP 的主要场景：打开 APP 进入首页，浏览下首页的滚动活动栏，看看是否有合适的活动。接下来，浏览下首页推荐的商品，看是否有心仪的。也有可能，通过 APP 的分类列表直接找到自己的目标分类，在目标分类商品列表中浏览自己喜好的商品。找到合适商品后，查看商品详情。如果用户选中了该商品，可能会选择收藏该商品，或者收藏该品牌，也可能加入购物车中，或者直接下单购买。在购物车中，用户可以一次选中多个心仪的商品，一起下单结算，享受到更多的优惠。在下单之后，用户可以直接付款，也可以稍

后去我的订单中查看订单详情，再决定是否付款。很多情况下，用户会去个人信息界面中查看自己以前的收藏，或者查看下自己的浏览历史，找一下自己以前看中的却没有购买的商品。因此，从业务的角度来说，根据用户的使用路径和习惯，总结出以下使用频率高，对用户体验至关重要的接口：

1. 打开 APP 加载首页
2. 获取 APP 首页滚动活动栏
3. 点击首页滚动活动栏，获取活动详情
4. 分类列表选择商品分类展示商品列表
5. 点击商品显示商品详情
6. 商品加入购物车
7. 获取用户购物车信息
8. 购物车结算
9. 结算完成后下单
10. 进入个人页面（读取收藏数，订单数，浏览历史数，待评论数）
11. 获取个人线上订单
12. 获取个人线下订单
13. 获取个人浏览历史

从技术角度来说，我们希望了解到这些接口在最佳性能情况下、一般使用的情况下、高并发的场景以及全负载的情况下分别的性能以及不同压力下对系统性能的影响。自业务升级 O2O 系统上线之后，目前系统注册用户数有 1 千多人，而汇金百货的 MIS 系统中有贵宾客户 6 万人，目前转化率不足 20%。随着 O2O 系统试运营结束，汇金百货进行正式的运营推广，目标达到 60%的转化率，同时加上新客户的转化，最终预计注册用户数可以达到 4 万人，活跃用户 5 千人。考虑到在节假日大促销例如圣诞元旦时某些热门商品可能有大量用户来秒杀抢购的情况，预计有 300 人同时抢购。因此，考虑系统的稳定性，设置系统在满负载的情况下有每秒 500 并发用户访问同一个接口，高并发下每秒有 100 用户，而一般使用场景下每秒有 10 个用户并发访问同一个接口，最佳性能情况下，设置每秒 1 个用户访问来测试该接口可以达到的最佳性能。

最终，制定出以下测试计划：分别从 4 个并发量的压力情况下，最佳性能（每秒 1 个用户）、正常使用（每秒 10 个用户）、高并发（每秒 100 个用户）以及全负载（每秒 500 个用户），对系统上述的 13 个接口进行性能测试。

3.3.2 测试环境

为了减少测试环境和生产环境的不同对测试结果造成的影响，本文在控制成本的情况下尽量模拟生产环境搭建了测试环境系统。在生产环境中，目前 MEC 系统运行在单台 DELL 服务器，在服务器上运行了 Node 主服务进程，同时，通过 Apache Tomcat 作为应用服务器，部署了 Solr 搜索引擎和 Drools 促销引擎。存储层的 MySQL 数据库单独部署在一台 DELL 服务器。在机房内网与 MIS 系统相连。在测试环境中，同样采用了两台服务器 A、B 模拟生产环境的情况，一台服务器 A 作为应用服务器部署了 Node 主服务进程以及通过 Apache Tomcat 部署了 Solr 搜索引擎和 Drools 促销引擎。另一台服务器 B 作为数据库服务器部署了 MySQL 数据库，同时，为了保证数据的真实性，从生产环境数据库导出了截至测试日期之前的全部生产环境数据到测试数据库。

在测试工具的选择上，目前主流的测试工具有 Apache ab，Apache JMeter，Tsung，LoadRunner 等。由于测试环境的限制，我们希望能够以尽量少的资源占用来达到尽量多的负载压力。因此，在众多测试工具中，选择了 Tsung 压力测试工具来作为测试工具。Tsung 工具基于 erlang 语言开发，erlang 语言就是面向高并发开发的语言[25]，因此天然具有高并发的优势，而且又使用了 epoll 技术，在一个进程中就可以管理上万数量级的 socket，相比于其它使用多线程的压力测试工具，打出相同负载压力的情况下，所占用的资源要少得多。由于机器资源紧张的原因，最终选择将 Tsung 部署在数据库服务器 B 上来进行压力测试。如表 3-1 所示为测试环境配置：

表 3-1 测试环境配置
Table 3-1 Test experiment settings

机器@IP	处理器	内存	硬盘	系统环境
A@202.120.40.150	Xeon E5310, 1.6GHZ*8	32G	1T	Node Server, Tomcat(Solr, Drools)
B@202.120.40.142	Xeon E5405, 2GHZ*8	32G	1.5T	MySQL, Tsung

3.3.3 测试过程

在测试环境部署好之后，开始根据指定的测试计划进行测试：

1. 打开 APP 加载首页：以 10s 为测试时长，分别每秒生成 1 个用户，10 个用户，100 个用户，500 个用户请求 APP 首页，进行 4 次测试，对比结果
2. 获取 APP 首页滚动活动栏：以 10s 为测试时长，分别每秒生成 1 个用户，

10 个用户，100 个用户，500 个用户请求获取 APP 首页滚动活动栏，进行 4 次测试，对比结果

3. 点击首页滚动活动栏，获取活动详情：以 10s 为测试时长，分别每秒生成 1 个用户，10 个用户，100 个用户，500 个用户点击请求首页滚动活动栏，首页活动栏共 5 个，考虑到第一个滚动栏可以获得更多用户的关注，因此假设 5 个活动栏被点击的概率分别为 30%，20%，20%，20%，10%，进行 4 次测试，对比结果
4. 分类列表选择商品分类展示商品列表：以 10s 为测试时长，分别每秒生成 1 个用户，10 个用户，100 个用户，500 个用户点击请求分类，从数据库中导出所有分类的 csv 文件，在测试脚本中每次随机取出一个分类并随机出一个小于 100 的任意页码进行请求，进行 4 次测试，对比结果
5. 点击商品显示商品详情：以 10s 为测试时长，分别每秒生成 1 个用户，10 个用户，100 个用户，500 个用户点击请求商品，从数据库中导出所有商品 ITEMCODE 的 csv 文件，在测试脚本中每次随机取出一个商品 ITEMCODE 进行请求，进行 4 次测试，对比结果
6. 商品加入购物车：以 10s 为测试时长，分别每秒生成 1 个用户，10 个用户，100 个用户，500 个用户将商品加入购物车，从数据库中导出所有商品 ITEMCODE 的 csv 文件，在测试脚本中每次随机取出一个商品 ITEMCODE,随机选出 1-5 个数量，进行加入购物车操作，进行 4 次测试，对比结果
7. 获取用户购物车信息：以 10s 为测试时长，分别每秒生成 1 个用户，10 个用户，100 个用户，500 个用户请求自己的购物车数据，从数据库中导出已注册的所有用户的 USERCODE，在测试脚本中每次随机取出一个用户 USERCODE，进行获取购物车的操作，进行 4 次测试，对比结果
8. 购物车结算：以 10s 为测试时长，分别每秒生成 1 个用户，10 个用户，100 个用户，500 个用户分别进行商品结算，从数据库中导出所有商品 ITEMCODE 的 csv 文件，在测试脚本中每次随机取出一个商品 ITEMCODE，随机选出 1-5 个数量，进行加入购物车操作，进行 4 次测试，对比结果
9. 结算完成后生成订单：每 5 秒生成 1 个用户持续 50 秒共 10 个用户，每秒生成 1 个用户，10 个用户持续 10 秒分别共 10 个，100 个用户进行商品结算，从数据库中导出所有商品 ITEMCODE 的 csv 文件，在测试脚本中

每次随机取出一个商品 ITEMCODE，随机选出 1-5 个数量，进行下单操作，进行 3 次测试，对比结果

10. 进入个人页面（读取收藏数，订单数，浏览历史数，待评论数）：以 10s 为测试时长，分别每秒生成 1 个用户，10 个用户，100 个用户，500 个用户请求自己的购物车数据，从数据库中导出已注册的所有用户的 USERCODE，在测试脚本中每次随机取出一个用户 USERCODE，进行获取个人页面信息的操作，进行 4 次测试，对比结果
11. 获取个人线上订单：以 10s 为测试时长，分别每秒生成 1 个用户，10 个用户，100 个用户，500 个用户请求自己的购物车数据，从数据库中导出已注册的所有用户的 USERCODE，在测试脚本中每次随机取出一个用户 USERCODE，进行获取个人线上订单的操作，进行 4 次测试，对比结果
12. 获取个人线下订单：以 10s 为测试时长，分别每秒生成 1 个用户，10 个用户，100 个用户，500 个用户请求自己的购物车数据，从数据库中导出已注册的所有用户的 USERCODE，在测试脚本中每次随机取出一个用户 USERCODE，进行获取个人线下订单的操作，进行 4 次测试，对比结果
13. 获取个人浏览历史：以 10s 为测试时长，分别每秒生成 1 个用户，10 个用户，100 个用户，500 个用户请求自己的购物车数据，从数据库中导出已注册的所有用户的 USERCODE，在测试脚本中每次随机取出一个用户 USERCODE，进行获取个人浏览历史的操作，进行 4 次测试，对比结果

3.3.4 测试结果

下面分别列出各个接口的测试结果：

1. 打开 APP 加载首页，见下表 3-2：

表 3-2 APP 加载首页测试结果
Table 3-2 APP get home page result

每秒请求用户数	总请求用户数	成功响应用户数	成功响应平均时间	最快成功响应时间	最慢成功响应时间
1	10	10	0.54sec	0.51sec	0.64sec
10	100	100	2.46sec	0.89sec	5.53sec
100	1000	1000	40.46sec	12.15sec	57.36sec
500	5000	2764	1min11sec	36.36sec	1min26sec

2. 获取 APP 首页滚动活动栏，见下表 3-3：

表 3-3 获取 APP 首页滚动活动栏测试结果
Table 3-3 APP get activity test results

每秒请求用户数	总请求用户数	成功响应用户数	成功响应平均时间	最快成功响应时间	最慢成功响应时间
1	10	10	28.17msec	25.16msec	30.63msec
10	100	100	27.63msec	24.63msec	32.57msec
100	1000	1000	1.81sec	1.17sec	3.34sec
500	5000	5000	8.78sec	5.93sec	13.28sec

3. 点击首页滚动活动栏，获取活动详情，见下表 3-4：

表 3-4 获取滚动活动栏详情测试结果
Table 3-4 Get activity detail test results

每秒请求用户数	总请求用户数	成功响应用户数	成功响应平均时间	最快成功响应时间	最慢成功响应时间
1	10	10	0.32sec	0.24sec	0.37sec
10	100	100	1.63sec	0.75sec	2.02sec
100	1000	1000	32.52sec	9.44sec	1min4sec
500	5000	3162	1min4sec	21.56sec	1min17sec

4. 分类列表选择商品分类展示商品列表，见下表 3-5：

表 3-5 商品分类列表测试结果
Table 3-5 Item category test results

每秒请求用户数	总请求用户数	成功响应用户数	成功响应平均时间	最快成功响应时间	最慢成功响应时间
1	10	10	0.27sec	0.13sec	0.35sec
10	100	100	0.55sec	0.22sec	1.18sec
100	1000	1000	8.21sec	4.63sec	11.46sec
500	5000	3481	36.76sec	13.01sec	53.18sec

5. 点击商品显示商品详情，见下表 3-6：

表 3-6 显示商品详情测试结果
Table 3-6 Item detail test results

每秒请求用户数	总请求用户数	成功响应用户数	成功响应平均时间	最快成功响应时间	最慢成功响应时间
1	10	10	0.71sec	0.63sec	0.78sec
10	100	100	3.14sec	2.03sec	4.34sec
100	1000	1000	1min4sec	12.81sec	2min1sec
500	5000	2237	1min28sec	28.57sec	2min15sec

6. 商品加入购物车，见下表 3-7：

表 3-7 商品加入购物车测试结果
Table 3-7 Add to shopping cart test results

每秒请求用户数	总请求用户数	成功响应用户数	成功响应平均时间	最快成功响应时间	最慢成功响应时间
1	10	10	26.35msec	25.77msec	26.94msec
10	100	100	37.19msec	20.60msec	63.23msec
100	1000	1000	1.65sec	1.38sec	1.95sec
500	5000	5000	9.27sec	6.56sec	10.84sec

7. 获取用户购物车信息，见下表 3-8：

表 3-8 获取购物车信息测试结果
Table 3-8 Get shopping cart test results

每秒请求用户数	总请求用户数	成功响应用户数	成功响应平均时间	最快成功响应时间	最慢成功响应时间
1	10	10	0.36sec	0.22sec	0.47sec
10	100	100	0.62sec	0.37sec	1.12sec
100	1000	1000	11.53sec	3.44sec	34.24sec
500	5000	3053	42.65sec	7.56sec	1min14sec

8. 购物车结算，见下表 3-9：

表 3-9 购物车结算测试结果
Table 3-9 Shopping cart get total price test results

每秒请求用户数	总请求用户数	成功响应用户数	成功响应平均时间	最快成功响应时间	最慢成功响应时间
1	10	10	1.87sec	1.47sec	2.51sec
10	100	100	3.52sec	1.88sec	6.95sec
100	1000	1000	47.94sec	9.09sec	1min47sec
500	5000	1895	1min15sec	13.70sec	2min5sec

9. 结算完成后生成订单，见下表 3-10：

表 3-10 下订单测试结果
Table 3-10 place order test results

每秒请求用户数	总请求用户数	成功响应用户数	成功响应平均时间	最快成功响应时间	最慢成功响应时间
0.2	10	10	12.10sec	8.31sec	15.69sec
1	10	10	13.52sec	8.70sec	17.28sec
10	100	100	1min7sec	46.14sec	1min45sec

10. 进入个人页面（读取收藏数，订单数，浏览历史数，待评论数），见下表 3-11：

表 3-11 进入个人页面测试结果
Table 3-11 Get personal info test results

每秒请求用户数	总请求用户数	成功响应用户数	成功响应平均时间	最快成功响应时间	最慢成功响应时间
1	10	10	0.62sec	0.47sec	0.74sec
10	100	100	1.01sec	0.66sec	1.4sec
100	1000	1000	24.35sec	6.49sec	42.34sec
500	5000	2873	51.75sec	20.56sec	1min13sec

11. 获取个人线上订单，见下表 3-12：

表 3-12 获取个人线上订单测试结果
Table 3-12 Get online order test results

每秒请求用户数	总请求用户数	成功响应用户数	成功响应平均时间	最快成功响应时间	最慢成功响应时间
1	10	10	80.19msec	68.47msec	91.59msec
10	100	100	0.25sec	78.86msec	0.24sec
100	1000	1000	4.27sec	70.18msec	18.65sec
500	5000	4163	21.56sec	6.10sec	47.28sec

