

西安电子科技大学

硕士学位论文



基于SpringBoot的电商秒杀系统的设计与实现

作者姓名 李宜稼

学校导师姓名、职称 赵明英 副教授

企业导师姓名、职称 王双平 高工

申请学位类别 工程硕士

学校代码 10701
分 类 号 TP31

学 号 19041212139
密 级 公开

西安电子科技大学

硕士学位论文

基于 SpringBoot 的电商秒杀系统的设计与实现

作者姓名：李宜稼

领 域：控制工程

学位类别：工程硕士

学校导师姓名、职称：赵明英 副教授

企业导师姓名、职称：王双平 高工

学 院：机电工程学院

提交日期：2022 年 6 月

Design and implementation of e-commerce seckill system based on SpringBoot

A thesis submitted to
XIDIAN UNIVERSITY
in partial fulfillment of the requirements
for the degree of Master
in Engineering

By

Li Yijia

Supervisor: Zhao Mingying Title: Associate Professor

Supervisor: Wang Shuangpin Title: Research Fellow

June 2022

西安电子科技大学
学位论文独创性（或创新性）声明

秉承学校严谨的学风和优良的科学道德，本人声明所呈交的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中所罗列的内容以外，论文中不包含其他人已经发表或撰写过的研究成果；也不包含为获得西安电子科技大学或其它教育机构的学位或证书而使用过的材料。与我一同工作的同事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谢意。

学位论文若有不实之处，本人承担一切法律责任。

本人签名: 李宜稼 日期: 2022. 6. 20

西安电子科技大学
关于论文使用授权的说明

本人完全了解西安电子科技大学有关保留和使用学位论文的规定，即：研究生在校攻读学位期间论文工作的知识产权属于西安电子科技大学。学校有权保留送交论文的复印件，允许查阅、借阅论文；学校可以公布论文的全部或部分内容，允许采用影印、缩印或其它复制手段保存论文。同时本人保证，结合学位论文研究成果完成的论文、发明专利等成果，署名单位为西安电子科技大学。

保密的学位论文在____年解密后适用本授权书。

本人签名: 李宜稼 导师签名: 赵明渠
日期: 2022. 6. 20 日期: 2022. 6. 20

摘要

随着互联网技术的高速发展，线上购物作为电子商务的重要一环取得了突飞猛进的发展。由于具有高效便捷的优点，在线购物已经成为一种不可或缺的新型生活方式，近年来各大互联网企业旗下的电商平台更是在“双十一”、618年中大促等新兴的购物节日一再达成巨额交易量，获得了巨大成功。而对于这些平台来说，如何在保证交易不出错的前提下应对短时间内的大流量访问，确保交易系统的高性能成为了技术难点。

本文主要叙述了基于 SpringBoot 框架开发的线上购物商城的实现过程，着重于解决商城系统应对高并发访问的技术难题。使用的编程语言为 Java 和 Lua，数据库使用的是 MySQL 和 Redis。首先设计了数据库表用来存放注册的用户信息，商品信息以及促销信息等，并且为了提升性能，增强可用性，对数据库进行了分表处理。然后采用了 SpringMVC 的架构整合了持久层、表示层、以及业务层，完成了注册、登录、下单与促销等基础功能的实现。接着引入了分布式架构，通过 Nginx 服务器将用户的请求反向代理，分发到两台采取了轮询策略的服务器上完成了水平扩展，减轻了服务器负载，提高了访问效率；同时对前端资源进行了动静分离，提升了系统的响应速度，并且通过使用更符合现代互联网传输特色的 token+Redis 的结构实现分布式会话管理，便于对登陆到商城的用户进行管理。接着讨论了不同的多级缓存方案并选择了以 Nginx SharedDic、Redis 以及 GuavaCache 组成的三级缓存用来优化查询能力、提升系统性能；在引入缓存后研究了不同的数据一致性方案，最终选择了消息队列和事务型消息技术确保了库存与缓存数据的一致性，再辅以业务逻辑和代码层面的优化，使商城系统的性能有质的提升。

最后一部分使用 Jmeter 设置线程组模拟高并发的使用场景，分别对商城系统的查询、数据一致性以及数据库服务器进行压力测试，结果证明足以应对高峰流量的冲击。并且对系统的后续优化进行了展望。

关 键 词：SpringBoot 框架， 高并发， Redis， 多级缓存

ABSTRACT

With the rapid development of Internet technology, online shopping as an important part of e-commerce has made rapid progress. With the advantages of efficiency and convenience, online shopping has become an indispensable new way of life. In recent years, the e-commerce platforms of major Internet companies have achieved great success by repeatedly reaching huge transaction volumes in emerging shopping festivals such as the "Double 11" and the 618 mid-year promotion. For these platforms, the technical challenge is how to cope with the high traffic volume in a short period of time and ensure the high performance of the transaction system while ensuring error-free transactions.

This paper describes the implementation of an online shopping mall based on the SpringBoot framework, focusing on the technical challenges of dealing with high concurrent access to the mall system. Java and Lua were used as the programming languages and MySQL and Redis were used as the database, and database tables were designed to store registered user information, product information and promotion information. The SpringMVC architecture was then used to integrate the persistence, representation and business layers, completing the basic functionality of registration, login, order placement and promotions. The distributed architecture was then introduced, with user requests being reverse-proxied through an Nginx server and distributed to two servers with a polling strategy to reduce server load and improve access efficiency. The distributed session management is achieved by using a token+Redis structure that is more in line with modern Internet transport characteristics and facilitates the management of users logging into the mall. Different multi-level caching options were then discussed and a three-level cache consisting of Nginx SharedDic, Redis and GuavaCache was chosen to optimise query capabilities and improve system performance; after introducing the cache, different data consistency options were investigated and message queues and transactional messaging techniques were chosen to ensure consistency between stock and cached data, supplemented by This was complemented by business logic and code level optimisation, resulting in a qualitative improvement in the performance of the mall system.

The final section uses Jmeter to set up thread groups to simulate high concurrency usage scenarios, and stress tests are conducted on the mall system's queries, data consistency and database server respectively, and the results prove to be sufficient to cope with the onslaught of peak traffic. The results proved to be sufficient to cope with peak traffic.

Keywords: SpringBoot framework, High concurrency, Redis, Multi-level cache

插图索引

图 2.1	Spring 框架模块	6
图 2.2	SpringMVC 三层架构	7
图 2.3	SSM 框架	8
图 2.4	bio 模型	10
图 2.5	epoll 模型	11
图 2.6	master-worker 进程模型	11
图 3.1	商品模型关系图	16
图 3.2	通用返回类图	17
图 3.3	CommonError 类图	18
图 3.4	返回错误信息以及异常处理流程图	19
图 3.5	用户不存在返回信息	19
图 3.6	成功返回用户信息	19
图 3.7	注册成功界面	21
图 3.8	注册成功的用户信息表	21
图 3.9	注册成功的用户密码表	22
图 3.10	用户登陆成功界面	23
图 3.11	年龄校验效果	24
图 3.12	商品创建成功界面	26
图 3.13	商品列表界面	27
图 3.14	促销活动信息表	28
图 3.15	促销商品界面	30
图 3.16	系统总体结构框图	31
图 4.1	Nginx 配置文件内容	34
图 4.2	Nginx 前端资源路由	34
图 4.3	Nginx 反向代理	35
图 4.4	Server 块的配置	35
图 4.5	Location 块的配置	36
图 4.6	基于 cookie 传输 session 原理图	36
图 4.7	localStorage 存储 token	37
图 4.8	Redis 商品详情页	38
图 4.9	proxyCache 配置	40

图 4.10 Nginx proxy cache 缓存	41
图 4.11 多级缓存访问流程图	42
图 4.12 CAP 理论	43
图 4.13 先更新数据库再更新缓存方案	45
图 4.14 先更新缓存再更新数据库方案	45
图 4.15 先删除缓存再更新数据库方案	46
图 4.16 先更新数据库再删除缓存方案	46
图 4.17 RocketMQ 原理	47
图 4.18 MqProducer 类图	49
图 4.19 MqProducer 类图	49
图 4.20 异步扣减库存流程图	50
图 4.21 创建订单事务与扣减库存事务关系图	51
图 4.22 扣减库存方案一	51
图 4.23 事务型消息方案流程图	52
图 5.1 Jmeter 线程组设置界面	56
图 5.2 查询测试设置	58
图 5.3 直接访问数据库的性能检测图	58
图 5.4 优化后的数据库服务器性能图	59

表格索引

表 2.1	Redis 数据类型与应用场景	9
表 3.1	商品详情表	13
表 3.2	商品库存表	14
表 3.3	用户信息表	14
表 3.4	用户密码表	14
表 3.5	订单信息表	15
表 3.6	促销信息表	15
表 3.7	枚举错误信息表	18
表 3.8	用户模型校验表	24
表 3.9	创建商品校验表	25
表 5.1	硬件配置表	55
表 5.2	查询实验表	57
表 5.3	缓存一致性实验表	59

符号对照表

符号	符号名称

缩略语对照表

缩略语	英文全称	中文对照
MD	Message Digest Algorithm	信息摘要算法
JDBC	Java DataBase Connectivity	Java 数据库连接
HTML	Hyper TextMarkup Language	超文本标记语言
XML	Extensible Markup Language	可扩展标记语言
IO	Input Output	输入输出

目录

摘要	I
ABSTRACT	III
插图索引	V
表格索引	VII
符号对照表	IX
缩略语对照表	XI
第一章 绪论	1
1.1 研究背景及意义	1
1.2 国内外发展现状	1
1.3 本文主要工作及内容安排	3
第二章 相关理论及技术介绍	5
2.1 加密算法 MD5 与 MD5 加盐算法	5
2.1.1 MD5 加密算法.....	5
2.1.2 MD5 加盐算法.....	5
2.2 开发框架	6
2.2.1 Spring 框架的介绍	6
2.2.2 SpringMVC 框架的介绍	7
2.2.3 SpringBoot 框架的介绍.....	7
2.2.4 MyBatis 框架的介绍	8
2.2.5 SSM 框架的介绍	8
2.3 数据库技术	8
2.3.1 MySQL 数据库的介绍	8
2.3.2 Redis 数据库的介绍	9
2.4 Nginx 服务器的介绍	9
2.5 RocketMQ 消息中间件的介绍.....	12
2.6 本章小结	12
第三章 秒杀商城基础项目的构建	13
3.1 数据库, 领域模型以及通用返回类型的设计	13
3.1.1 数据库的设计	13
3.1.2 领域模型的设计	16
3.1.3 通用返回对象的设计	17

3.2	商城基础功能的实现	20
3.2.2	登陆功能的实现	22
3.2.3	校验功能的实现	23
3.2.4	商品模块功能的实现	25
3.2.5	交易下单功能的实现	27
3.2.6	促销活动功能的实现	28
3.3	本章小结	30
第四章	交易与查询性能优化	33
4.1	分布式扩展	33
4.1.1	Nginx 的动静分离	33
4.1.2	Nginx 反向代理	34
4.1.3	基于 token 的分布式会话管理	36
4.2	多级缓存	37
4.2.1	Redis 缓存商品详情页	38
4.2.2	本地热点数据缓存的接入	38
4.2.3	Nginx 缓存的接入	40
4.3	缓存数据一致性	42
4.3.1	CAP 理论的介绍	43
4.3.2	数据一致性方案	44
4.4	消息队列异步扣减库存	47
4.4.1	异步消息队列的介绍	47
4.4.2	库存缓存异步化的设计与实现	48
4.5	事务型消息	50
4.5.1	事务型消息的引入	50
4.5.2	RocketMQ 实现事务消息	53
4.6	本章小结	53
第五章	性能测试	55
5.1	硬件测试环境	55
5.2	压测工具介绍	55
5.3	并发查询压测实验	57
5.4	数据库服务器查询压测实验	58
5.5	库存缓存数据一致性实验	59
5.6	本章小结	60
第六章	总结与展望	61

目录

6.1 工作总结	61
6.2 研究展望	61
参考文献	63
致谢	65
作者简介	67

第一章 绪论

1.1 研究背景及意义

伴随着近年来互联网产业的极速发展，全球网民数量激增，网络逐渐融入人们的日常生活，成为了不可或缺的一部分。就我国而言，根据中国互联网络信息中心（CNNIC）第 49 次《中国互联网络发展状况统计报告》，截至 2021 年 12 月份，中国已具有 10.3 亿的网民数据规模，国民使用互联网的占比高达 64.5%，光纤通网和 4G 网络下乡在各级地方单位使用比例均高达 98%，用户使用固定互联网宽带的规模超过 4.5 亿户^[1]。目前我国网民数量已经居全国第一，并且远超排名第二的美国。这样的背景下自然催生了互联网企业的火爆，我国的互联网公司也不遑多让，特别是在我国政府大力推行“互联网+”的背景下我国的互联网技术水平更是在各个领域都取得了长足的进步。在这样的浪潮的驱使下，电子商务的发展自然也是日新月异，在短短几年的时间，电子商务的交易额已经扩大了几十倍^[2]。得益于互联网技术的成熟以及网上购物平台的便捷性，线上购物开始走进千家万户，变得越来越日常化。

以上一系列条件的成熟催生了“双十一”，618 年中大促等众多购物节的火爆，消费者在购物节以优惠价格买到心仪的商品，商家也借此机会进行促销，电商平台也获得巨额收入，同时还能带动一系列周边产业的发展，例如物流业，可谓一举多得，因此颇具人气。根据天猫和京东发布的消息，2021 年 11 月 1 日零时，天猫“双十一”总交易额定格在 540 亿元；京东“双十一”累计下单金额超 3491 亿元^[3]。而其中大量交易都集中在极短的时间内，这便是所谓的“秒杀”，即电商卖家发布一些限时促销的低价商品，所有买家等待在促销活动开始在电商平台抢购的一种促销手段。由于商品价格低廉，往往一上架就被抢购一空^[4]。由于短时间内要面对大流量的冲击，“秒杀”促销手段无疑对商城系统的并发性能提出极高的要求，这样的应用场景给服务器带来了巨大的压力。若想要让服务器能更好地应对以上情况，可以尝试从多个角度出发进行考量，例如，进行业务拆分、使用集群式服务器群、负载均衡、对用户限流等方法。但是根本问题还是要设计出能够应对高并发访问压力的 Web 应用。同时，该系统还需具备一定的扩展能力，以便于应对平台未来发展可能带来的更多客流量。若设计不当，则很容易造成响应过慢、订单出错甚至系统宕机等一系列致命错误，不光会影响消费者的购物体验，甚至还可能造成不可估量的经济损失。

1.2 国内外发展现状

电子商务起源于 20 世纪 70 年代末的美国，在全球所有电子交易额中，目前约有

50%以上都发生在美国。作为电子商务的领头羊,美国的电子贸易发展经历四个阶段^[5],第一阶段 1991-1994 年,该阶段是美国电子商务的萌芽期,在此期间伴随因特网的诞生,众多互联网公司如雨后春笋般冒头,此时的互联网公司多靠汇集信息吸引浏览者,通过广告获得收入。彼时电子商务使用的技术栈极为落后,用户通过访问零星的静态网页进行商品的挑选,一般只能用作展示,并没有太高的实用价值。

第二阶段 1995-1999 年, B2C (Business to customer) 直接面向消费者的零售模式兴起。根据官方数据,1999 年第四季度 B2C 交易额已超过 50 亿美元。该阶段伴随着 Java, Python 和 HTML 等编程语言和数据库技术的出现,网站技术有了日新月异的发展,线上购物变得更加便捷。该阶段网站大都是单一架构,即应用程序,数据库以及静态文件全部都部署在一个服务器上。由于系统缺乏模块化设计,导致系统整体难以解耦,使得系统的开发和部署都较为麻烦,并且受单台机器性能制约,并发性与可用性都很差。此时的代表性开发套件为 Linux 操作系统, Apache 网页服务器, MariaDB 数据库和 PHP 脚本语言。

第三阶段 2000-2004 年,是属于 B2B(Business to business)的时代,企业与企业之间通过线上平台进行交易。B2B 模式直到今天也依然是电子商务的重要组成部分。此时电子商务开始趋于成熟,也出现了众多优秀的技术手段,越来越多的电商公司开始重视用户体验,网页界面的设计也越来越美观。更重要的一点,伴随着个人电脑的逐渐普及,一些热门网站,包括线上购物网站的访问量开始水涨船高,单机服务器的性能瓶颈被暴露了出来,集群式服务器,应用与数据库分离的思想开始被提出。

第四阶段则是转型期,各大传统行业都投入到轰轰烈烈的互联网化的进程中。目前为止美国知名的跨境电商平台有亚马逊、eBay、沃尔玛、梅西百货等,可以看出其中不少都是非传统互联网公司。网站的功能越来越丰富,服务器的性能也越来越优秀。这一阶段分布式开始被应用于各大网络程序,垂直与水平扩展开始被广泛应用;读写分离,前后端分离,分库分表等设计思想在这一阶段出现。

我国的电商起步较晚,2000 年初我国最重要的两大电商企业:淘宝和京东相继诞生,2008 年我国成为全球网名最多的国家,电子商务交易额自然也是水涨船高突破 3 万亿人民币。经过十年的摸爬滚打,2010 年进入高速增长期有了移动支付手段也让线上交易更便捷,2014 年底我国电商行业已经全面超越欧盟以及日本。进入 2015 年后增长速度趋于平稳。目前我国较为成功的电商品牌有淘宝、京东和拼多多。在多年的发展中,电商平台使用的技术自然也经历了多次迭代和升级。例如为了提升系统的健壮性和可扩展性所采用的分布式扩展;为防止单点服务器故障而出现的计算机集群结构等。2014 年提出的微服务设计架构吸引了众多目光,微服务指的是将整个业务拆分成若干个小业务,每个服务器或者服务器集群只负责其中一个环节。这一思想如今已经被很多网络公司旗下的业务所使用,成为跟集群式,分布式等设计架构齐名