12. 获取个人线下订单，见下表 3-13：

表 3-13 获取个人线下订单测试结果
Table 3-13 Get offline order test results

每秒请求用户数	总请求用户数	成功响应用户数	成功响应平均时间	最快成功响应时间	最慢成功响应时间
1	10	10	0.23sec	0.11sec	0.37sec
10	100	100	0.57sec	0.44sec	0.73sec
100	1000	1000	9.48sec	4.49sec	22.56sec
500	5000	3921	42.92sec	18.33sec	1min15sec

13. 获取个人浏览历史，见下表 3-14：

表 3-14 获取个人浏览历史测试结果
Table 3-14 Get browse history

每秒请求用户数	总请求用户数	成功响应用户数	成功响应平均时间	最快成功响应时间	最慢成功响应时间
1	10	0	TIMEOUT	TIMEOUT	TIMEOUT
10	100	0	TIMEOUT	TIMEOUT	TIMEOUT

3.4 测试结果分析

根据测试结果，从响应成功率来说，可以发现当负载压力增大到每秒产生 500 个用户请求时，服务器已无法全部处理，最终可成功处理的请求数在 3000 到 5000

之间。通过对平均响应时间和响应成功率之间数据关系的观察，可以发现对于平均响应时间较低的接口，每秒可成功处理的请求数越多。而随着接口平均响应时间的上升，每秒可处理的请求数开始下降，同时查看测试日志可发现 ERROR_CONNECT_ECONNREFUESD 错误记录，说明服务器开始拒绝过多的连接请求。而通过对响应时间曲线的观察，如图 3-4 所示为获取首页接口在每秒 100 个并发请求负载下的响应时间曲线，可以发现响应时间呈上升趋势，可分析出由于大量未处理请求的积压，导致后续到来的请求只能排队等待，从而导致响应时间越来越长，而当积累的未处理请求达到一定的数量后，服务器开始拒绝更多的请求以保证服务器的可用性。由本文第二章介绍的 Node.js 的 I/O 模型和单进程模型可知，此时 Node.js 工作进程已经处于满负载的状态。由此可分析出，想要提高系统的并发性，一个途径是降低每个接口的响应时间，另一个途径是通过改进服务器架构，提高工作进程的数量来分担压力，提高并发性。

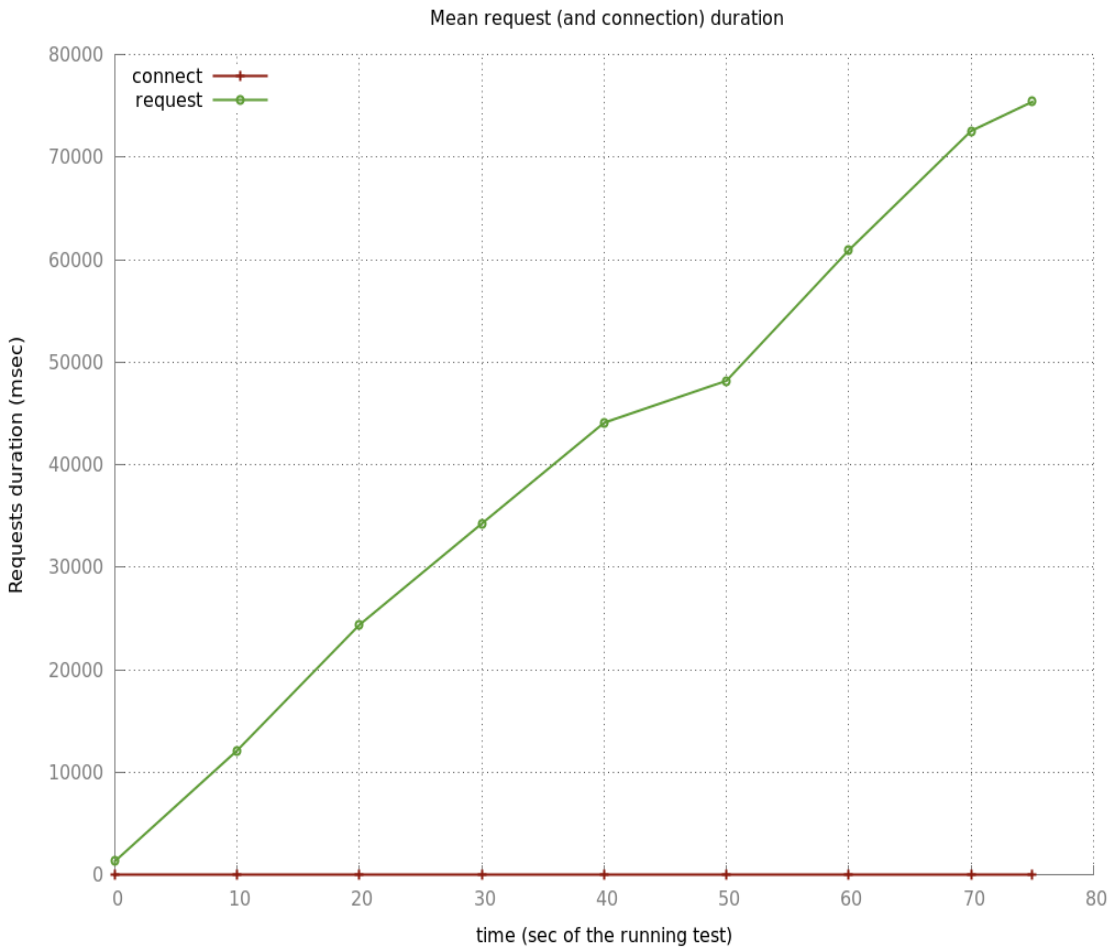

图 3-4 获取首页响应时间曲线图

Fig.3-4 Get home page response time curve chart

从响应时间来说，如图 3-5 所示为低负载情况下（每秒产生一个用户和每秒产生十个用户）的性能测试平均响应时间对比图，从图中可以看出其中有两个接口性能表现最好，分别是测试序号 2 获取首页滚动栏和测试序号 6 商品加入购物车接口。余下接口中测试序号 1 加载首页、测试序号 3 获取滚动栏详情、测试序号 4 获取商品分类列表、测试序号 7 获取用户购物车、测试序号 10 获取用户个人数据、测试序号 11 获取用户全部线上订单和测试序号 12 获取用户全部线下订单性能表现较为平均。在这些性能表现较好的接口中，可以发现每秒 1 个用户请求和每秒 10 个用户请求对响应时间基本不会有什么影响，服务器都可以快速处理。而剩下的接口中如测试序号 5 商品详情及测试序号 8 购物车结算性能较差，此时并发量的不同就会对系统性能产生比较明显的影响。而图中未列出的剩余两个接口，测试序号 9 生成订单以及测试序号 13 获取用户浏览历史则性能有明显的问题。测试序号 9 生成订单接口由于业务的复杂性导致性能和并发度太差，因此采用了不同于其它接口的测试压力，而测试序号 13 的接口获取用户浏览历史则由于数据量过大，数据库重新建立数据库视图失败而导致查询超时。

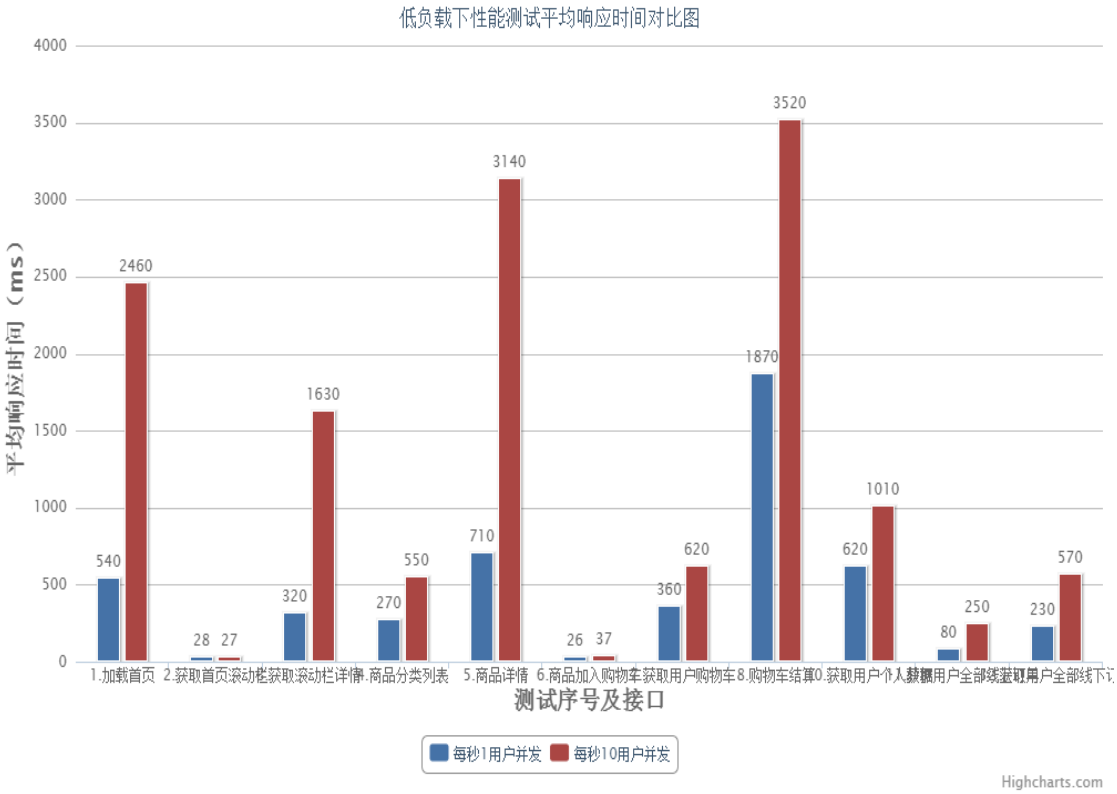

图 3-5 低负载下性能测试平均响应时间对比图

Fig.3-5 Benchmark average response time comparison chart under low load

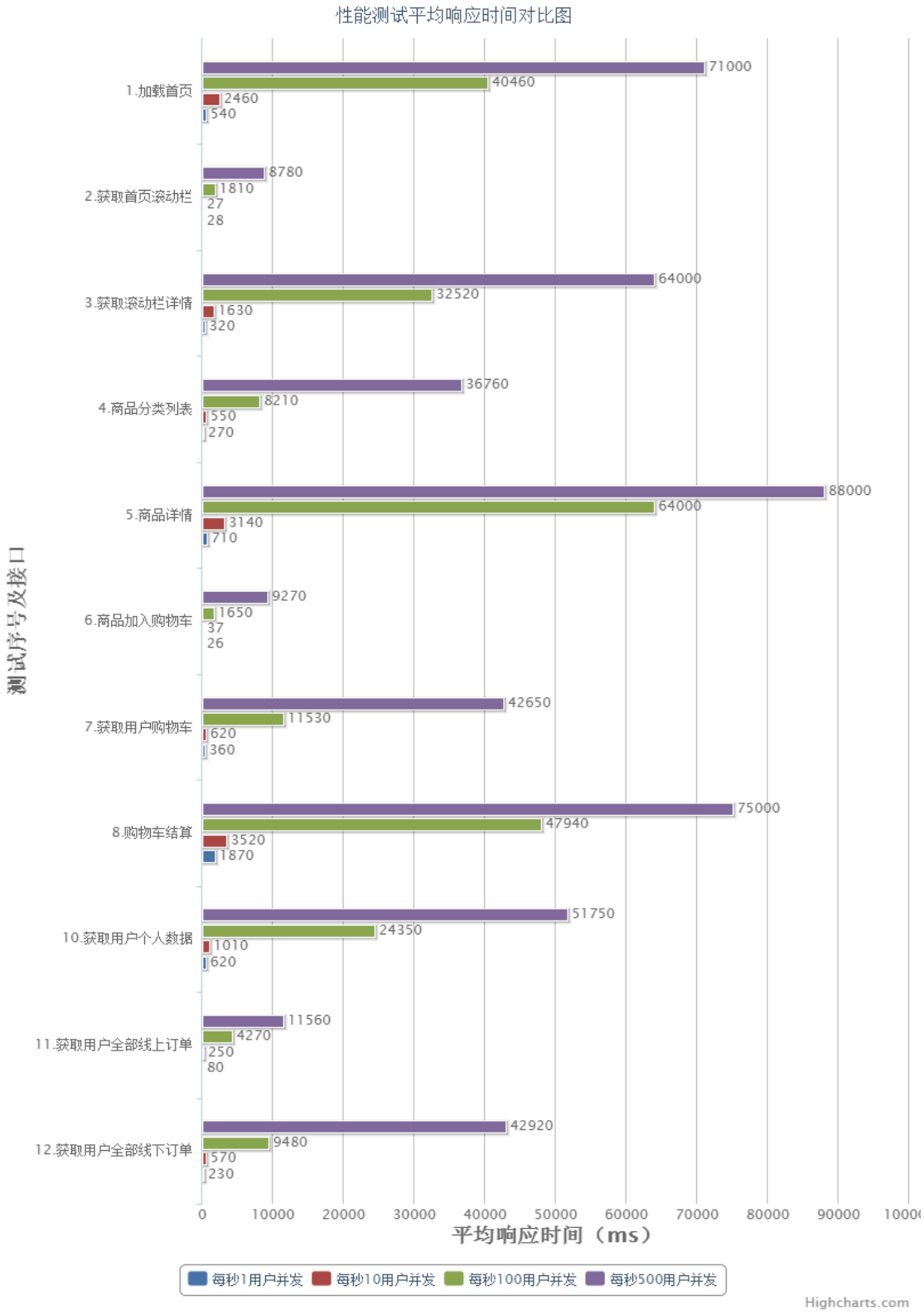

图 3-6 性能测试平均时间对比图

Fig.3-6 Benchmark average response time comparison chart

如图 3-6 所示为全部负载压力下性能测试平均响应时间对比图，通过对比可以看出当负载压力从每秒 10 个用户上升到每秒 100 个用户同时请求时，对系统的性能有较为严重的影响，而相比较而言，当从每秒 100 个用户上升到每秒 500 个用户时，性能的影响相对来说要小得多。下面根据接口表现出的性能水平将接口分为 A、B、C、D 四类，其中 A 类接口在测试中表现出的性能水平最好，而 B 类接口表现出了一般的性能水平，C 类接口的性能表现较差，值得关注和优化，D 类接口已经大大影响了系统的可用性和用户体验，需要尽可能大力优化，如下表 3-15 所示为根据接口性能进行的接口分类。