的架构。可以预见的是，伴随着 5G 时代的到来，信息的传递会变得更加迅捷，大数据分析会变得更加精准，电商平台必将掀起一轮新的技术革命。

1.3 本文主要工作及内容安排

第一章：绪论。本章先介绍了该题目的研究价值，然后介绍了国外电商平台的发展历程与我国电商平台的发展现状。

第二章：基础理论以及相关技术介绍。本章主要介绍了全文会用到的加密算法、数据库技术、服务器技术以及开发框架和架构和使用到的消息中间件等。

第三章：电商基础项目的构建。本章开头先是设计了数据库表、数据模型和自定义的通用返回类型，然后在此基础上完成了注册、登录、信息校验、商品创建、商品浏览和下单功能，还设计了促销模块，让商城具备了基础的功能。

第四章：查询与下单性能优化。针对高并发场景下的秒杀商城两个关键操作：查询商品以及下单减库存进行优化。首先对前端资源进行了动静分离，然后使用了分布式架构对原本的单机系统进行了水平扩展并优化了会话管理方式，接着讨论了不同的多级缓存方案。同时为了解决引入缓存后带来的下单后缓存与数据库库存数据不一致的问题使用了消息队列来确保下单扣减库存操作的数据一致性。

第五章：性能测试。使用 JMeter 压测软件模拟高并发场景，发起查询和下单扣减库存的请求，通过聚合报告中的吞吐量和响应时间等指标分析系统在高并发压力下的性能表现并验证前文优化方案的有效性。

第六章：总结与展望。对本文秒杀商城的基础功能的设计以及性能优化进行了总结，然后提出了一些可行的后续优化方案。

第二章 相关理论及技术介绍

本章将对商城系统所用到的相关技术和软件产品进行介绍。主要包括以下内容：使用到的加密算法 MD5 加盐算法。以及开发框架，包括 Spring 框架、SpringMVC、SpringBoot 等以及本文使用到的数据库，包括 MySQL 和 Redis，用到的服务器技术以及消息中间件的介绍，并阐述了选用理由和使用场合。

2.1 加密算法 MD5 与 MD5 加盐算法

2.1.1 MD5 加密算法

MD5 加密算法，全称是 Message-Digest Algorithm5，是应用最为广泛的 Hash 算法之一^[6]，由 Ronald Rivest 在 1991 年提出^[7]，是 MD4 算法的升级版。MD5 可以将一个长度随机的输入值转换成一个 16 字节的值^[8]，通常用于加密密码和生成电子签名。MD5 本质上是一个散列函数，加密使用的是哈希算法，从密码原文到加密后的密文是单向映射，该过程完全不可逆^[9]，不仅如此，MD5 加密过程本身就是一个有损的加密过程，因此几乎不能还原出原始数据，所以即便网络攻击者看到了数据库中存储的密码密文也无法逆推出密码原码。

2.1.2 MD5 加盐算法

MD5 算法也有致命的缺点，因为其本质上是一种哈希算法，所以它也有所有哈希算法的缺陷，即易出现哈希冲突，特别是当加密的数据量较大时这种情况出现的更频繁，这使得 MD5 算法的破解成为可能，而这种可能性在 2004 年就得到了证实。现在使用穷举法或者字典树都可以轻易破解。经过研究，人们又提出了 MD5 加盐算法。

MD5 加盐算法就是在原来 MD5 的基础上插入“salt”，又称盐值，即随机生成的字符串，其中可以包含字符、数字和英文字母等，使用数字的场合更多。由于是随机生成的，所以极大程度上增加了加密的复杂度和安全性。本文主要使用 MD5 加盐算法对用户密码进行加密。具体的实现思想为，在用户密码进行入库时，使用 Random 函数生成一个随机的固定长度的数字作为 salt，这样可以确保每个密码的唯一性^[10]；然后进行“加盐”，将盐值与原本 MD5 加密的密码拼接起来，拼接位置并不固定，可以不是简单的拼接到原码前面或者后面，也可以拼接在中间的某位，这样的处理也使得密码复杂性更高。本文使用 MD5 加盐算法对用户密码进行加密

2.2 开发框架

2.2.1 Spring 框架的介绍

Spring 是一款轻量级的 Java 框架，Spring 框架小巧而且灵活，是为了简化 Java 企业开发而诞生的。它使用一个特殊的类 JavaBean 来管理对象，将 Java 开发从繁杂的配置工作中解放出来^[11]。现在 Spring 已经是服务端开发不可获取的开发工具。Spring 框架包含了 7 个模块如图 2.1 所示：

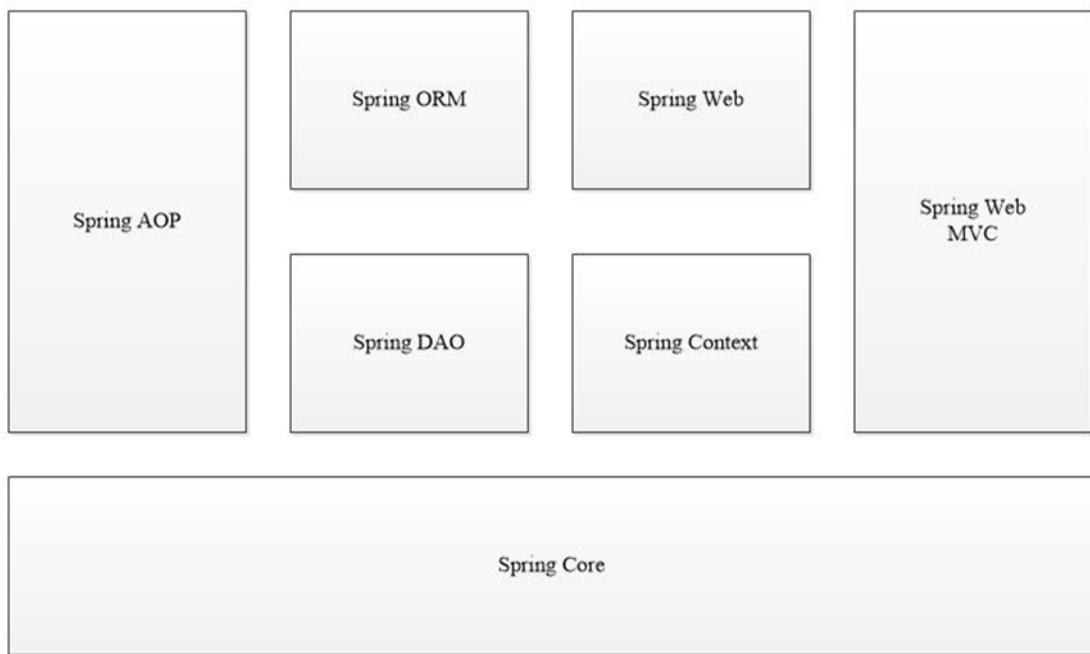


图2.1 Spring 框架模块

Spring 有两个重要的特性,控制反转和面向切面^[12]。

IOC:Inversion of Control, 又称控制反转, 是面向对象编程思想中的中的一种设计方法, 其思想为将对象的管理权交给 IOC 容器, 开发者只关注业务逻辑本身的实现即可。在启动 Spring 时, 会由 IOC 容器帮助对象找到相对应的依赖对象并注入, 而不是由对象主动去找, 这一过程称为依赖注入。依赖注入常用的方法由三种, **constructor** 构造方法注入、属性注入以及接口注入。本文主要使用的是构造方法注入。

AOP:Aspect Oriented Programming, 又称面向切面编程。AOP 事实上并非一种编程技术, 而是一种编程思想^[13]。即将与业务逻辑本身契合度不高但是又必不可少的代码块从项目中“切”出来, 需要的时候可以单独对这个切面进行功能修改而不影响到项目主干内容, 然后再将该切面引入原工程即可。同时该思想还能减少代码量, 使得代码更工整、易懂, 可读性更强^[14]。

2.2.2 SpringMVC 框架的介绍

SpringMVC 是 Spring 提供的一个基于 MVC 设计模式的轻量级 Web 开发框架。MVC 设计模式，M（Model）指数据模型层，V（View）指视图层，C（Controller）指控制层，每一层分别负责不同的功能。

视图层：是用户与系统交互的主要场所，负责将从控制层调取需要的数据并把它们呈现给用户，主要是一些使用 HTML、Flash 或者 XHTML 技术制作的界面。

控制层：控制层本身并不作出任何处理，主要负责接收视图层传入的请求并决定调用特定的模块去对处理请求，然后再指定视图并将结果发送给浏览器。

数据模型层：该层是整个 MVC 架构中最核心的部分，它主要负责封装数据并对数据进行操作。使用 MVC 结构的目的是为了将不同的业务分配给不同的层，实现业务的解耦。

视图层、控制层和数据模型层的关系如图 2.2 所示：

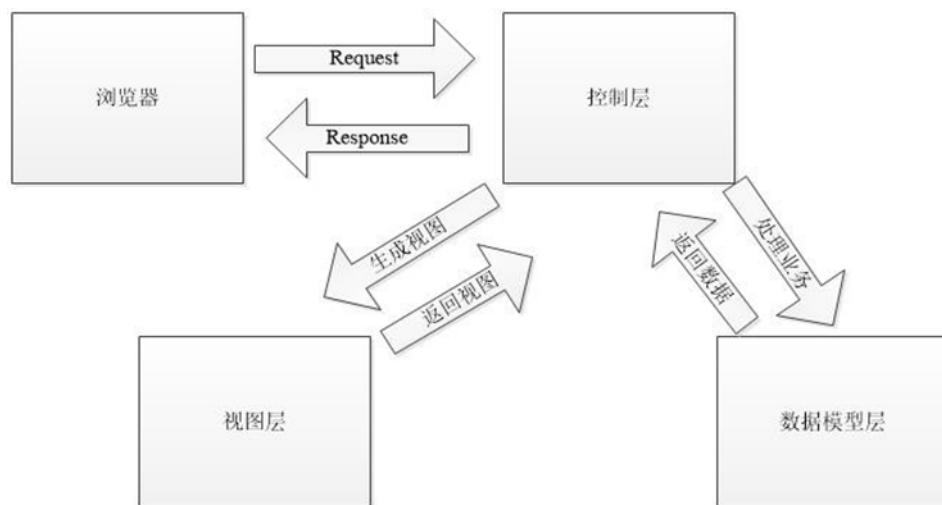


图2.2 SpringMVC 三层架构

2.2.3 SpringBoot 框架的介绍

SpringBoot 本质上是 Spring 框架的扩展，在传统的 Spring 开发中，有大量的 XML 配置文件存在于项目中，使得项目变得臃肿，繁琐的配置导致了开发和部署效率的降低，于是 SpringBoot 应运而生。SpringBoot 是 Spring4.0 的升级版，它继承了 Spring 的所有优点，保留了 IOC 以及 AOP 的功能，也同样支持 SpringMVC 的功能；同时消除了设置 Spring 应用程序所需的 XML 配置，简化了配置和部署过程，只需要在 pom 文件中添加需要的依赖即可，再加上注解的引入，使得开发过程变得更为高效，便捷。

2.2.4 MyBatis 框架的介绍

Mybatis 是一款常用的持久层框架，它支持自定义 SQL 查询语句和映射。MyBatis 免除了几乎所有的 JDBC 代码以及设置参数和获取结果集的工作。MyBatis 可以通过 XML 配置文件来配置和映射原始类型、接口为数据库中的记录。Mybatis 集成了大量实用插件，可以直接生成 Mapper 映射文件。方便开发的同时能更高效的管理数据库。

2.2.5 SSM 框架的介绍

SSM（Spring+SpringMVC+MyBatis）框架集由 Spring、MyBatis 两个框架整合而成（SpringMVC 是 Spring 中的部分内容），也可以将 Spring 换成 SpringBoot。现在主流的 Web 应用也都是基于表现层、业务层、持久层的业务架构方式进行开发的^[15]。如图所示：

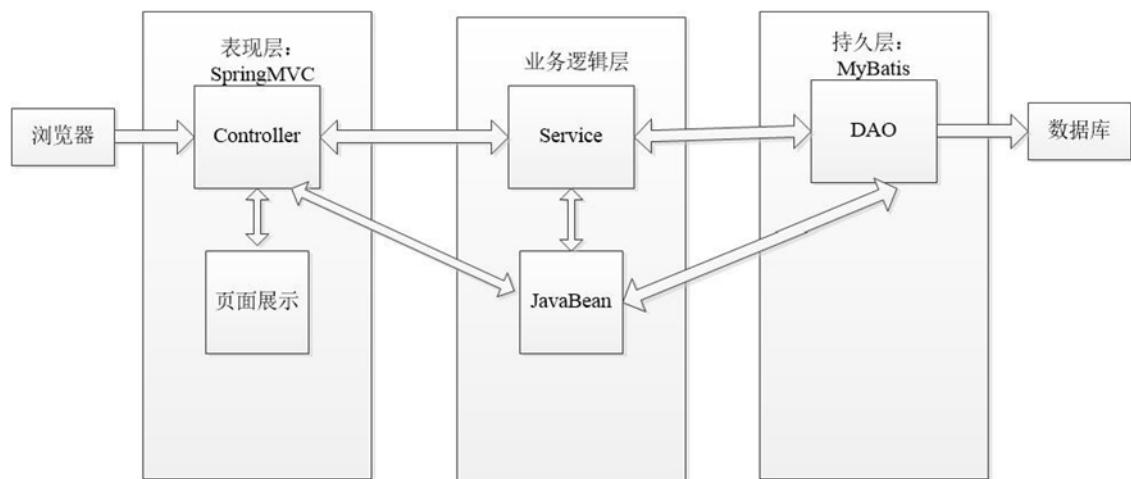


图2.3 SSM 框架

其中 SpringMVC 负责表现层，该层通过前端界面与用户进行交互，接收用户的请求然后传给 Service 层；Service 层又叫业务逻辑层，该层封装了大量方法用于实现业务功能和处理用户请求。由于用户请求多会用到数据库中的信息，所以 Service 层又会连接 DAO 层，DAO 层全称 data access object，也叫数据持久层，在该层中由 MyBatis 框架负责对数据库进行增删改查；而 Spring/SpringBoot 不属于其中的任何一层，它负责整个系统中的对象管理工作。

2.3 数据库技术

2.3.1 MySQL 数据库的介绍

MySQL 是现今使用最普遍的关系型数据库之一。区别于 NoSQL 非关系型的数

据库，MySQL 的数据库采用了表格的储存形式并且支持事务，可以进行回滚并且有较强的扩展能力。MySQL 体积小灵活，而且完全开源的，同时为多种编程语言提供了 API，这些优点使得 MySQL 备受欢迎。在本文中选用 MySQL 数据库存放用户，商品信息和促销活动信息。

2.3.2 Redis 数据库的介绍

Redis (Remote Dictionary Server)，是一个支持网络的，可基于内存的键值对数据库。Redis 有以下优点：

第一：支持存储多种数据结构。可以应对多种使用场景如下表所示：

表2.1 Redis 数据类型与应用场景

数据类型	简单介绍	应用场景
String	底层是使用字节数组实现的，可以存储任何数据，上限为 512m。	经常用来存储序列化的对象，比如序列化的图片等。
Hash	底层是用Hashtable实现的一个String-value的键值对。	适合用于存储序列化后的对象。
List	底层是用双向字符串列表实现的。	消息队列，排行等功能
Set	无序集合，内部键值对是唯一的，底层结构是字典。	社交软件的共同关注，共同喜好等功能
Sorted Set	底层可以使用ziplist或者skipList。	热点搜索等功能

第二：因为 Redis 是基于内存的，所以读写性能极高，读的速度为 110000 次/秒，写的速度是 81000 次/秒。

第三：Redis 的所有操作都是原子性的，即所有的操作都是独立互不干扰的。

第四：与 MySql 不同的是，Redis 的所有数据都是保存在内存中的，相较于 MySQL，Redis 的数据持久化方式更加多样，也更便捷，再加上优秀的读写速度，Redis 数据库经常被用作缓存工具。本文中就有使用 Redis 作为缓存

2.4 Nginx 服务器的介绍

Nginx，原名 engine x，是一个由俄罗斯人 Igor Sysoev 开发轻量级高性能的 web

服务器。它的并发能力强，占有内存小，能够支持响应上万个并发连接请求^[16]。因此 Nginx 备受各大互联网公司和电商平台的青睐。Nginx 高性能的原因有如下几点。