表 3-15 接口根据性能分类表

Table 3-15 Interface classification table based on performance

分类编号	接口
A	测试序号 2 获取首页滚动条、测试序号 6 商品加入购物车
B	测试序号 1 加载首页、测试序号 3 获取滚动栏详情、测试序号 4 获取商品分类列表、测试序号 7 获取用户购物车、测试序号 10 获取用户个人数据、测试序号 11 获取用户全部线上订单和测试序号 12 获取用户全部线下订单
C	测试序号 5 商品详情、测试序号 8 购物车结算
D	测试序号 9 生成订单、测试序号 13 获取用户浏览历史

接下来从系统实现的角度来分析各分类接口之间性能差距的原因：

1）A 类接口：

- 测试序号 1 获取首页滚动条接口简单的查询了一张数据库表即返回了结果。
- 测试序号 6 添加购物车接口也只是对单张数据库表的查询然后更新，由于查询然后更新操作会对数据库表的行锁定，因此当并发量增大时，测试序号 6 的接口会受到更大的影响。

2）B 类接口：

- 测试序号 1 加载首页接口会去数据库中首页活动商品视图中查询出首页活动商品，然后根据查询出的首页活动商品根据当前设置的首页模板组装成 html 文件返回。
- 测试序号 3 获取滚动条详情接口首先根据传入的滚动条活动编号去数据库表查询活动详情，如果存在该活动，则接下来去活动商品视图查询该活动关联的商品，然后进行返回。
- 测试序号 4 获取商品分类列表接口根据传入的分类以及排序和页码直接去商品分类视图中进行查询。

- 测试序号 7 获取用户购物车接口根据传入的 USERCODE 去用户购物车商品视图进行查询，返回对应结果。
- 测试序号 10 获取用户个人数据接口需要返回用户关于收藏商品的数量、收藏品牌的数量、浏览历史的数量、未付款订单的数量、已付款未完成订单的数量、未评论商品数，因此，需要根据 USERCODE 去依次到收藏商品表、收藏品牌表、浏览历史表、线上订单表、线下订单表和未评论商品视图查询各自数量，然后根据查询到的订单状态计算出未付款的订单和已付款的订单数量，返回对应结果。
- 测试序号 11 获取用户全部线上订单首先根据用户传入的 USERCODE 到线上订单表查询用户所有线上订单，然后根据订单号码到订单商品视图中依次查询订单中包含的商品，然后返回对应结果。
- 测试序号 12 获取用户全部线下订单会执行和测试序号 11 获取用户全部线上订单接口相似的操作，但是操作对象是线下订单表和视图。

3）C 类接口：

- 测试序号 5 商品详情接口执行流程如图 3-7 所示，该接口首先去线上商品视图中取出商品线上基本信息，然后取出该商品全部 SKC 码，根据 SKC 码查出各 SKC 库存，接下来根据商品信息判断是否有线上图片，如果有则取出线上图片，接下来取出该商品对应商场促销信息和供应商促销信息，最后，如果用户处于登录状态，则需要查出用户是否收藏了该商品以及该品牌，并记录下用户本次浏览历史，最后返回数据。可以看到这一次请求一共需要进行 9 次数据库操作。
- 测试序号 8 购物车结算执行流程如图 3-8 所示，从图中可以看出首先需要一次数据库操作根据用户的 USERCODE 查出 VIPCODE，接下来对每个选中商品，需要跟据选中商品的 SKCCODE 查出 ITEMCODE，再根据 ITEMCODE 查出柜台号，售价，促销标志，接下来查出供应商促销标志，如果选中 N 个商品，则需要 3N 次数据库操作，完成后，根据上面查出的 VIPCODE 到 MIS 查询每个商品的折扣数据，然后将上面全部查询出的信息整理好之后传入促销引擎计算价钱。总结下可以得出这个接口一次请求需要执行 1+3N 次数据库操作+2 次 Http 请求（1 次 MIS+1 次促销引擎）。

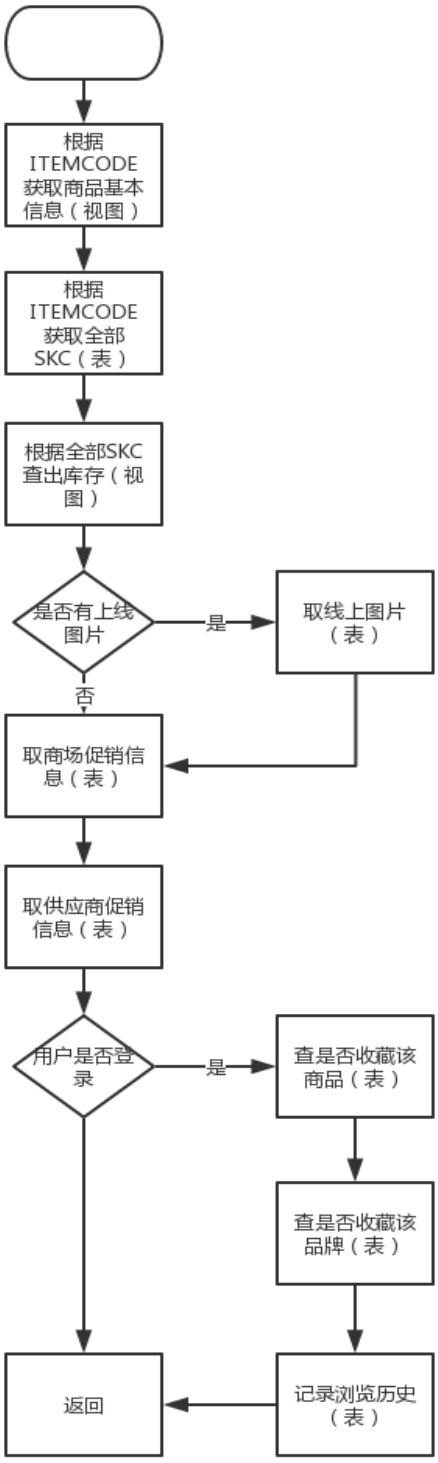

图 3-7 获取商品详情流程图
Fig.3-7 Get item detail flow chart

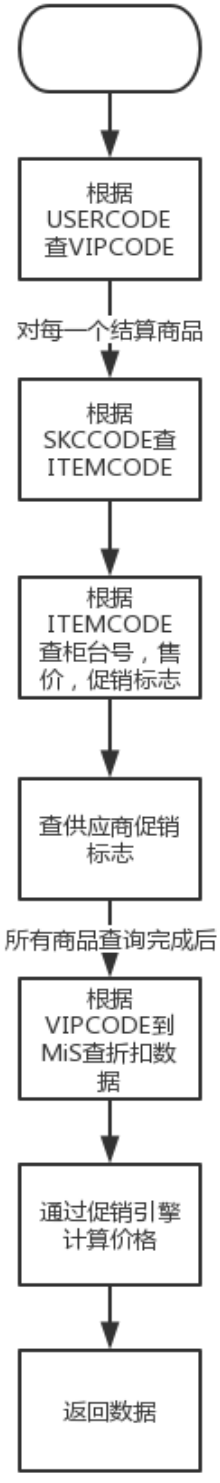

图 3-8 购物车结算流程图
Fig.3-8 Shopping cart get total price flow chart

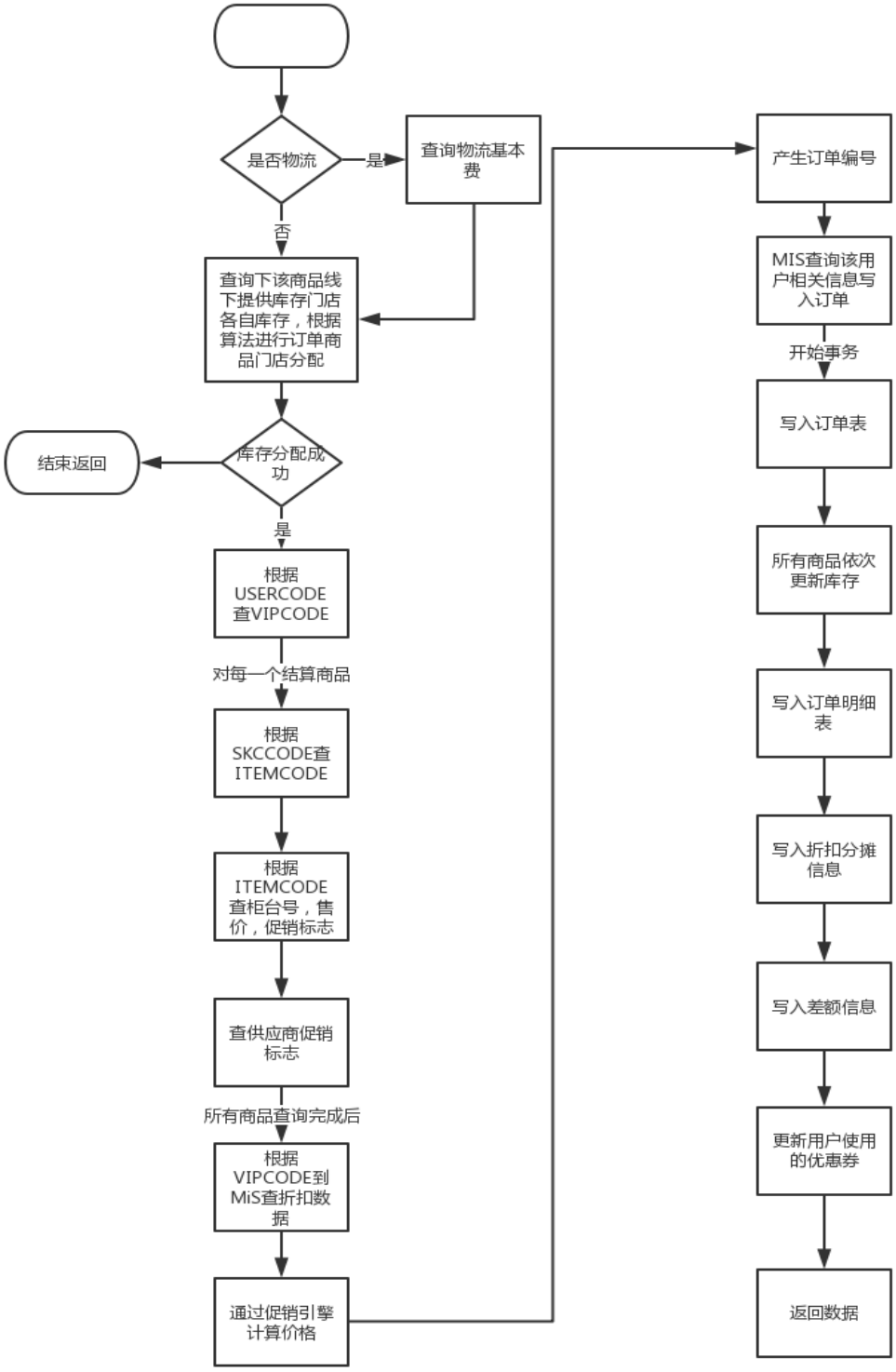

图 3-9 生成订单流程图
Fig.3-9 Place order flow chart

4）D 类接口：

- 测试序号 9 生成订单相比于上面所述接口处理流程要更为复杂，同时也是全部接口中最重要的接口，强调强一致性。如图 3-9 所示为生成订单流程图，从图中我们可以看出首先会查询物流基本费，接下来会查询一次库存来进行订单商品门店的分配，接下来类似于商品结算过程，查出 VIPCODE，依次查出各商品信息，然后通过促销引擎计算价钱，之后产生订单编号，接下来再次去 MIS 系统中查询需要写入到订单信息中的该用户相关信息。所有操作执行完成后，开启数据库事务，依次写入订单表，所有商品更新库存，写入订单明细表，写入折扣分摊信息表，写入差额信息表，更新用户使用的优惠券信息表，当事务完成以后，返回用户数据。总结下可以得出这个接口一次请求需要执行 5+6N 次数据库操作+3 次 Http 请求（2 次 MIS+1 次促销引擎）。
- 测试序号 13 获取用户浏览历史接口在本次测试中直接失去了可用性的保证，所有的请求都无法得到返回。然而本接口并不像上述复杂接口那样具有程序复杂性，程序实现中只是通过用户浏览历史商品视图直接根据用户 USERCODE 来进行查询。当跟踪到该视图时发现数据库对于该视图的查询始终超时不能返回结果。研究该视图发现该视图只是用户浏览历史记录表和线上商品基本信息视图的简单连接，但是当进一步深入研究时发现，线上商品基本信息视图由于需要提供线上商品展示所需要的全部信息，因此该视图结构极为复杂，通过主表商品信息表连接包括品牌表，商品类型表，商品预览图视图，商品分类视图，商品种类视图，而其中所涉及到的三张连接视图都使用了聚合函数并且线上商品基本信息视图也使用了聚合函数。因此当我们跟踪视图查询语句的执行过程可以发现，所有的视图都使用了 TEMPTABLE 算法，MySQL 执行引擎需要首先建立商品预览视图、商品分类视图、商品类型视图的临时表，然后这三张临时表和商品信息表，品牌表，商品类型表进行连接查询最终生成商品基本信息视图，此时商品基本信息视图已经具有 22 万条记录，接下来，拥有 3 万条记录的用户浏览历史表与商品基本信息视图进行连接查询，生成临时表，此时测试机的数据库引擎已经无能无力，一直处于运算过程中。而这仅仅是试运营时的情况，当系统正式推广，浏览历史视图一定会呈数量级规模增长，很快生产环境数据库也将会无能为力。

通过对测试结果的详细分析，最终我们发现了系统性能瓶颈的所在并找到了改进的方向。首先是需要对当前服务器架构进行升级，以满足更多的并发需求；其次，在系统储存层，基于传统线下架构设计的低冗余，重事务的关系型数据库在面对业务升级，系统复杂性增加的情况下已经无能为力，如何改进存储层的设计是性能优化过程中的关键；最后，在性能分析过程中发现存在某些接口存在一次请求需要进行十多次数据库操作，并且这些接口还是对系统至关重要的诸如显示商品详情，购物车结算以及生成订单操作，从程序层面优化接口实现也是性能优化的重要方向。

3.5 本章小结

本章首先介绍了徐家汇汇金百货的零售系统以及在此基础上进行的 O2O 业务升级。接下来，对进行了业务升级之后的 O2O 系统进行了全面的性能测试，通过对系统性能测试的测试结果进行研究与分析，识别出了业务升级后 O2O 系统所存在的性能瓶颈和优化方向。

下一章，我们将根据当前系统存在的性能瓶颈，分别从服务器架构、存储层架构、程序实现的角度分别对系统进行性能优化。

第四章 优化设计与实现

4.1 优化的服务器架构

从测试结果中可以看到，当前的服务器架构已经达到了所能处理请求的负载极限，那么如何在当前服务器架构的基础上扩展服务器集群就成为了服务器能否承受高并发请求的关键影响因素。

4.1.1 常见的服务器集群架构方案

目前分布式服务器集群主要有以下几种常用的集群架构：负载均衡模式、主从响应模式、P2P 模式。如图 4-1 所示为几种常见集群架构的示意图，下面将对每一种的集群架构模式做出简要说明，并选择出适合本文环境下的集群架构方案。