(1) 支持 epoll 多路复用模型。

在 epoll 模型之前，经常使用的是 bio 阻塞式的 IO 模型，在这种模型中一个请求占用一个线程，服务端可客户端通过 Socket 进行通信。

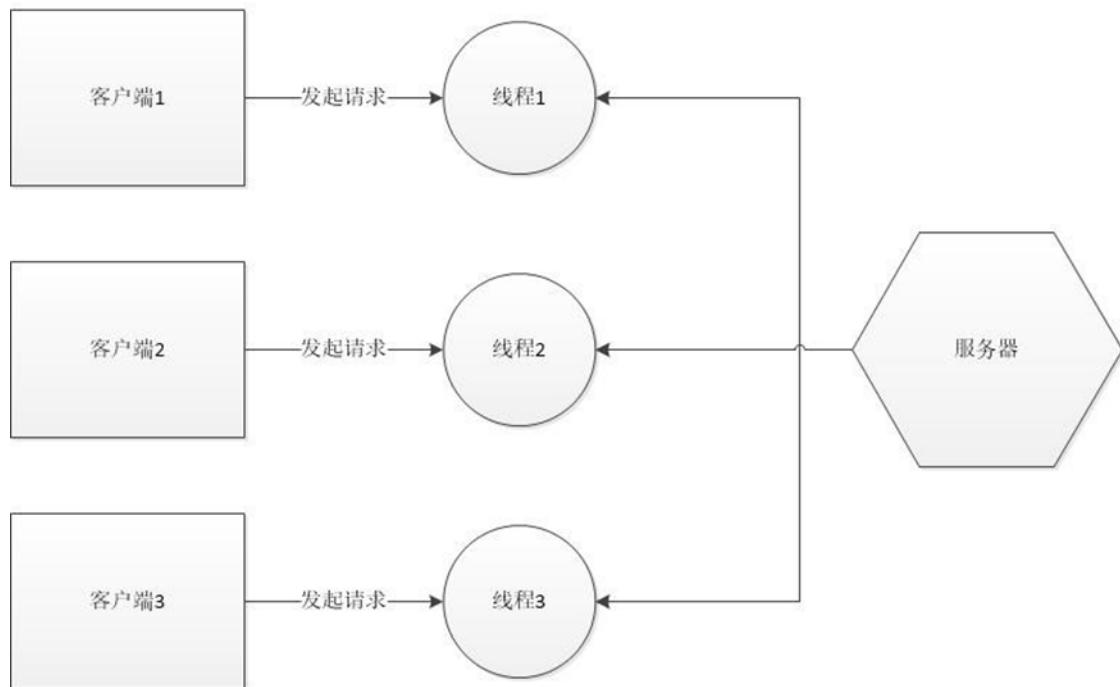


图2.4 bio 模型

bio 模型最大的缺点是在客户端跟服务器端通讯时，一个请求如果因为网络延迟没有及时获取到数据，那么这个线程就会一直等待，从而造成 IO 阻塞，而且一个线程对应一个客户端带来的网络开销非常大。这些缺点在高并发的应用场景会显得尤为致命。而 epoll 是一种多路复用的 IO 模型，该模型摒弃了一个请求对应一个线程的方式，只用一个线程通过记录 IO 流来对所有的请求进行统一管理，只需要关注其中有变化的活跃连接然后告知服务器，服务器设置对应的回调函数并执行。这样的处理机制极大程度上提升了信息处理的效率，也有效的避免了 IO 阻塞，如图 2.5 所示：

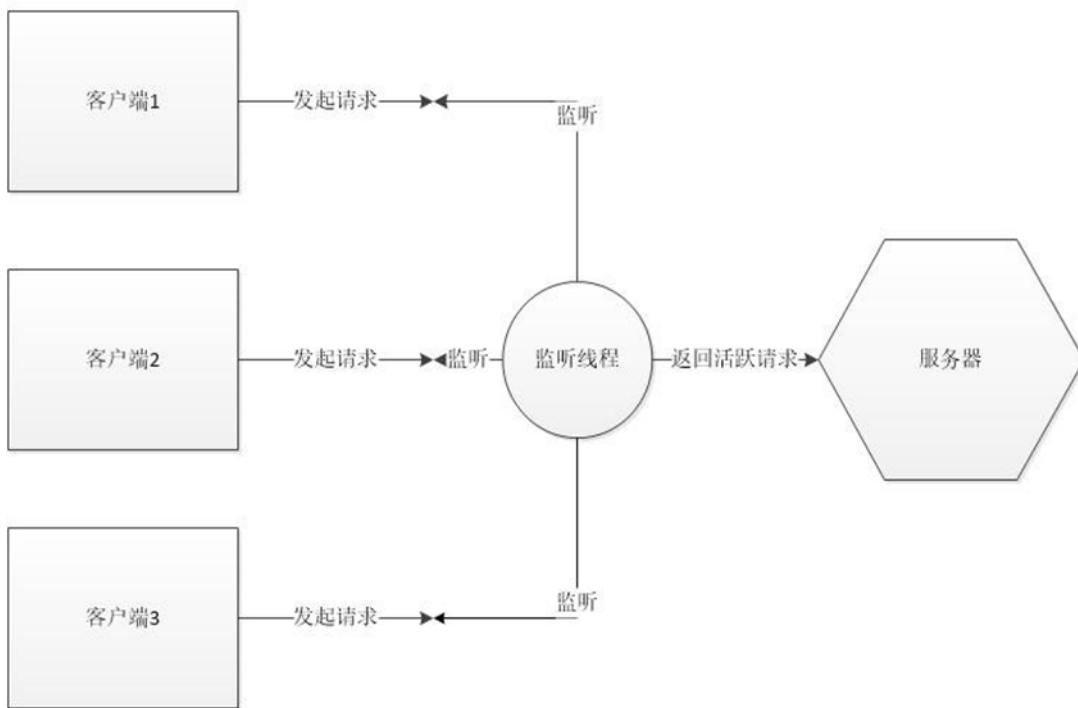


图2.5 epoll 模型

(2) master-worker 进程模型。

当启动 Nginx 时，同时会启动一个 master 进程，和若干 worker 进程，启用的 worker 进程的数量可以在 nginx.conf 文件中进行配置，如图 2.6 所示：

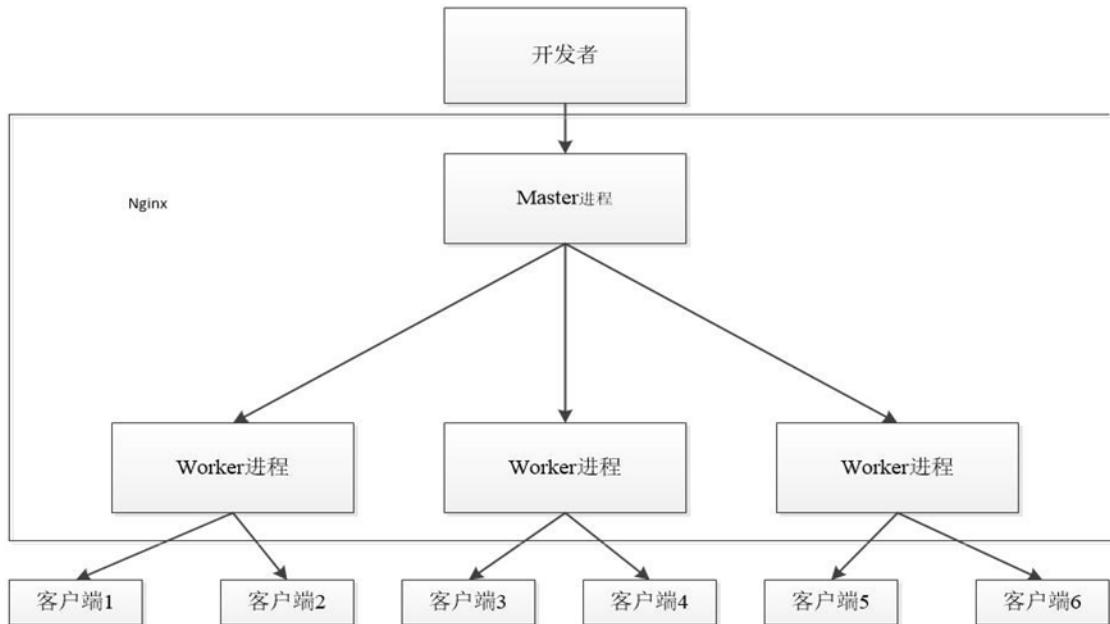


图2.6 master-worker 进程模型

由 Worker 和客户端进行基于 epoll 模型的连接并处理请求，然后通过 Master 进

程来管理 Worker 进程，并且能够获取 Worker 中的全部数据。如果某个 Worker 进程出现异常或者宕机，Master 会马上接管该 Worker 进程的权限并新建一个 Worker 进程并将故障 Worker 进程的数据复制给新建进程，这样的故障切换速度非常快。依赖于这种机制，即便 Nginx 重新启动或者某个 Worker 进程故障，也不会出现客户端连接失败或者需要重新发起请求的情况，也就是可以实现“平滑重启”。

（3）协程机制。

协程是一种依附于线程的内存模型，一个线程可以包含一个或多个协程。区别于线程，协程是异步的，而且切换开销较小。任何一个协程出现时，Nginx 会剥夺该协程的执行权，调用另一个空闲的协程来处理之前的请求这样的特点也造就了协程的高效。

鉴于以上优秀的特性，Nginx 它的用途也多种多样，主要有：用作 Web 服务器；用作反向代理服务器和动静分离服务器。本文使用了 Nginx 的动静分离和反向代理的功能。

2.5 RocketMQ 消息中间件的介绍

MQ（Message Queue）也叫消息队列，是消息通讯中储存消息的容器，也叫消息中间件。消息的发布者将消息添加到消息队列中，消费者取走消息后根据具体要求执行。

消息队列可以确保信息传递的可靠性，并且实现业务解耦。主流的 MQ 有 RocketMQ、kafka、RabbitMQ 等，本文选择使用 RocketMQ 来处理异步扣减库存的事务。