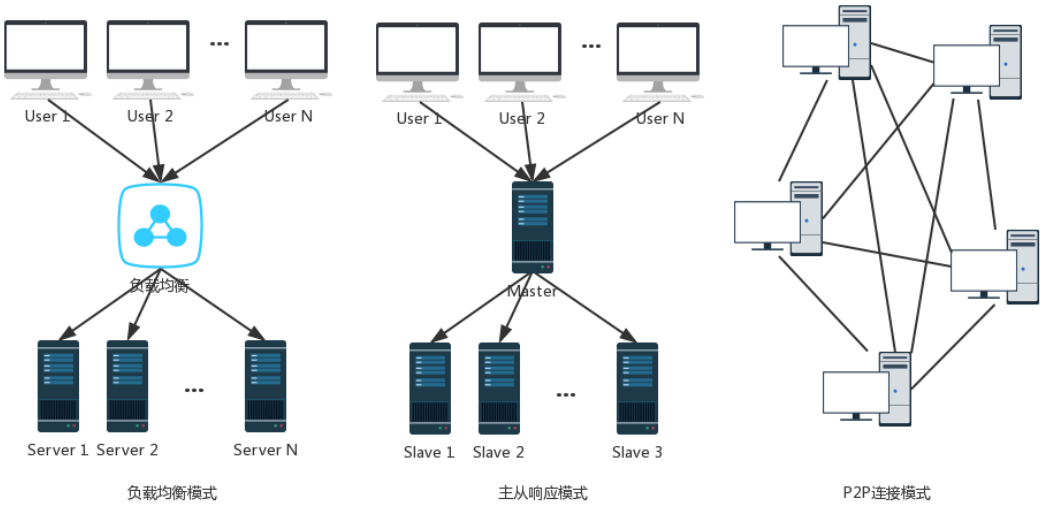

图 4-1 三种服务器集群方案
Fig.4-1 Three kinds of server cluster architecture

1）负载均衡模式：负载均衡是一种用来在多个计算机、网络连接、CPU、磁盘驱动器或其他资源中分配负载的计算机网络技术，来达到最佳化资源使用、最大化吞吐率、最小化响应时间同时避免过载的目的[26]。如上图 4-1 中左图所示即为负载均衡模式。在负载均衡模式下，在服务器和客户端之间加入了一层中间层——负载均衡层。所有的用户请求会首先发送到负载均衡层，负载均衡层收到用

户请求后，根据一定的规则和算法将请求分配到服务层进行处理。常用的负载均衡器有硬件负载均衡器，例如 A10、F5、Radware 等；也有软件负载均衡器，如 LVS、nginx 以及 HAProxy 等。硬件负载均衡器的特点在于性能好，售后支持服务好，但是相对应带来的成本就是花销大，而软件负载均衡的优势在于开源、免费。负载均衡模式是常用的服务器集群架构方案。

2）主从响应模式：主从响应模式如上图 4-1 中中图所示，服务器节点被分为 Master 节点和 Slave 节点，用户的所有请求直接访问 Master 节点，然后由 Master 节点以同步或异步的方式访问 Slave 节点[27]。负载均衡模式下各节点地位一致，一般不会互相通信，因此无法处理各节点间的一致性依赖关系，而主从响应模式下通过 Master 节点可以很好的管理各节点间的一致性关系。一般有较高数据一致性要求的系统会采用主从模式，例如关系型数据库集群通常采用主从模式来保证数据间的强一致性。

3）P2P 模式：P2P 模式如上图 4-1 中右图所示。在 P2P 模式中，所有的节点具有两个角色，即是服务器，也是客户端。节点之间地位对等，通过分布式协议 Paxos 来保证集群的一致性[28]。P2P 模式通常应用于社交网络或者节点间直连分享中。

通过上文的分析可知，负载均衡模式是最合适当前系统的服务器集群方案。同时，基于成本控制的考虑，软件负载均衡是最好的选择。

4.1.2 Node.js 的多核解决方案

在本文第二章中介绍了 Node.js 平台的单进程模型，Node.js 为了达到高性能高并发的目的放弃了多线程模型，虽然节省下来了线程创建和线程切换间的开销，然而，由于现代服务器早已迎来了多核，多 CPU 时代，因此 Node.js 的单进程单线程模型只使用了服务器中单个 CPU 核心是对系统资源极大的浪费。当运行在单个 CPU 核心上的 Node.js 主线程处于满负载状态时，其它 CPU 核心却处于闲置状态，同时由于 Node.js 不需要创建额外的线程，因此大量的内存也处于闲置状态。综上所述，如何能够高效率的利用现代服务器全部的性能是提高服务器高并发能力至关重要的第一步。

经过学习和研究，提出了以下几种 Node.js 在多核服务器下高效率利用系统资源的解决方案。接下来将各个方案分别介绍：

- 由负载均衡模式受到启发，单台服务器可以通过虚拟机的形式分别提供资源，由此在单台机器上搭建出服务器集群。如图 4-2 所示为在单台服务器

上通过软件负载均衡以及虚拟机搭建出服务器集群的示意图。在操作系统上留出一个 CPU 核心和部分内存分配给 Nginx 作为负载均衡程序，在剩下的资源中搭建出虚拟机环境，根据 CPU 核心数量部署对应个虚拟机，在虚拟机上运行 Node Server，并配置到负载均衡 Nginx[29]中。由此，我们在单台服务器上通过虚拟机搭建出负载均衡集群，充分利用了服务器的硬件资源。然而，此方案有一些明显的缺点，首先，环境搭建复杂，需要在原操作系统上安装虚拟机管理程序，然后安装对应数量个虚拟机，每个虚拟机需要配置运行时环境，然后依次部署 Node.js 平台并配置到 Nginx 负载均衡中，同时，每次系统更新升级，需要到虚拟机环境中对 Node Server 依次升级，运行维护成本很高。同时，由图 4-2 可以看出，相比于此前直接运行于宿主机操作系统，虚拟机增加了两层抽象，Node Server 运行于虚拟 CPU，虚拟内存上，通过虚拟机管理程序来映射到物理 CPU，物理内存，这之间会导致不必要的性能浪费。

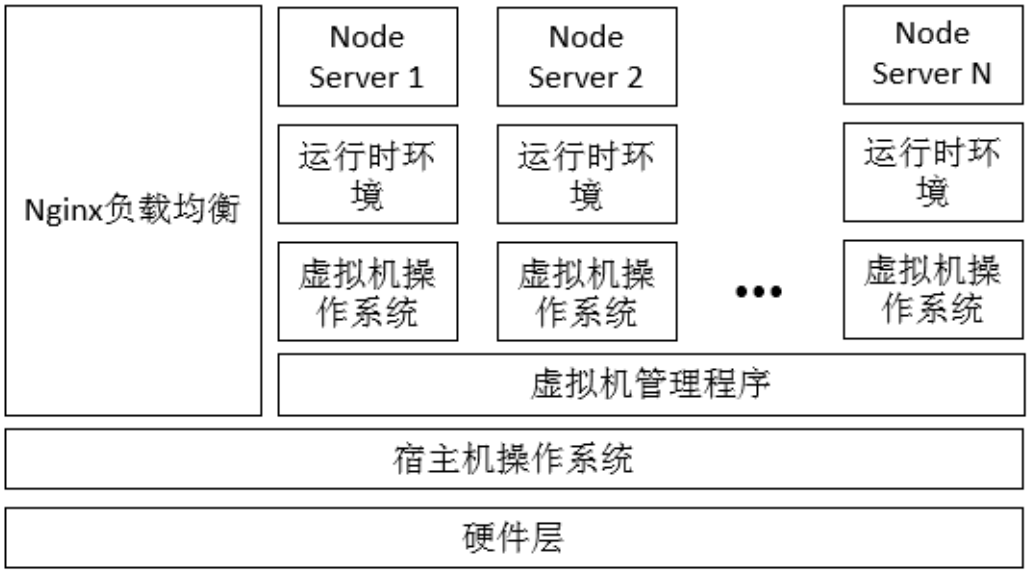

图 4-2 单台服务器多虚拟机负载均衡架构图
Fig.4-2 Multi virtual machine on a real machine load balance architecture

- 面对上述方案的缺点，Docker 容器化技术出现了[30]。Docker 通过 linux 容器的方式进行环境隔离，通过更少的抽象层，直接使用物理机硬件资源，而不是通过虚拟机管理程序实现硬件虚拟化，获得了更好的性能。同时，Docker 提供了一整套打包封装部署的流程，因此极大的方

便了程序的部署。相比于使用虚拟机技术，由于 Docker 使用了更低的隔离级别，因此隔离性和安全性要差于虚拟机技术。然而，由于是在内网环境中使用，而不是提供云服务。因此，隔离性和安全性并不需要过分关注。由此，可以考虑能不能更进一步减少隔离性，提供更高的性能。操作系统进程作为程序运行和分配资源的基本单位，可以分配到各自独立的内存、I/O 和 CPU 资源，由操作系统提供了基础的隔离性。因此，可以考虑通过多进程来启动多个 Node Server，搭建服务器集群。同时，为了避免操作系统对于进程调度所带来的额外开销，可以利用 Linux 操作系统提供的绑定指定进程到指定 CPU 的技术。最终，可以形成如图 4-3 所示集群架构，一个 CPU 核心运行 Nginx 负载均衡，余下的 CPU 核心分别绑定一个 Node Server 进程并监听不同的端口，最后通过把这些端口配置到 Nginx 实现负载均衡集群模式。

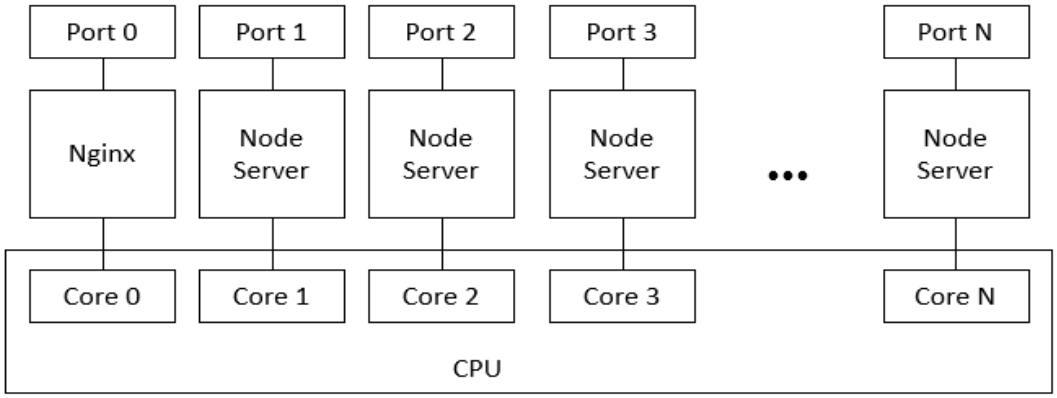

图 4-3 Node.js 多进程负载均衡架构
Fig.4-3 Node.js Multi-Process load balance architecture

- 上述方案可以很好的解决 Node.js 单进程所造成的硬件资源浪费的问题。Node.js 官方也发现了单进程模型下对于系统资源的浪费，因此，在 Nodejs 发展到 V0.6.0 版本时，内置了 cluster 特性，通过 child_process 基于进程间的通信来创建子进程，通过多进程来合理利用系统资源。如图 4-4 所示，Cluster 通过 Master 进程来创建出多个 worker 子进程，并让 worker 进程监听同一个端口，当新连接到来，操作系统会唤醒其中一个 worker 进程执行处理逻辑。此时，系统的负载均衡由操作系统负责，理论上，操作系统会针对运行中的进程收集大量的指标，因此操作系统最有资格决定唤醒哪个进程进行处理。然而在实际观察中，

我们发现大多数连接最终都由其中两个或三个进程处理了。从操作系统的角度来看这样可以最大限度的避免上下文间的切换，然而这表现出了非常不均衡的负载。为了解决这一问题，Node.js 决定由自己负责进行负载的分配，最终形成了的多进程模型如图 4-5 所示，Master 进程在创建出多个 worker 进程后，仍然自己监听端口，接收请求，到请求到来之后，自己根据负载均衡算法分配到指定 worker 进程。目前的实现中负载均衡算法采用了 Round Robin 轮询调度算法，对于到来的请求，按照 worker 进程的顺序依次分配个各 worker 进程，并不断循环。这个算法的优点是其简洁性，无需管理状态，只需要依次分配连接。

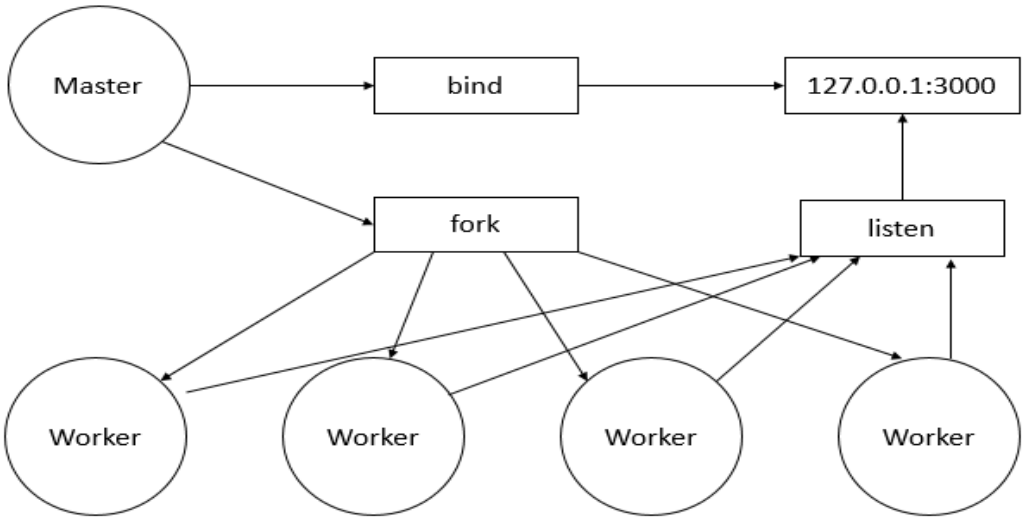

图 4-4 Node.js cluster 模块示意图
Fig.4-4 Node.js cluster module

通过上述方案对比可以发现，Nginx + 多进程负载均衡方案的优势在于由 Nginx 专门负责负载均衡，可以配置多种负载均衡算法，并且多进程间的隔离性更好，相互影响更小；劣势在于维护成本增加，需要增加一个 Nginx 负载均衡的配置和维护，并且需要对多进程进行操作系统 CPU 核心绑定以及端口的绑定。相对来说，采用 Node.js 内置多进程方案没有增加额外的维护成本，劣势在于负载均衡算法只能选择轮询调度。然而，对于无状态，同质性的服务器集群来说，轮询调度算法是一种低成本，高效率的调度算法。因此，综合各方面考虑，最终选择 Node.js 内置多进程方案来充分利用硬件资源，提高系统并发性。