RocketMQ 是阿里巴巴旗下的一款开源 MQ 产品，相较于其他 MQ，RocketMQ 有以下优势：

- (1) 支持集群模型，负载均衡和水平扩展功能。
- (2) 有上亿级别的消息堆积能力。
- (3) 有消息失败重试机制。
- (4) 提供丰富的 api。

2.6 本章小结

本章介绍了本文使用的开发技术，包括开发框架，数据库技术，反向代理服务器以及消息中间件，并且详细说明了这些技术的优势以及在本文中的使用场景。

第三章 秒杀商城基础项目的构建

本章将实现了基于 SpringBoot 框架的电商项目搭建和基础功能的实现，采用前后端分离的方法和 SpringMVC 架构，完成了从数据库与数据模型的设计到注册、登录以及商品创建下单、促销等基础功能的实现。同时还设计了自定义的通用返回类型和参数校验的模块，增强了项目的可用性，也优化了用户的使用体验。

3.1 数据库，领域模型以及通用返回类型的设计

该部分是在实现电商基础功能前要做的准备工作。首先是数据库的设计，采用了分库分表的设计思路；然后是基于数据库的数据模型的设计，该模型的设计遵循了领域模型的设计思路；最后设计了通用的返回类型用于向前端返回请求的数据或者错误信息。

3.1.1 数据库的设计

首先进行商品模型的设计，商品应该具有的属性有商品编号、商品名称、价格、秒杀、价格以及商品图片，根据这些基础需求设计商品详情表如表 3.1 所示：

表3.1商品详情表

字段名	数据类型	是否为空	备注
id	int	否	自增主键
title	varchar	否	商品名称
price	decimal	否	商品价格
description	varchar	否	商品描述文案
sales	int	否	商品销量
img_url	varchar	否	商品图片链接

商品编号，即 id 作为唯一标识商品表中的某一条记录的字段，设置为自增主键，表中所有的字段都设置为非空；第二张表是商品的库存表，如表 3.2 所示。之所以将商品相关的表格分为两张是采用了分库分表的设计思想。随着单表中的数据量越来越大，数据库的查询响应时间越来越长，相应的，数据库的读写所需的时间也越来越多，分库分表即把单个的数据库或者表格分为分散开来用以提升性能，能够降低应用与数据的耦合性，做到应用与数据的无关性^[17]。同时该项目对商品模型的操作主要分为两

种，一种是查询商品详情，第二种是下单时完成时减库存，两种操作互相独立，所以将两张表做分表处理是更为合理的。库存表与详情表通过 item_id 这一外键相关联。

表3.2 商品库存表

字段名	数据类型	是否为空	备注
id	int	否	自增的主键
stock	int	否	库存数
item_id	int	否	库存对应的商品id

接下来是用户相关的表，首先是用户信息表 3.3：

表3.3 用户信息表

字段名	数据类型	是否为空	备注
id	int	否	是该表的主键，自增
name	varchar	否	用户名，长度64位
gender	enum	否	枚举类型，男与女
age	varchar	否	用户年龄
telephone	int	否	用户登陆手机号码，设置唯一约束

包含用户 id、用户名、性别、年龄、电话号码等字段，同样的，用户 id 被设置为自增主键，且每个字段均为非空。同时为电话号码添加唯一索引，避免用同一号码注册多个账号的情况出现。用户密码字段则独立成为另一张用户密码表，如表 3.4：

表3.4 用户密码表

字段名	数据类型	是否为空	备注
id	int	否	自增主键
encrpt_password	varchar	否	加密后的用户密码，以密文的方式存储在表中
user_id	int	否	作为外键关联到用户表

出于安全性的考虑，在数据库的设计层面，密码跟用户主表是必须分开存取的，所以将密码字段独立创表。id 作为自增的主键，user_id 作为外键和用户信息表中的 id 字段相关联，encrpt_password 存放的是经过加密后的非明文密码，最大程度上保护

用户的私密信息，提升系统的安全性。

除了用户以及商品相关的表，还需要专门的数据库表来记录订单信息，如表 3.5 的 `order_info` 订单信息表格：

表3.5 订单信息表

字段名	数据类型	是否为空	备注
<code>id</code>	<code>int</code>	否	非自增主键
<code>user_id</code>	<code>int</code>	否	下单用户id
<code>item_id</code>	<code>int</code>	否	下单商品的id
<code>item_price</code>	<code>double</code>	否	下单商品的价格
<code>amount</code>	<code>int</code>	否	下单数量
<code>order_price</code>	<code>double</code>	否	订单的总价

`id` 字段用来标识每一次交易，`user_id` 与 `item_id` 分别关联到交易的用户 id 以及商品 id；`item_price` 表示商品单价，`amount` 是该次交易的购买数量，`order_price` 是该次交易的总金额。

除了以上的模型，还需要一个促销模块用来管理商品在优惠促销期间的相关业务。由此设计促销数据库表格如表 3.6：

表3.6 促销信息表

字段名	数据类型	是否为空	备注
<code>id</code>	<code>int</code>	否	自增主键
<code>promo_name</code>	<code>varchar</code>	否	促销活动名称
<code>start_date</code>	<code>datetime</code>	否	促销活动开始具体时间，精确到秒
<code>item_id</code>	<code>int</code>	否	商品描述文案
<code>promo_item_price</code>	<code>int</code>	否	商品销量
<code>end_date</code>	<code>datetime</code>	否	促销活动结束具体时间，精确到秒

类似的，`id` 作为主键用来标识促销活动的标号，类似的，`id` 作为主键用来标识促销活动的标号，`promo_name` 标识该次促销活动的名称，`start_date` 是活动开始时间，格式是一个 14 位的字符串包括年月日时分秒，`item_id` 是促销的商品 id，`promo_item_price` 则是促销价格。

自此数据库的设计暂告一段落，在工程中导入 SpringBoot 对 MyBatis 的依赖，使用 druid 连接池。编写 Mybatis-generator 工具来将数据库中的表格映射到工程中，生成对应的实体类 DataObject、映射文件 Mapper 以及数据访问接口 DAO。

3.1.2 领域模型的设计

领域模型（Domain Model）是对业务角色和业务实体之间应该如何联系和协作以执行业务的一种抽象。业务对象模型从业务角色内部的观点定义了业务用例。该模型为产生预期效果确定了业务人员以及他们处理和使用的对象之间应该具有的静态和动态关系。它注重业务中承担的角色及其当前职责。这些模型类的对象组合在一起可以执行所有的业务用例^[18]。

基于领域模型的设计思想，该系统中一共设计了三种模型，Model 对象模型、DO（DataObject）数据对象模型、VO（ViewObject）可视化模型。Model 对象模型是一个概念模型，它包含一个对象应该有的全部属性，在后续的设计中所有的模型转化也都是基于 Model；DataObject 模型是由数据库直接映射过来的模型，它其中的所有属性都是跟数据库中的字段一一映射的，因为在设计数据库的阶段部分数据库表采取来了分库分表的设计思路，所以这些表格的 DataObject 需要嵌套组成 Model。例如，需要将 UserDO 用户信息数据模型与 UserPasswordDO 拼接起来才能得到一个完整的 UserModel，类似的还有 ItemDO 与 ItemStockDO 拼接才能得到 ItemModel；VO 可视化模型在该系统中使用了两种，UserVO 与 ItemVO，UserVO 是用户可视模型，不包括 UserModel 模型中的用户密码信息；ItemVO 是面向用户的商品模型，其中包含了 ItemModel 商品模型，以及 PromoModel 中的促销信息。具体各模型关系如下图所示：

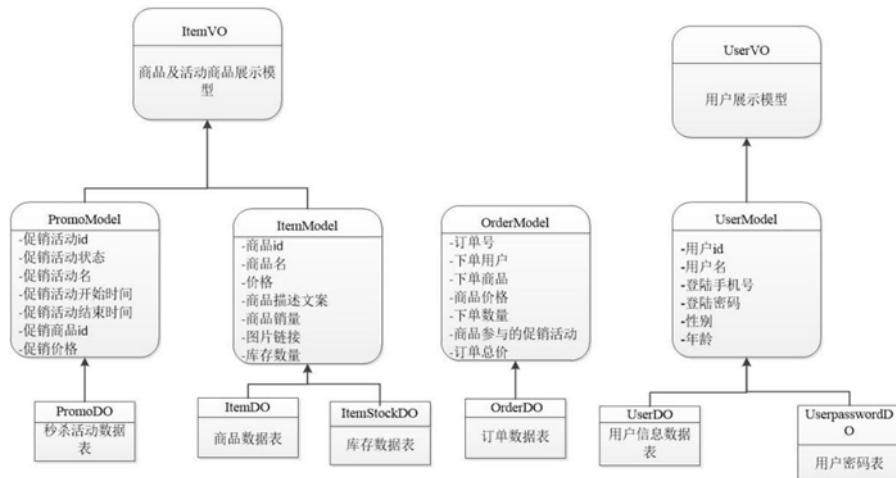


图3.1 商品模型关系图

3.1.3 通用返回对象的设计

作为一个前后端分离的项目，后端的控制层需要响应前端的请求并做出反馈，反馈包括两种类型的信息，第一种是请求受理成功，后端返回前端需要的信息；第二种是请求受理失败，后端返回前端一个错误信息。而在通常情况下，服务器会采用的是错误信息的反馈方式是一个状态码加上服务器自带的错误页面，对于前端来说这样的反馈信息十分笼统难以解析，没有反映出具体的错误信息，无法对异常作出处理，也无法很好的告知用户。所以设计一种通用的返回对象用于构建自定义的反馈信息告知前端是非常有必要的。

新建一个 `response` 包用来处理反馈信息的业务，其中定义一个 `CommonReturnType` 的类作为通用返回对象，如图：

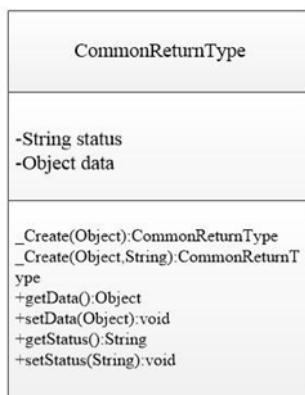


图3.2 通用返回类图

该对象包括一个状态 `status` 和返回数据 `data`，和一个 `create` 方法。前端通过 `status` 属性来表示后端是否有正确受理前端发起的请求，后端控制层通过 `create` 重载方法构建通用返回类型，代码如下：

```

public static CommonReturnType create(Object result)
{
    return CommonReturnType.create(result, "success");
}

public static CommonReturnType create(Object result, String status)
{
    CommonReturnType type = new CommonReturnType();
    type.setStatus(status);
    type.setData(result);
}
  
```

```

    return type;
}

```

status 属性有两种，第一种“success”表示后端服务器处理的状态是正常的，此时 data 为返回前端的 json 数据。如果不需要返回数据则调用第一个 create 方法，传入 null 即可；如果需要返回数据，则调用第二个 create 方法传入该数据和状态。

Status 属性的第二种值“fail”则表示服务器处理状态是错误的，此时会返回一个自定义的通用错误类型，同时抛出一个自定义的异常类。

考虑到方便后续进行可能的功能与业务扩展，首先定义一个 CommonError 的接口如图 3.3 用于构造通用的错误类型，其中属性有错误码 errCode 与错误信息 errMsg 和对应的构造器。

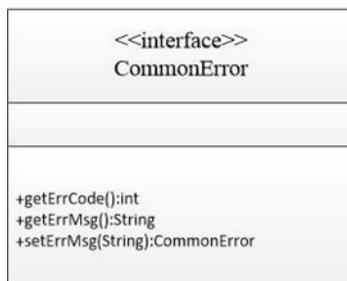


图3.3CommonError 类图

定义枚举类 EmBusinessError 来表示所有业务中可能出现的错误码及对应的信息。具体枚举类型如表：

表3.7枚举错误信息表

错误类型	错误码	错误信息
PARAMETER_VALIDATION_ERROR	10001	参数不合法
UNKNOWN_ERROR	10002	未知错误
USER_NOT_EXIST	20001	用户不存在
USER_LOGIN_FAIL	20002	用户手机号或密码不正确
USER_NOT_LOGIN	20003	用户还未登陆
STOCK_NOT_ENOUGH	30001	库存不足

其中 1 开头的是通用错误类型；2 开头为用户信息相关错误定义；3 开头为交易信息错误定义。该枚举类实现了 CommonError 接口，并定义了对应的构造器方法。

完成以上设计后定义一个业务异常类 BussinessException，该异常类继承了 Exception 类，并实现了 CommonError 接口，通过 CommonError 来构造业务异常类。

最后设计一个基类控制器 BaseController 接入控制层用来捕获控制层的所有异常并构造 CommonReturnType 返回。具体流程如图所示：



图3.4 返回错误信息以及异常处理流程图

所有控制层出现异常后均被 BaseController 捕获，将该异常通过 CommonError 接口调用 EmBusinessError 的错误枚举类型包装成 BussinessException 异常类抛出；然后调用 create 方法构建 CommonReturnType 类型返回错误信息。

完成该设计后在用户信息数据库表中填入一些用户信息用于测试，使用映射路径 /user/get?id 来获取对应 id 的用户信息，分别输入存在的和不存在的 id 值，返回前端的结果如下：

```
{
  status: "fail",
  - data: {
      errorCode: 20001,
      errMsg: "用户不存在"
    }
}
```

图3.5 用户不存在返回信息

```
{
  status: "success",
  - data: {
      id: 10,
      name: "测试用户",
      gender: 1,
      age: 25,
      telephone: "123456"
    }
}
```

图3.6 成功返回用户信息

有了以上的机制后就可以自定义抛出的 Exception 类型和返回前端的信息，如果后续业务扩展需要新的错误类型，可以直接在 EmBusinessError 的枚举类中再新增错误信息，易于扩展。

3.2 商城基础功能的实现

该部分是商城基础功能的实现，包括用户注册、登陆、下单和商品以及促销活动的创建。所有功能的实现均遵循前后端分离的思路，先设计后端功能，再完成前端页面。后端代码中每一项功能都有对应的 Controller 代码块、Service 接口以及 Service 接口的实现类 ServiceImpl，DAO 层则由 Mybatis-generator 工具根据数据库表格自动生成，无手动编写，并且在 Controller 层接入了上一部分中设计的通用返回类型。除此之外还对参数校验功能进行了优化。

用户在注册界面填入注册信息，包括手机号，用户名，年龄等信息以后提交便可以完成注册。

后端用户注册功能的实现：

Service 层：在 Service 层定义一个 UserService 接口来处理用户注册的请求，其中定义一个 register 注册方法，该方法接收一个用户模型 UserModel。在该接口的实现类 UserServiceImpl 中首先使用@Transactional 注解将该注册方法声明为一个事务。register 方法中先判断传入的 UserModel 是否为空，如果为空则抛出一个 BusinessException(EmBusinessError.PARAMATER_VALIDATION_ERROR) 表示参数不正确异常。然后分别将 passwordDO 与 userDO 从传入 UserModel 转换出来，调用 userDOMapper 映射文件的 insertSelective 方法将两个转化而来的数据模型分别插入对应的数据库中，在 userDO 插入时如果抛出 DuplicateKeyException 异常，表示注册手机号重复，抛出一个 BusinessException(EmBusiness.PARAMETER_VALIDATION_ERROR，“手机号已经注册”)。

Controller 层：在 Controller 层的 UserController 中定义一个方法 register 来实现注册流程，映射路径为“/register”；请求方式是 post；指定处理请求的提交内容类型为 “application/x-www-form-urlencoded”。使用 @RequestParam 获取将用户在注册界面填入的信息，包括手机号码、用户名、性别、年龄等，返回值为通用返回类型 CommonReturnType。在该方法中创建一个 UserModel 对象，通过 UserModel 的 set 方法将注册信息填入进去，其中密码的加密使用到 MD5 加盐算法，代码如下：

```
String generateSalt(String password)
{
    Random r = new Random();
    String salt = r.nextInt(10000000000L) + "";
```

```

StringBuilder sb = new StringBuilder(16);
sb.append(r.nextInt(99999999)).append(r.nextInt(99999999));
int len = sb.length();
if (len<16){
    for (int i = 0;i<16-len;i++){ sb.append("0");
    }
}
String salt = sb.toString();
//将盐值与经过 MD5 加密过后的密码拼接起来
password = this.EncodeByMd5(password)+salt;
}

```

然后调用 Service 层 UserServiceImpl 实现类的 register 方法将填入注册信息的用户模型传入。

前端注册页面的实现：首先引入 JQuery 以及需要用到的页面样式的 js 文件。使用 label 和 input 标签接收用户的所有注册信息，然后创建一个注册按钮，使用 onclick 绑定该注册事件。

将注册时需要填入的信息放到 JavaScript 中，同时接受这些内容的同时要进行校验，例如所有信息都不能为空、年龄范围需在 8-100 岁之间等。使用 Ajax 请求，请求路径为“/user/register”，data 则是所有的注册信息。成功注册的回调函数会提示“注册成功”；失败则会显示“注册失败”及其原因。注册功能的实现效果如图 3.7 所示：



图3.7 注册成功界面

可以看到数据库中插入了用户信息与密码如图 3.8 与 3.9 所示，其中密码是密文形式储存的。至此完成了注册功能的实现。

	id	name	gender	age	telephone
▶	10	测试用户	男	25	18912345678

图3.8 注册成功的用户信息表

id	encrpt_password	user_id
10	uG0ENdIZwRJua25KN	10

图3.9 注册成功的用户密码表

3.2.2 登陆功能的实现

该系统使用手机号与密码进行登录。用户登录的后端代码的设计：

Service 层：在 UserService 的接口中定义一个 validateLogin 方法用来对用户登录时输入的手机号码与密码进行校验，然后返回该登录用户的 UserModel。在其实现类 UserServiceImpl 中实现该功能：