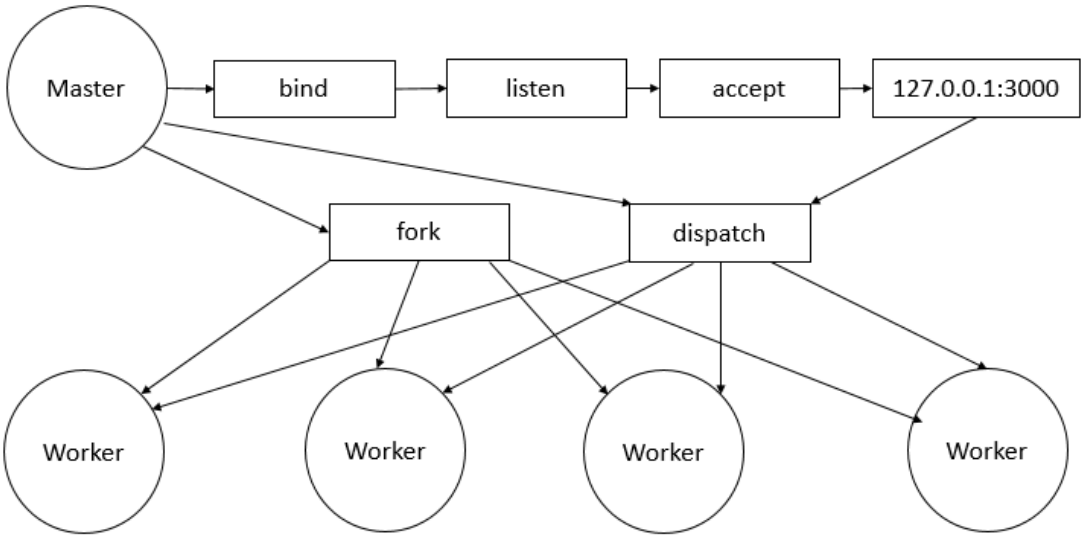

图 4-5 升级 Node.js cluster 模块示意图
Fig.4-5 Upgraded Node.js cluster module

4.1.3 服务器集群高可用实现

生产环境中服务器集群的高可用性是集群架构设计与实现过程中必须关注的部分，通常情况下可以在操作系统层面对服务进程进行守护和管理，例如 Supervisord。Supervisord 是利用 Python 开发的一套通用的进程管理服务[31]，能够将一个普通的命令行程序变为后台守护进程，并监控进程状态，异常退出时自动重启。然而，Supervisord 只能保证主进程的状态，却无法监控到主进程所创建的子进程状态。因此，为了保证服务器集群的高可用性，必须在主进程的内部实现子进程的状态监控。在设计上通过在 cluster 模块中 dispatch 功能模块上添加子进程状态监控，所有的 worker 进程定时向 dispatch 反馈心跳，收集自身的信息报告给 dispatch，dispatch 通过 worker 的心跳信息来监控 worker 进程的状态。在实现过程中，研究了 Node.js 开源进程管理模块 PM2。PM2 在 cluster 模块的基础上集成了进程管理与监控，不停机维护升级以及负载均衡配置。最终，由于 PM2 已经可以满足服务器集群的需求，因此选择了使用在项目中集成 PM2 而非自己实现。

4.1.4 性能对比测试

为了验证实验效果，将 PM2 集成到项目中后进行对比测试。为了测试数据的准确性，保持其它变量不变。即在第三章测试环境的基础上集成 PM2 模块，其它环境保持不变。为了减少数据库操作对性能测试的影响，选择了数据库操作最简

单性能最好的 A 类接口中获取首页滚动活动栏接口进行对比测试。测试方案不变，分别设置 PM2 启动 1 个进程，即不启用负载均衡模式，启动 2 个进程，即 1 个 master 进程 1 个 worker 进程和启动 4 个进程，即 1 个 master 进程和 3 个 worker 进程。测试结果如下表 4-1 所示：

表 4-1 服务器集群优化对比测试结果
Table 4-1 Server cluster optimization comparison test results

启动进程数	每秒请求用户数	总请求用户数	成功响应用户数	成功响应平均时间	最快成功响应时间	最慢成功响应时间
原	1	10	10	28.17msec	25.16msec	30.63msec
1	1	10	10	27.25msec	25.89msec	29.11msec
2	1	10	10	30.17msec	27.35msec	31.54msec
4	1	10	10	28.36msec	24.72msec	29.98msec
原	10	100	100	27.63msec	24.63msec	32.57msec
1	10	100	100	27.89msec	25.22msec	32.33msec
2	10	100	100	28.55msec	26.15msec	31.86msec
4	10	100	100	26.47msec	25.91msec	28.26msec
原	100	1000	1000	1.81sec	1.17sec	3.34sec
1	100	1000	1000	1.86sec	1.15sec	3.16sec
2	100	1000	1000	2.02sec	1.33sec	3.81sec
4	100	1000	1000	82.67msec	60.39msec	1.23sec
原	500	5000	5000	8.78sec	5.93sec	13.28sec
1	500	5000	5000	8.94sec	5.76sec	14.54sec
2	500	5000	5000	8.62sec	5.73sec	13.71sec
4	500	5000	5000	2.49sec	1.23sec	4.87sec

通过上表 4-1 所示的测试结果我们可以看到，在集成了 PM2 模块后，如果只启动 1 个进程，则仍是传统的单进程模式。如果启动 2 个进程，此时为一个 master 进程负责分发任务，一个 worker 进程负责处理，由于 master 进程只负责分发任务而仍然只有一个 worker 进程在具体处理，因此性能上与启动单个进程相似。而当我们启动了 4 个进程，此时一个 master 进程将任务平均分配到 3 个 worker 进程，通过测试结果可以看出在响应时间上得到了很大的优化。由此，建议在生产环境中启动进程的数量应等于当前系统闲置的 CPU 核心数量以达到最好的并发性。

4.2 优化的存储层架构

在优化了服务器集群后，系统在并发能力上得到了很大的提升。然而，在 O2O 交易系统中，数据库 I/O 占有着极大的比重。数据存储层的性能是影响系统整体性能的关键因素，优化存储层的架构有着重要的意义。

4.2.1 存储层的性能瓶颈与解决方案

在本文 3.4 节的测试结果分析中可以发现当前系统的存储层对整体性能造成了较大的影响。尤其是对于部分功能接口，造成了数据库操作次数过多，性能很差的问题。当对问题进行总结聚焦时，可以发现性能问题出现的最大原因在于线上商品信息的视图，当所有涉及到该视图的查询时，性能都不可避免的受到影响。另一个性能问题则出现在对于线上用户相关信息的管理。而当两个最大的问题碰到一起，就造成了用户浏览历史服务不可用的严重后果。因此，优化数据库性能问题的关键则在于优化线上商品信息以及线上用户信息。

对于线上商品信息，由于 O2O 交易系统的复杂性，尤其是线上线下融合系统。所有线上商品信息都是基于原始线下商品的信息补充得到，而原始线下信息表又是基于传统的关系型数据库设计原则设计。当需要查询出线上商品相关信息，势必要进行多个表的连接查询。从程序的设计角度以及可视化角度，大量的引入了数据库视图，但是数据库视图又造成查询无法利用到索引以及生成临时表并且全表扫描的问题。要解决此问题，首先想到的解决方案是想办法利用上数据库索引，比如对视图进行拆分，尽量可以使用 MERGE 算法执行的视图，或者在某些地方不使用视图，直接使用优化过的 SQL 语句执行[32]，然而，这一方案会造成大量的程序修改，并且对程序的逻辑性和可读性有极大的损失，同时很难保证优化效果。另一个方案可以考虑基于线上高并发的角度整体重新设计商品信息存储，然而，由于线下商品是线上商品的基础。这种方案会对当前稳定运行的线下系统有极大的影响并且有很大的工作量，成本极高。最终，注意到商品信息具有高频率读，低频率写的特性，并且修改频率很低，因此，决定采用在 MySQL 存储层之上添加一层缓存层的方案来提供高性能的商品读请求。

对于线上用户信息，特点是数据量大并且改动极为频繁，以用户浏览历史为例，当用户处于活跃状态时，每一次浏览记录都需要写入一次数据库，这是一个极高频的操作。同样的，用户的收藏，购物车等，都是高频易修改数据。并且随着用户数量的增长，表数据会迅速增长。常规的思考是对于大数据量的数据采用分库分表[33]，然而，用户数量的增长以及用户行为是不具有明显规律性的，很难进行粒度良好的分表。同时，分表并不能解决用户数据频繁读写的问题。因此，分表带来的成本并不能和收益成正比。考虑到用户数据没有事务性以及一致性的要求，可以考虑采用频繁读写性能更好的 NoSQL 来重新建立用户信息系统。

综上所述，系统存储层的瓶颈在于基于传统线下业务设计的线上商品信息读取以及用户行为数据的频繁大量写入读取。选择的解决方案为加入一层缓存层解

决线上商品信息读取问题并且加入 NoSQL 数据库应对用户行为数据的频繁读写。

对于缓存方案的处理，最常用的解决方案有 Memcached 和 Redis，两者都是基于键值对（Key-Value）的基于内存的数据存储。Memcached 是单纯的内存缓存[34]，而 Redis 具有内存型数据库的特性[35]。Redis 相比于 Memcached 多提供了数据持久化存储的功能，当服务器出现断电或遭遇故障，已经写入磁盘的持久化数据不会丢失，开机即可从磁盘中恢复。而 Memcached 由于数据全部保存在内存中，故障关机后则会损失所有数据。同时 Redis 相比 Memcached 提供了更多的数据类型，例如 Hash，List 和 Set 等，并提供了一些原子性操作的命令。因此在本文的使用场景中，Redis 是更适合的选择。

对于线上用户行为数据的 NoSQL 存储，最自然的解决方案是 MongoDB。MongoDB 是一个介于关系型和非关系型之间的数据库，是一个面向集合（Collection）、模式自由（Schema-Less）、文档型（Document）的数据库[36]。面向集合指的是在 MongoDB 中数据被划分在不同的数据集合中存储，类似于关系型数据库的表的概念。模式自由指的是同一个集合内的文档不要求一定要相同的格式，对集合不需要定义任何模式。在操作数据时不需担心数据库的数据格式，只需按需存储数据即可。文档型指的是集合中的数据全部是以文档的形式存储，一个集合中可以包含无数多个文档。文档是一组键值对，其中键是字符串而值可以是任意类型，这种类型类似于 Json 被 MongoDB 定义为 Bson。Bson 数据结构支持内嵌对象和数组。MongoDB 中的文档对象可以对应为关系型数据库中的一条记录。MongoDB 是一个基于磁盘的数据库，它的优势是在海量数据储存的情况下仍具有较高的访问效率，并且支持动态查询，可以根据值中的数据进行查询。同时 Bson 格式对于 Json 格式数据友好，十分适合直接存储 Json 类型数据。

然而，如果采用上述解决方案，需要首先在缓存层引入 Redis，然后在存储层引入 MongoDB，最终形成异构数据库。这对于系统的开发，部署以及维护都会带来很高的成本。此时，需要考虑是否有更好的解决方案或者折衷的方案。考虑到用户数据的数据结构，数据规模以及查询模式，可以发现使用 MongoDB 是否有些大材小用。对于用户行为数据，主要的存储数据有用户浏览历史、用户搜索历史、用户商品收藏、用户品牌收藏、用户购物车等。这些数据都是限量数据，比如浏览历史最多显示 50 个，搜索历史最多显示 20 个等。同时，这些数据的格式决定了查询方式只能通过主键来进行查询而用不上通过值进行的高级查询功能。最后，这些数据的一致性和持久性并没有强烈的需求，我们可以容忍一分钟的浏览历史或者品牌收藏等数据的丢失。因此，可以考虑能否利用 Redis 的持久化特

性来将这部分数据也储存在 Redis 中，从而减少系统架构的复杂性。目前 Redis 持久化提供了两种方式：RDB 和 AOF。RDB 就是快照储存，是默认的持久化方式。Redis 会根据设定的的策略周期性的将数据保存到磁盘，由于是周期性写入，因此在上次写入到 Redis 停机之间这段时间的数据就会丢失。AOF（Append-Only File）相比于 RDB 方式有很好的持久性，每次 Redis 收到一个写命令都会通过 write 函数追加到持久化文件中，当 Redis 重启，这些数据会通过重新执行一遍文件中保存的写命令来还原数据库内容。然而，AOF 持久化方式会带来持久化文件越来越大的问题，例如执行了 INCR test 命令 100 次会在文件中保存这 100 条命令，但实际上效果等效于 SET test 100，其中 99 条数据都是多余的。在两个方式的选择上，如果想要很高的数据保障性，那么应该同时启用两种持久化方式。而如果可以接受几分钟的数据丢失，那么 RBD 持久化模式就可以满足需求了。通过上面的分析可以看到，Redis 所能提供的持久化可以满足本文系统环境下对于数据持久性的要求，因此，最终决定选择 Redis 来作为商品信息缓存以及用户行为数据储存的解决方案。

4.2.2 商品信息缓存

如图 4-6 所示的 JSON 数据结构是目前线上商品信息视图所包含的字段。通过对字段的分析，可以把这些字段分为四类：

- 商品固有基本属性：包括 ITEMCODE，BARCODE，ITEMNAME 等，此类属性完全属于商品自身，并且具有改动频率小的特点。。
- 商品变动属性：包括 SCORE，SCOREPOPULARITY，PURCHASE，PURCHASEPOPULARITY，PRAISERATE 等。此类属性属于商品自身，但是具有易变动性。
- 商品连接属性：包括 BRANDNAME，ITEMTYPENAME，CATEGORYNAMES，REGIONNAMES 等，此类属性属于商品自身，但是值需要通过数据库连接去另外表查询。
- 商品附加属性：包括 ISSHOWONLINE,IMGURL 等，这两个属性用来标记该商品是否在 APP 主动展示以及预览图的 URL。修改几率略高于其它固有属性。

通过对上述四类属性的分析，在商品信息缓存的设计上就有两种方案可选，一是将四种类型属性分别缓存，通过 ITEMCODE 作为主键，在查询时进行四次查询得出结果。此方案的好处是更为灵活，四类属性互相不影响。另一种方案是考

虑到属性的不易变性以及 Redis 提供的 Hash 数据结构，将全部四种属性统一存在 ITEMCODE 键下的 hash 数据结构里，当需要更新时，借助 Redis 提供的对 Hash 数据结构根据主键和域单独更新的命令执行更新。此种方式的劣势在于对于商品连接属性存在在缓存中存在冗余以及当商品连接属性的连接表里属性变动时，缓存中的值难以更新。对于这两个问题，可以发现冗余属性所占用的内存极小，在当今服务器硬件条件下可以忽略不计。对于连接属性，通过业务分析，品牌名，商品种类名，分类名等此类属性一经确定，改动可能性很小。因此可以接受在大部分请求中减少一个查询而在极少出现的改动情况下进行一次大量更新。

图 4-6 线上商品信息字段
Fig.4-6 Online item detail fields

下面讨论商品相关信息数据缓存。从本文第三章测试结果分析中对获取商品详情的接口解析可以看到，在执行获取商品详情时，首先会获取商品基本信息，

接下来，会获取商品相关的 SKC 信息以及 SKC 对应的库存信息。SKC 信息对于商品来说属于不易变动属性，满足读多写少属性，因此可以放入缓存中加速性能。从缓存的设计角度，可以把商品的 SKC 信息作为 Hash 数据结构存放在商品 ITEMCODE 主键下。对于库存信息，属于读多写多的属性，并不适合放在缓存中。然而，从宏观的角度来看，库存的数量属于易变值，而库存的状态有无则变化频率要低的多，只有在商品库存全部卖完时会出现变化，因此，商品库存信息可以以有无库存的形式进行缓存。

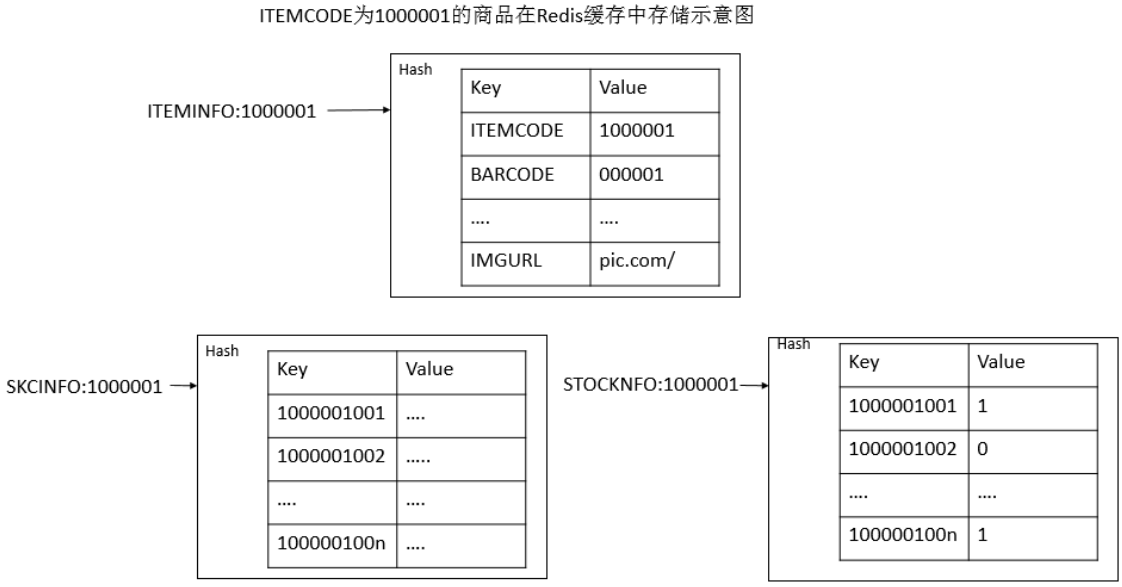

图 4-7 示例商品 Redis 缓存结构图
Fig.4-7 The sample good Redis cache architecture

综上所述，如图 4-7 所示为示例商品 100001 在 Redis 缓存中的存储结构设计，分别使用了三个 Hash 数据结构来分别存放商品基本信息，SKC 信息以及库存信息，命名空间分别为：“ITEMINFO” + ITEMCODE、“SKCINFO” + ITEMCODE、“STOCKINFO” + ITEMCODE。在 Hash 结构内部，分别以信息字段和 SKCCODE 作为 Hash 域。当执行查询操作，通过命名空间连接 ITEMCODE 即可一次查询到所需的全部信息，当需要单独查询，则可以在主键的基础上加上查询域即可查询到对应域数据。

对于线上商品的图片，由于不是所有的商品都在线上主动显示，因此不适合放在商品基本信息缓存里，所以单独建立线上商品图片 URL 缓存。

4.2.3 用户行为存储

由第三章中系统架构图可以看到，用户行为主要包含购物车，浏览历史，收藏和评价。其中用户评价相对于其他行为来说是低频操作，并且具有强持久化的要求，并不适合移动到 Redis 中进行储存，因此评论相关行为仍然保留在关系型数据库 MySQL 中。下面依次讨论浏览历史，用户收藏和购物车的存储设计：

- 浏览历史：浏览历史数据从结构上看可以理解为一个队列，每当用户浏览一个商品时会将这个商品添加到浏览历史队列的尾部，当用户查看浏览历史，将整个队列倒序展现给用户即可。同时，无论从存储成本角度以及用户体验的角度，用户浏览历史都不能无限制的记录下去。从目前的业务角度，可以设置用户浏览历史记录量为 50。同时，虽然过多的浏览历史对于用户来说并无价值，但是对于销售商来说，这些数据却是有价值的，通过用户的浏览历史可以进行数据挖掘和分析，了解出用户的偏好以制定更好的销售策略。因此，用户多余的浏览历史仍然需要做持久化保存以供后续使用。从设计角度来说，最简单的实现是基于 Redis 的 List 存储结构，每当用户有一个浏览记录，则进行加入队列操作，并且持久化到关系型数据库中。然而，这种实现没有利用上 Redis 读写速度快的优势，仍然需要频繁的写入数据库。由此，考虑利用 Redis 缓存来减少关系型数据库的读写次数。新的方案利用 Redis 来暂时储存多出的浏览记录，当达到一定的数量后，一次性持久化到数据库中。如图 4-8 所示，将 Redis 记录用户浏览历史的队列长度监控为 100，当用户浏览历史记录到 100 后，一次性将前 50 个过期的浏览历史移出队列并写入关系型数据库。

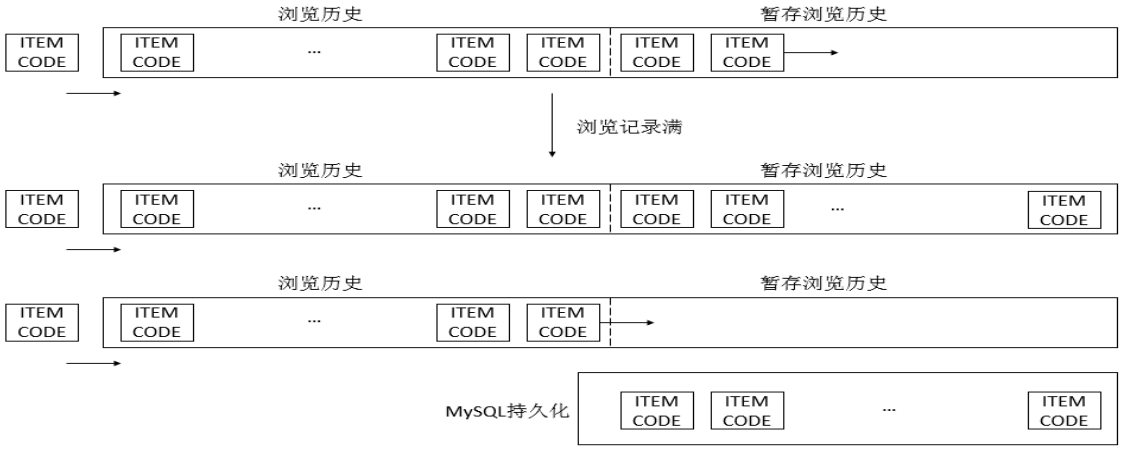

图 4-8 用户浏览历史 Redis 缓存结构图
Fig.4-8 User browse history Redis cache architecture

- 用户收藏：用户收藏从数据结构上来说可以当作集合。无序不重复的商品，因此恰好使用 Redis 提供的 Set 数据结构。对每个用户分别使用两个 Set 来存储商品收藏和品牌收藏。
- 购物车：购物车的实现上可以采用 Hash 数据结构，主键为用户 USERCODE，Hash 结构中域为商品 SKCCODE，值为购买数量。

4.2.4 系统缓存优化

下面根据本文第三章中对测试接口的分析，对系统其余部分的缓存进行优化设计并实现：

- 加载首页：首页是 APP 用户的入口，读取频率很高，原始的实现中直接通过首页活动商品视图查询出首页活动商品信息，在商品信息进行缓存优化后，此处可以将首页活动商品 ITEMCODE 作为 List 缓存在 Redis，每次加载首页只需要从 Redis 中取出活动商品列表后依次取出活动商品，节省了数据库 I/O 操作。
- 获取 APP 首页滚动活动栏：虽然获取首页滚动栏只经过了一次读数据库表操作，但是仍然可以通过将活动栏数据放入 Redis 缓存来减少数据库 I/O 的负载，得到更好的性能。
- 获取活动详情：获取活动详情类似于加载首页的实现，通过获取缓存中的活动商品列表然后取出活动商品。
- 商品分类列表：商品分类列表又是一个明显的读多写少的接口，因此可以通过 Redis 缓存来提高系统性能。然后分类列表需求较为复杂，需要满足多种排序以及分页需求，因此对于缓存方案的设计是一个考验。考虑到排序需求并且有分类中商品可能有增加或者删除的可能，Redis 提供的有序集合数据结构可以帮助我们完成这一需求。根据不同的排序方案缓存不同分类列表到各自的有序集合，集合中的权值就是排序的分值。当动态添加商品时，只需在分类排序集合中添加并设置对应的分值即自动完成了排序操作。考虑到缓存的高利用率，对每种分类只需缓存排序前十页商品，十页之后的数据仍然通过数据库读取。
- 获取用户个人数据接口：引入 Redis 缓存层之后，可以将用户个人数据在 Redis 中通过 Hash 数据结构存储。每次用户执行了相关操作之后对此数据进行更新，可以免去多次数据库遍历 COUNT 的开销。

4.2.5 性能对比测试

为了验证引入缓存层后的服务器性能是否得到了提升，在未升级的服务器集群的基础上，通过引入 Redis 缓存层，进行对比测试。为了保证在相同的实验条件下，将 Redis 数据库部署在部署有 MySQL 数据库的数据服务器上而没有引入额外的硬件或者云服务。测试接口选择了在初始测试时性能较差的测试接口 5 显示商品详情以及未响应的测试接口 13 获取用户浏览历史。为了让加入 Redis 测试环境尽量符合生产环境的状态，实现了一个程序遍历当前测试环境 MySQL 数据库中商品视图的数据并存放到 Redis 缓存中。然后遍历了浏览历史数据存储到 Redis 中。如下表 4-2，4-3 所示分别为测试接口 5 获取商品详情和测试接口 13 获取用户浏览历史的对比测试结果。

表 4-2 商品详情接口对比测试结果表
Table 4-2 Item detail comparison test results

测试环境	每秒请求用户数	总请求用户数	成功响应用户数	成功响应平均时间	最快成功响应时间	最慢成功响应时间
原始测试	1	10	10	0.71sec	0.63sec	0.78sec
对比测试	1	10	10	55.12msec	48.27msec	63.65msec
原始测试	10	100	100	3.14sec	2.03sec	4.34sec
对比测试	10	100	100	57.87msec	50.04msec	65.41msec
原始测试	100	1000	1000	1min4sec	12.81sec	2min1sec
对比测试	100	1000	1000	0.41sec	0.33sec	0.55sec
原始测试	500	5000	2237	1min28sec	28.57sec	2min15sec
对比测试	500	5000	5000	5.31sec	1.69sec	8.47sec

表 4-3 用户浏览历史接口对比测试结果表
Table 4-3 User browse history comparison test results

测试环境	每秒请求用户数	总请求用户数	成功响应用户数	成功响应平均时间	最快成功响应时间	最慢成功响应时间
原始测试	1	10	0	TIMEOUT	TIMEOUT	TIMEOUT
对比测试	1	10	10	32.12msec	27.48msec	39.46msec
原始测试	10	100	0	TIMEOUT	TIMEOUT	TIMEOUT
对比测试	10	100	100	33.87msec	25.76msec	41.29msec
原始测试	100	1000	0	TIMEOUT	TIMEOUT	TIMEOUT
对比测试	100	1000	1000	38.38msec	29.51msec	44.48msec
原始测试	500	5000	0	TIMEOUT	TIMEOUT	TIMEOUT
对比测试	500	5000	5000	2.74sec	0.65sec	3.28sec

通过上表的对比测试结果可以看到，在引入Redis缓存层减少数据库I/O之后，系统的性能得到了极大的提升。因此，对于读多写少适合缓存的数据，应尽量通过 Redis 缓存访问，将数据库 I/O 留给需要强一致性的事务操作。

4.3 优化的订单库存系统

订单库存系统是 O2O 交易系统的核心业务系统，同时也是要求强一致性事务性的业务系统。由本文第三章进行的性能测试可以看到，生成订单接口在性能，并发性上与其它接口有严重的性能差距。因此，在进行了系统服务器架构升级以及存储层架构升级后，仍需要对订单系统进行优化升级，提升订单系统的性能及并发性。

4.3.1 订单事务优化

在本文第三章对系统性能测试的结果分析中，根据图 3-8 所示的生成订单流程图所示,可以发现在一次生成订单的流程中进行了 5+6N 次数据库操作+3 次 Http 请求（2 次 MIS+1 次促销引擎），并且我们注意到订单执行流程第二步执行了库存查询操作，根据库存分配了订单到门店。然而，此时并没有加锁，接下来又经过了复杂的订单生成前准备工作，先根据了 USERCODE 查询了 VIPCODE，然后查询出商品相关信息经过促销引擎计算出了商品折扣，查询了两次 MIS 系统获取生成订单所需的数据，接着开启事务，写入订单表，接下来开始依次执行库存更新操作，设想如果此时是热门商品抢购的高并发场景，那么必然有大量订单请求来抢购少量库存。在一开始，全部请求都可以通过库存检查，然后在大量订单都经过了生成订单前的准备工作后，开始阻塞在加锁的扣库存请求，结果只有少量的订单可以完成扣库存操作，大量订单会回滚写入的订单表，同时浪费了上述从库存校验分配之后所有的操作。为了避免上述的情况下所造成的回滚和 I/O 操作浪费。可以考虑在第一次检查库存时加锁，这样通过了库存检查和分配的事务一定可以继续执行到事务结束。通过此途径节省了不必要的操作浪费，但是大量操作请求阻塞在第一步库存检查处。对高并发支持较差。尤其是当库存量大于并发请求量时，仍然需要依次排队等待执行。由上述方案可知，由于程序逻辑的复杂性，数据库提供的事务能力已经无法满足系统对于高并发的需求。想要能够获得更高程度的并发，必须自己处理事务逻辑而不能再依赖于数据库的事务管理。数据库提供的事务回滚本质上就是记录了操作的执行过程，当出错后，将操作回滚。那么，通过将库存与订单业务解耦，自己负责事务的管理，则可以不依赖数据库而获得更高的并发性能。经过仔细观察生成订单流程图，可以发现在经过库存分配检查后，程序执行流程并没有其它分支，也就是说，在正常情况下，程序经过了库存检查分配之后，会一直顺序执行到流程结束。那么，通过将库存事务与订

单解耦，将库存事务前提到库存检查分配时，对库存检查分配加锁，执行通过后则扣去库存，然后解锁。此时，在大部分情况下，程序会一路顺序执行到流程结束，生成订单。在极少数情况下，程序可能会执行出错，此时通过程序控制回滚库存[37]。在极少数情况下牺牲数据的一致性换来了大部分情况下的性能优化。并且，这种情况下只可能出现多剩余了库存而不会导致库存不足超卖的情况发生，保证了良好的用户体验，避免出现用户抢购到商品却被告知系统出错库存不足的情况。

4.3.2 订单流程优化

在本文上一节，通过两段式事务解决了因为数据库事务导致的并发性损失问题。在本节中，将通过详细分析生成订单流程中的每一步 I/O 操作，尽可能省去不必要的 I/O 操作，优化程序流程。首先，流程第一步会先查询物流基本费，然而，由上述库存事务分析可知，如果在接下来的库存检查分配中无法满足库存，则此次数据库 I/O 就会导致系统资源的浪费。因此，应将库存检查放到订单流程第一步，在经过库存分配检查之后才能进行接下来的流程。同时，考虑物流基本费这一需求，并没有必要通过数据库 I/O 在每次订单过程中查询，可以通过将数据缓存到 Redis 中。接下来，根据 USERCODE 查询 VIPCODE 同样可以放到 Redis 中通过缓存查询。下一步，对每一个商品，为了准备促销引擎所需要的数据，进行了三次的数据库 I/O 查询操作，而在存储层缓存升级后，数据可以通过 Redis 查询到。对于两次到 MIS 系统的 HTTP 查询由于流程原因无法优化，接下来的写入订单事务由于强一致性的要求无法优化，因此，经过调整后的生成订单流程如下图 4-9 所示，经过流程优化的订单操作，首先会进行库存分配检查确保不会造成阻塞和系统资源的浪费。接下来，通过 Redis 缓存来减少数据库 I/O，最后在事务写入阶段，首先进行用户优惠券的计算以免由于并发情况下造成的用户优惠券使用失败从而造成大量数据库操作的回滚。通过优化之后的下单流程，将数据库 I/O 次数优化到了一次加锁库存查询及更新加上一系列连续写入的事务。

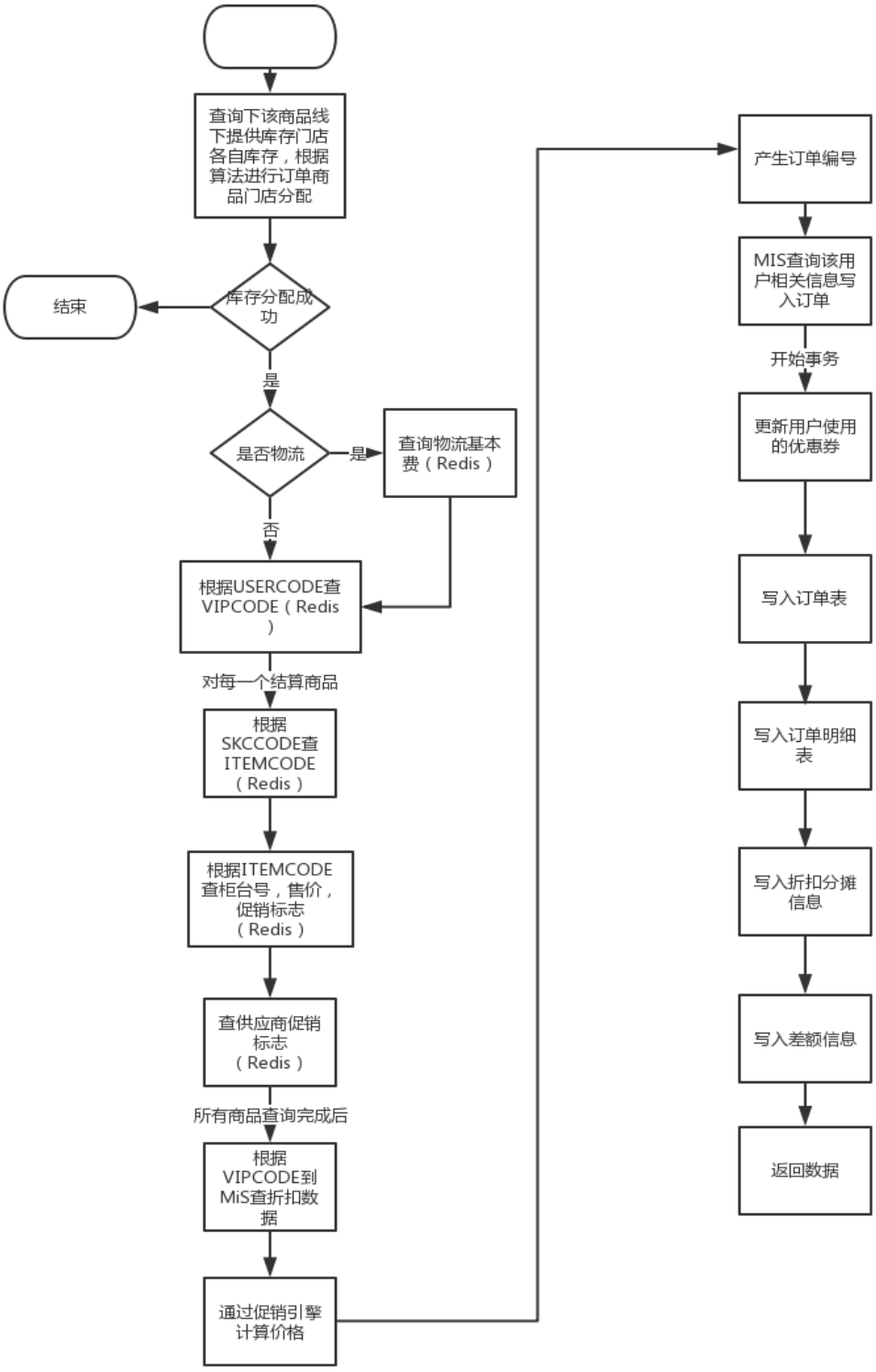

图 4-9 优化后生成订单流程图
Fig.4-9 Optimized place order flow chart

4.3.3 性能对比测试

为了验证对于订单库存系统优化的有效性，对优化后的订单库存系统进行对比测试。由于订单库存系统中的流程优化依赖于存储层 Redis 的引入，因此在本文 4.2.5 的引入存储层的环境基础上进行性能对比测试。由于优化后的性能提升，因此在测试时提高了系统的负载。如下表 4-4 所示为对比测试结果。

表 4-4 生成订单优化对比结果
Table 4-4 Place order comparison results

测试环境	每秒请求用户数	总请求用户数	成功响应用户数	成功响应平均时间	最快成功响应时间	最慢成功响应时间
原始测试	0.2	10	10	12.10sec	8.31sec	15.69sec
原始测试	1	10	10	13.52sec	8.70sec	17.28sec
对比测试	1	10	10	1.47sec	0.86sec	2.34sec
原始测试	10	100	100	1min7sec	46.14sec	1min45sec
对比测试	10	100	100	8.87sec	3.52sec	12.37sec
对比测试	100	1000	1000	1min23sec	47.26sec	1min52sec

通过上表的对比测试结果可以看到，通过对订单库存系统的流程和事务优化，大大提升了订单库存系统的性能和并发量，可以满足当前的性能需求。

4.4 本章小结

本章从服务器集群架构、数据库集群架构以及订单库存子系统三个方面对当前 O2O 交易系统进行了性能瓶颈的研究，在此基础上针对性的做出了架构升级和优化，低成本的实现了高并发、高可用、易拓展的 O2O 交易系统。

下一章，我们将对架构升级后的系统进行全面性能测试，并与本文第三章的测试结果进行对比，验证优化升级后的系统比原系统具有更好的响应时间和并发性。

第五章 实验验证与分析

本章将对优化升级后的系统进行整体基准测试，并与本文第三章中优化升级前的基准测试数据进行对比，验证优化升级方案的有效性。

5.1 测试环境与方案

测试环境在本文第三章测试环境的基础上，在数据库服务器上部署了 Redis，在 Node.js 服务器上集成了 PM2 模型，并启动 4 个进程进行服务。环境部署如表 5-1 所示：

表 5-1 对比测试环境部署
Table 5-1 Comparison test environment

机器@IP	处理器	内存	硬盘	系统环境
A@202.120.40.150	Xeon E5310, 1.6GHZ*8	32G	1T	Node Server(PM2, 4 进程), Tomcat(Solr, Drools)
B@202.120.40.142	Xeon E5405, 2GHZ*8	32G	1.5T	MySQL,Redis,Tsung

测试将采用与本文第三章中相同的测试方案进行测试，以对比验证优化实现的效果。

5.2 测试结果与分析

下面将根据本文第三章中对各接口根据性能表现分成的 A,B,C,D 四类分别进行测试结果对比及分析。

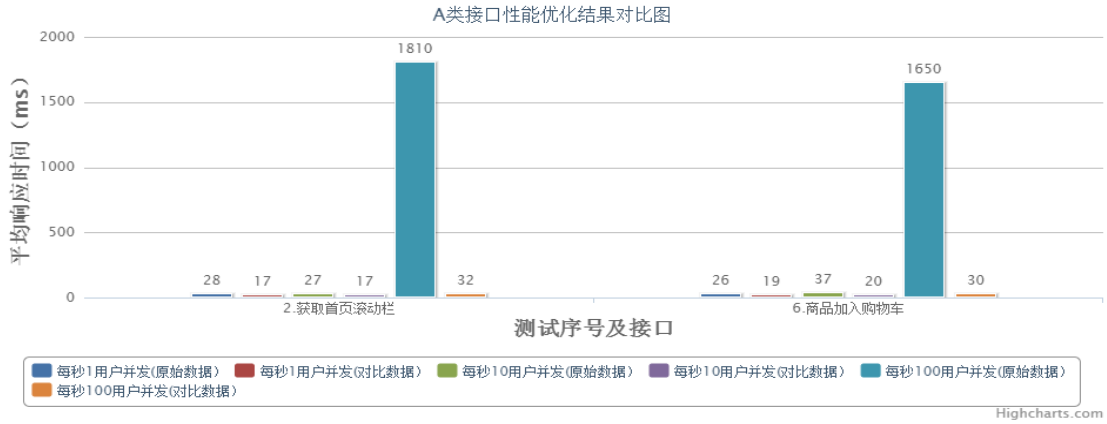

图 5-1 A 类接口性能优化结果对比图
Fig.5-1 Class A interfaces performance optimization comparison test results

如图 5-1 所示为 A 类接口原始测试结果与优化后的测试结果对比图，可以观察到由于 A 类接口本身性能表现较好，因此在低并发压力下的优化效果并不明显，但是随着并发压力的提升，优化后的系统架构表现出了明显的优势。由图中可以看出，每秒 100 并发请求的压力情况下，成功将平均响应时间由秒级别优化到了毫秒级别，在高并发下仍保持了接近于最佳情况下的性能水平。得益于多进程的加入，此前积压于一个进程的任务被分配到多个进程处理，既大大减少了每个进程的压力，也减少了每个请求的排队时间，导致高并发下的响应时间大大缩短。

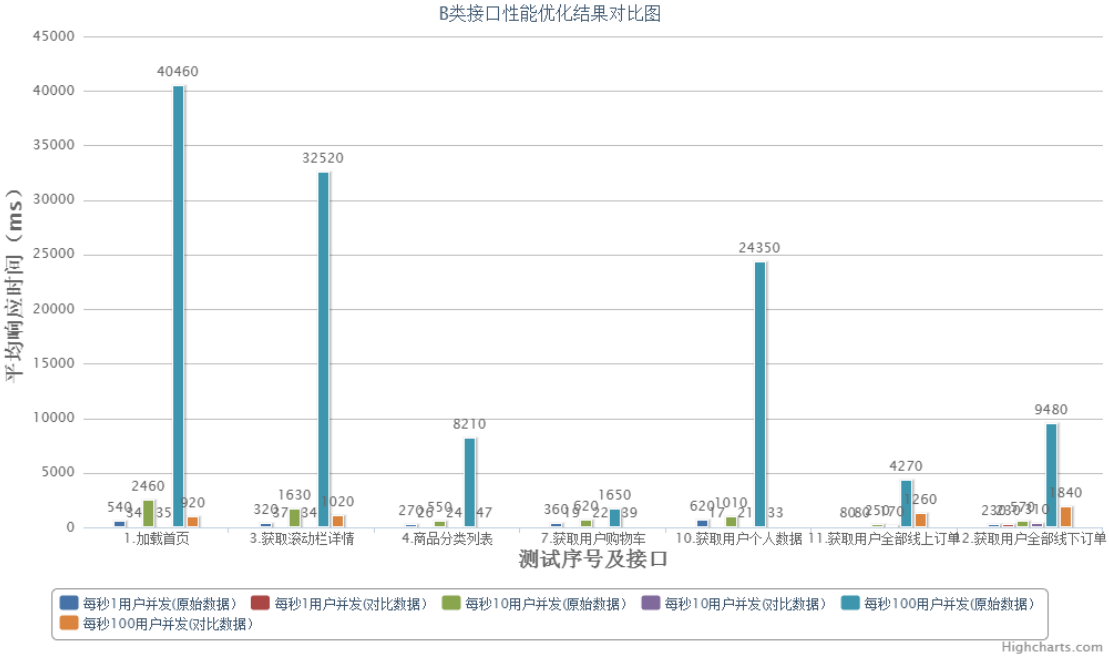

图 5-2 B 类接口性能优化结果对比图

Fig.5-2 Class B interfaces performance optimization comparison test results

如图 5-2 所示为 B 类接口性能优化结果对比图。从图中可以看到在低负载情况下，各接口都有更好的性能表现，平均响应时间由百毫秒数量级优化至十毫秒数量级，在高并发的情况下，则有明显的性能提升。可以看到大部分接口平均响应时间由秒，十秒数量级优化至十毫秒数量级，保持了很好的性能水平。可以看到通过将经常读取的信息缓存进 Redis，而不再通过 MySQL 视图查询，大大的提高了系统的性能表现。而其中 1,3 接口在高并发条件下性能相对于其他接口受到了更为严重的影响，通过研究分析，发现因为 1,3 接口取出数据后需要将数据通过模板生成为 html 文件然后返回，由此导致了性能下降更为明显，这可以作为后续的优化方向。同时，由于订单信息作为高一致性数据未移入 Redis 缓存，因此获取订单接口 11,12 只受益于服务器架构的优化。

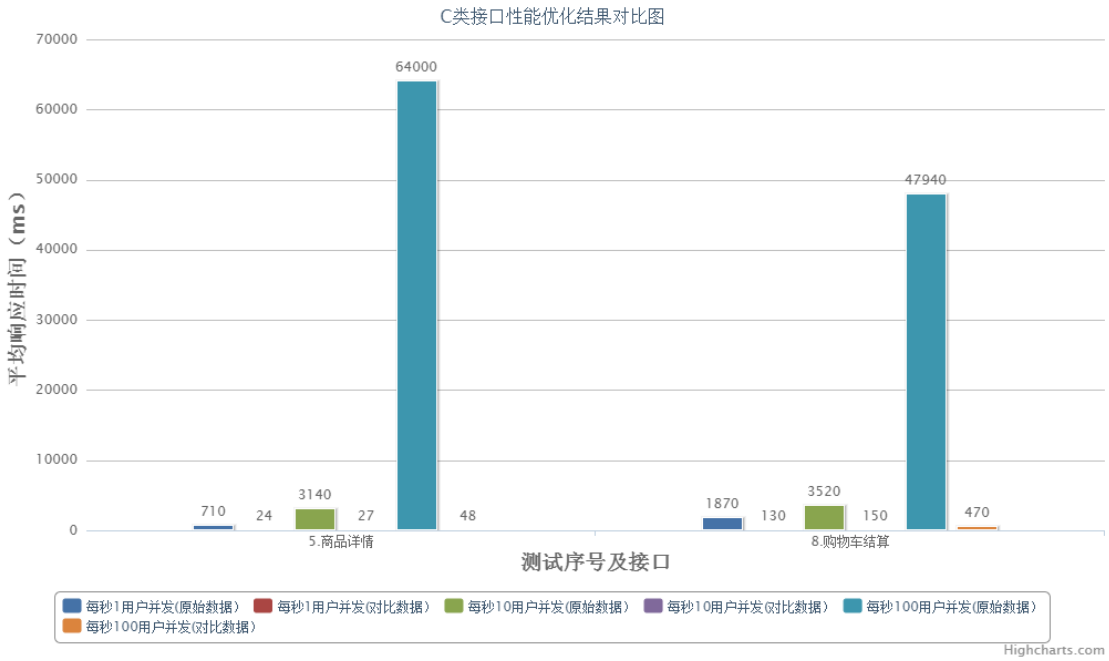

图 5-3 C 类接口性能性能优化结果对比图
Fig.5-3 Class C interfaces performance optimization comparison test results

如图 5-3 所示为 C 类接口性能优化结果对比图。从图中可以看出接口的性能得到了明显提升，尤其是商品详情接口在高并发条件下仍然能够保持很好的性能。通过将商品信息缓存在 Redis 中，在每次商品信息详情的查询过程中，大大节省了 MySQL 数据库 I/O 操作次数，极大的提升了性能。对于购物车结算接口，从本文第三章中对于该接口的流程分析可以看到，该接口中的处理流程是作为生成订单接口的子流程出现，因此得益于对订单系统的优化，该接口也性能也得到了明显的提升。然而，由于价格等数据属于强一致性数据，导致不可避免的 MySQL 数据库 I/O 以及流程上不可省略的两次 MIS 系统交互加一次促销引擎交互共三次 Http 请求，导致了在总体性能上落后于商品详情接口。

对于 D 类接口，根据本文 4.2.5 以及 4.3.3 小节的对比测试结果可以看到，对于用户浏览历史接口，通过将用户相关信息放入 Redis 中储存，不仅解决了由于 MySQL 视图数据量太大导致的服务超时未响应的问题，而且有很好的响应时间和并发性，验证了优化的成果。而对于生成订单接口，通过一系列的事务和流程优化，性能得到了显著的提升，虽然性能表现仍不是最佳，但可以满足目前的业务需求，如何继续提升性能可以作为后续研究方向。

5.3 本章小结

本章采用本文第三章中设计的测试方案，对架构优化升级后的系统再次进行了基准测试，并将测试结果与本文第三章中未优化升级前的测试结果进行对比，由对比测试结果可以验证及分析本文所研究的优化升级方案的有效性。

下一章我们将对本文的研究工作进行总结，并展望未来可以继续改进的方向。

第六章 总结与展望

6.1 全文总结

2015《互联网+流通行动计划》明确指出：支持大型实体零售企业利用电子商务平台开展网上订货实体店取货等业务。“互联网+零售”已经处于国家级战略高度。因此，许多传统零售企业纷纷开始布局进军电子商务领域，探索转型 O2O，实施“互联网+”行动计划，谋求转型升级。然而，传统线下零售企业在转型初期往往面临着技术储备不足，投入资金有限等问题，通常选择通过对线下零售系统的业务升级来转型 O2O。但是，传统线下零售系统架构难以满足 O2O 系统线上多用户高并发的性能要求。

本文以徐家汇汇金百货的交易系统为背景，介绍了其线下零售系统 RMS 系统到 O2O 交易系统 MEC 的业务升级以及系统技术架构。基于在 O2O 系统运营期间所暴露出的性能问题，根据业务场景设计了一套性能基准测试方案，并模拟真实生产环境搭建了一套测试环境，在测试环境中对当前系统的整体性能进行了测试。通过对测试结果的分析，找到了当前系统存在的性能瓶颈以及优化的方向，包括服务器架构，存储层架构以及订单库存系统。本文针对这些这些不足，研究优化升级了系统的设计与实现。

对于服务器架构的优化，通过对 Node.js 服务器平台的研究，发现了其所暴露出的对于系统硬件资源利用不充分的问题。针对此问题，通过调查和研究后提出了三种解决方案，经过分析对比后选择了基于 Node.js 的内置 cluster 模块来做出优化升级而不引入其它系统增加复杂性以及维护成本。在方案实现过程中，研究发现开源项目 PM2 可以满足目前我们的需求，因此选择了采用 PM2 来管理我们的服务器进程并维持服务器资源的高可用性。

对于存储层架构，通过对程序实现角度的分析，发现传统的关系型数据库设计过于遵守关系型数据库设计原则以及 ACID 特性，而没有考虑 O2O 系统对于信息获取的需求，从而导致了读请求过于复杂以及大量数据库视图的利用，而数据库视图并不能提供我们所需要的性能。针对此问题，在提出的多个设计方案中选择了对当前系统运营影响最小，成本最低的方案，即增加一层缓存层，尽量减少对于线下系统的改动而利用 O2O 系统读多写少的特点，优化系统的性能。

对于订单库存系统，通过对生成订单执行流程的分析，发现当前的执行流程

存在设计不合理的缺陷，通过对订单库存系统的事务以及流程进行优化，提升了订单系统的性能。

最后，通过使用相同的测试方案对优化升级后的系统进行对比测试，验证了优化方案的有效性，对 O2O 交易系统的性能优化与实现达到了研究目标。

6.2 工作展望

在优化方案的实现以及优化后的测试过程中，我们发现对于获取首页以及滚动栏活动详情这两个接口取出数据库数据后根据模板生成了 HTML 格式的文件并返回，这导致了这两个接口在高并发下的表现要更差。而对于首页以及活动详情这种大量读取的请求，后续可以考虑将 HTML 静态文件缓存以优化性能。

对于订单库存系统，可以发现经过优化后虽然性能大大提升，但是相对于其它接口而言仍然性能较差，后续的工作可以着重对此进行优化。比如在高并发的时候可以考虑加入请求队列，对于过多的请求直接返回以保证服务的可用性以及用户体验。而通过流程优化中可以发现，执行流程中在插入数据库记录的事务之前所有订单操作已执行完成，此时可以考虑加入消息队列，将需要插入的数据加入消息队列交给其他进程来处理，而当前的请求可以直接返回。

参 考 文 献

[1] 郭燕, 陈国华, 陈之昶. "互联网+"背景下传统零售业转型的思考[J]. 经济问题, 2016(11):71-74.

[2] Du Y, Tang Y. Study on the Development of O2O E-commerce Platform of China from the Perspective of Offline Service Quality[J]. International Journal of Business and Social Science, 2014, 5(4): 308-312.

[3] 程芷依. 我国传统零售业向 O2O 模式转型升级研究[D]. 哈尔滨商业大学, 2015.

[4] Chen M. A Dynamic E-commerce System Based on Middleware Technology[C]//Networks Security, Wireless Communications and Trusted Computing, 2009. NSWCTC'09. International Conference on. IEEE, 2009, 2: 733-736.

[5] Tilkov S, Vinoski S. Node. js: Using JavaScript to build high-performance network programs[J]. IEEE Internet Computing, 2010 (6): 80-83.

[6] Lee K D. Event-Driven Programming[M]//Python Programming Fundamentals. Springer London, 2011: 149-165.

[7] Zhang J H, Zhang Y M. IOCP machanism study and application on the scalable winsock communication system[J]. Computer & Modernization, 2004.

[8] Dahl R. Node. js: evented I/O for v8 javascript[J]. URL: https://www. nodejs. org, 2012.

[9] Buettner D, Kunkel J, Ludwig T. Using Non-blocking I/O Operations in High Performance Computing to Reduce Execution Times[C]// Recent Advances in Parallel Virtual Machine and Message Passing Interface, European Pvm/mpi Users' Group Meeting, Espoo, Finland, September 7-10, 2009. Proceedings. 2009:134-142.

[10] Sun J. TCP/IP Socket I/O Multiplexing Using a Hybrid Polling Event System[J]. 2009:277-284.

[11] Yoo H C. Comparative analysis of asynchronous I/O in multithreaded UNIX[J]. Software Practice & Experience, 1996, 26(9):987–997.

[12] Node.js 的事件驱动模型.
http://www.edwardesire.com/2015/05/09/nodejs-event-model/.

[13] 许会元, 何利力. NodeJS 的异步非阻塞 I/O 研究[J]. 工业控制计算机, 2015, 28(3):127-129.

[14] Lee K D. Event-Driven Programming[M]//Python Programming Fundamentals. Springer London, 2011: 149-165.

[15] Codd E F. A relational model of data for large shared data banks[J]. Communications of the ACM, 1970, 13(6): 377-387.

[16] MySQL View. http://dev.mysql.com/doc/refman/5.5/en/view-algorithms.html.

[17] Strozzi C. NoSQL-A relational database management system[J]. Lainattu, 1998, 5: 2014.

[18] "NoSQL DEFINITION: Next Generation Databases mostly addressing some of the points: being non-relational, distributed, open-source and horizontally scalable". http://nosql-database.org/

[19] Brewer E. Pushing the cap: Strategies for consistency and availability[J]. Computer, 2012, 45(2): 23-29.

[20] Gilbert S, Lynch N. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services[J]. ACM SIGACT News, 2002, 33(2): 51-59.

[21] Vogels W. Eventually consistent[J]. Communications of the ACM, 2009, 52(1): 40-44.

[22] Mazumder S. NoSQL in the Enterprise[J]. InfoQ. http:// www. infoq. com/ articles/ nosql-in-the-enterprise. Accessed, 2015, 6.

[23] Chang F, Dean J, Ghemawat S, et al. Bigtable: A distributed storage system for structured data[J]. ACM Transactions on Computer Systems (TOCS), 2008, 26(2): 4.

[24] Lakshman A, Malik P. Cassandra: a decentralized structured storage system[J]. ACM SIGOPS Operating Systems Review, 2010, 44(2): 35-40.

[25] Armstrong J. A history of Erlang[C]//Proceedings of the third ACM SIGPLAN conference on History of programming languages. ACM, 2007: 6-1.

[26] Iqbal M A, Saltz J H, Bokhart S H. Performance tradeoffs in static and dynamic load balancing strategies[J]. 1986.

[27] Saxena N, Bhargava N, Mahor U, et al. An Efficient Technique on Cluster Based Master Slave Architecture Design[C]//Computational Intelligence and Communication Networks (CICN), 2012 Fourth International Conference on. IEEE, 2012: 561-565.

[28] Lamport L. Paxos made simple[J]. ACM Sigact News, 2001, 32(4): 18-25.

[29] Reese W. Nginx: the high-performance web server and reverse proxy[J]. Linux Journal, 2008, 2008(173): 2.

[30] Merkel D. Docker: lightweight linux containers for consistent development and deployment[J]. Linux Journal, 2014, 2014(239): 2.

[31] Supervisord. http://supervisord.org/.

[32] Chaudhuri S, Shim K. Optimizing queries with aggregate views[C]//International Conference on Extending Database Technology. Springer Berlin Heidelberg, 1996: 167-182.

[33] Stensby A M. Scalability and Horizontal Partitioning[J]. 2009.

[34] Fitzpatrick B, Vorobey A. Memcached: a distributed memory object caching system[J]. 2011.

[35] Zawodny J. Redis: Lightweight key/value store that goes the extra mile[J]. Linux Magazine, 2009, 79.

[36] Chodorow K. MongoDB: the definitive guide[M]. " O'Reilly Media, Inc.", 2013.

[37] Al-Houmaily Y J, Samaras G. Two-phase commit[M]//Encyclopedia of Database Systems. Springer US, 2009: 3204-3209.

致 谢

时光荏苒，白驹过隙，研究生两年半的生活很快就要结束了。在这段时间里，我所经历的、感悟的、收获的都是我一生宝贵的财富。在这篇硕士论文从选题到最终撰写完成的一年时间里，得到了很多人的关心支持和鼓励。在本文的最后，我要由衷的感谢所有在我学习工作生活中给予我帮助、带给我成长、一直鼓励着我的人们。

首先，我要感谢我的导师吴刚副教授。研究生两年半的时间，吴老师给予我非常多的帮助和指点。无论是学业上的迷津，或是项目中的难点，亦或是本文写作中的困难，导师都对我耐心帮助指点。导师令我敬佩的不仅仅是专业能力，还有严谨的治学态度和钻研精神，对待科研问题一丝不苟，而这一切对我们都是一种潜移默化。吴刚老师不仅仅是我们的导师，更是我们的榜样和朋友。

其次，我要感谢实验室里的所有同学。在两年半的时间里，在课程学习，项目进展和科研实验中，大家都相互帮助，共同奋斗，一起攻克了很多困难。所谓三人行必有我师，从大家身上我学会了很多专业知识和做人做事的道理。

另外还要感谢我的父母对我的支持、关系和理解，让我能够更加专注的学习工作，他们的关怀和鼓励是我前进的动力，使我在困难面前始终保持一颗沉着的心。

最后，由衷地感谢各位参与论文评审的专家和老师，非常期待你们的耐心指点和宝贵建议，感谢你们百忙之中抽出时间审阅本文。

攻读硕士学位期间已发表或录用的论文

[1] Li Junnan, Wu Gang. Benchmark and Optimization Implementation for O2O Commercial System [C]. International Conference on Computation Techniques in Information and Communication Technologies (ICCTICT), 2017(已录用)