(1) 首先在 UserDOMapper 的映射文件中定义一个 selectByTelephone 方法，通过用户的手机获取 UserDO；代码如下：

```
<select id="selectByTelephone"
    resultMap="BaseResultMap">select
    <include
        refid="Base_Column_List"/>
    from user_info
    where telephone = #{telephone,jdbcType=VARCHAR}
</select>
```

对 UserDO 进行判空处理，如果获取的模型为空则抛出一个登录失败的异常。然后通过该 UserDO 获取对应的 UserPasswordDO 中的密码，使用 StringUtils.equals 方法与用户输入的密码进行对比，如果不同则抛出登陆失败的异常。将获取到的 UserDO 与 UserPasswordDO 组合为完整的 UserModel 并返回。

Controller 层：定义一个方法 login 来处理登陆流程，接收前端用户输入的手机号码与密码，同样返回 CommonReturnType，映射路径为"/login"；请求方式是 post；指定处理请求的提交内容类型为"application/x-www-form-urlencoded"。在该方法中首先对传入的参数进行判空处理，接着调用 Service 层 UserServiceImpl 实现类的 validateLogin 方法，接收返回的 UserModel，该模型即为登陆进来的用户模型。使用 HttpServletRequest.getSession().setAttribute() 方法将一个登陆凭证和登陆的用户模型放到 session 中，后续下单操作时就通过该凭证来判断用户是否登录。以上完成了用户登录的后端代码的设计。

(2) 用户登录前端界面 login 的实现：参照一般类似系统的登录界面一般有两个按钮，登录与注册，点击注册的按钮的话会跳转到 register 注册界面。登录功能的前端代码实现类似于注册，同样使用 Ajax 请求，只需接收用户电话号码及密码；回调

函数同样有两个，成功的回调函数会提示登录成功。效果如图 3.10 所示：



图3.10用户登陆成功界面

3.2.3 校验功能的实现

在上文的注册以及登录功能的实现中有多处需要参数校验，而之前的代码大多都只做了判空处理，在几乎所有程序的开发中，当涉及到程序的业务逻辑时，请求参数的数据校验是必要的。当请求参数格式不正确的时候，需要程序监测到，并且返回对应的错误提示，以此来达到数据校验的目的。对于前后端分离开发过程中，数据校验还需要返回对应的状态码和错误提示信息。状态码和错误信息的设计上文已经做出了说明，接下来介绍校验方法的设计。

本系统使用 Hibernate-Validator 对数据进行校验，它是 Hibernate 项目中的一个数据校验框架。Hibernate 是一个开放源代码的对象关系映射框架，通常用来处理持久层业务。Validator 提供了常用来验证实体类是否满足需求，同时还提供了注解的方式对类中的属性进行校验。

首先要导入 Hibernate-validator 的依赖，本系统使用的是 5.2.4 版本。然后为了使代码各部分耦合度更低，对校验结果做一个封装。在工程目录中新建一个 validator 的包来存放这个封装，在其中定义一个存放校验结果的类 ValidationResult。其中包含两个属性：布尔类型的 hasError 来表示校验结果是否有错以及存放错误信息的 errorMsg，其底层实现使用 HashMap；一个 getErrorMsg 方法来通过字符串来返回全部错误信息。

首先通过工厂的初始化方式将 Hibernate-Validator 实例化，得到了一个校验器的实例 validator。接着定义校验方法 validate，返回值即为 ValidationResult，传入一个 Object 类，即为需要校验的实体类。在该方法中调用 validate 方法，新建一个集合 set 来存放该方法返回的结果。逻辑校验部分只需要判断该 set 是否为空，不为空则代表校验的对象有参数错误，将校验 ValidationResult 的 HasErrors 设置为 true，然后将 set 中的错误内容遍历存放到 errorMsg。将 validator 校验器接入到 Service 层中，将注册方法 register 中传入的 UserModel 用 validate 方法进行校验，如果返回的校验结果中

的 hasErrors 属性为 true，则抛出一个通用错误类型。

接着使用 Hibernate-validator 提供的注解对实体类的属性进行校验，如表 3.8

表3.8 用户模型校验表

注解名	添加注解的属性	注释
@NotNull(message = "用户名 不能为空")	name	用户名不能为空
@NotNull(message = "性别 不能不填写")	gender	性别必须填写
@NotNull(message = "年龄 不能不填写") @Min(value = 0, message = " 年龄必须大于 0 岁") @Max(value = 150, message = " 年龄必须小于 150 岁")	age	年龄必须填写，且不能大于 150 或是小于 0
@NotNull(message = "手机号 不能为空")	telephone	手机号不能为空
@NotNull(message = "密码 不能为空")	password	密码不能为空

以注册年龄为例，当填入不符合要求的年龄时，效果如图：



图3.11 年龄校验效果

3.2.4 商品模块功能的实现

商品模块主要实现三个功能：创建商品功能、商品列表浏览功能以及商品详情页显示功能。在商品创建好后，用户可以浏览全部商品列表，然后点击商品进入商品详情页，其中显示有商品的各项信息。

创建商品：通过前端创建商品页面来新建商品，后端接收到商品信息填入数据库中来完成新商品的入库。

Service 层：在 Service 层中定义一个 `createItem` 方法，接收一个商品模型 `ItemModel` 的参数，返回值也是 `ItemModel`。在其实现类 `ItemServiceImpl` 的 `createItem` 方法中首先对传入的 `ItemModel` 参数使用 `validator` 校验：

表3.9 创建商品校验表

注解名	添加注解的属性	注释
<code>@NotBlank(message = "商品名称不能为空")</code>	<code>title</code>	商品名不能为空
<code>@NotNull(message = "商品价格不能为空") @Min(value = 0,message = "商品价格必须大于 0")</code>	<code>price</code>	商品价格不能不填也不能为 0
<code>@NotNull(message = "库存不能不填")</code>	<code>stock</code>	库存不能不填
<code>@NotBlank(message = "商品描述信息不能为空")</code>	<code>description</code>	必须填入商品描述
<code>@NotBlank(message = "商品图片信息不能为空")</code>	<code>imgUrl</code>	图片链接不能为空

完成校验后将传入的 `ItemModel` 转化为一个商品数据对象 `ItemDO` 与库存数据对象 `StockDO`。然后将分别将两个数据对象插入数据库即可完成创建商品并入库的功能。再在 `ItemServiceImpl` 中定义一个 `getItemById` 方法，通过传入商品 id 来返回 `ItemModel`，由于需要根据 id 来获取库存信息，所以需要在 `ItemStockDOMapper` 文件中自定义一个方法，代码如下：

```
<select id="selectByItemId"
parameterType="java.lang.Integer" resultMap="BaseResultMap">
    select
        <include refid="Base_Column_List" />
```

```

from item_stock
where item_id = #{itemId,jdbcType=INTEGER}
</select>

```

然后在 getItemById 方法中调用自定义的 selectByItemId 即可获取该商品的 Data Object，然后将其转化为对应的商品模型 ItemModel 并返回。

Controller 层：在 ItemController 控制器的 createItem 方法中接收前端传入的商品名称、商品描述、商品价格以及商品库存等信息，映射路径为“/create”；请求方式是 post；指定处理请求的提交内容类型为“application/x-www-form-urlencoded”。然后使用 ItemModel 的构造器方法将以上接收到的信息 set 进去，同样是基于领域模型的设计思路，将 ItemModel 转化为 ItemVO 返回给前端。

创建商品模块的前端页面的设计与登录界面类似，填入创建商品的各信息时进行前端校验，然后点击创建按钮便可以完成，效果如图：



图3.12 商品创建成功界面

商品列表功能的实现：

Service 层：先在 ItemDOMapper 文件中自定义 listItem 查询方法，不需要传入参数，直接查询全部商品，并按照销量降序排列，返回一个 List。在 ItemService 接口中定义一个 listItem 方法用于获取所有商品，在其实现类 ItemServiceImpl 中的 listItem 方法中调用查询方法获取所有商品并返回。

Controller 层：listItem 方法映射路径为“/list”，请求方式是 get。在方法体中调用 ItemServiceImpl 中的 listItem 方法获取所有商品并转化为 ViewObject 然后返回。

商品列表的前端界面：

不同于之前的界面，商品列表应设计为一个响应式的表格。响应式表格是响应式设计中常用的一种，随着智能手机以及平板电脑等一系列智能设备的流行，为了使 Web 界面能够适应各种尺寸的显示屏幕，响应式设计应运而生。使用了该种技术的界面可以根据显示器屏幕的大小自动调整显示效果，当显示屏小于 798px 时，会出现滚动条。使用 Bootstrap 框架提供的 table-responsive 容器具有响应式效果，只需将 class

的类型设置为 table-responsive 即可使用。使用 Ajax 请求，如果请求成功，使用一个空数组来存放所有商品，在另一个函数 Reload 中遍历该数组，获取每个 ItemVO 然后将里面的商品信息拼接起来，同时要对图片信息的规格做限制，宽度设为 100px。商品列表实现效果如图 3.13：

商品列表浏览				
商品名	商品图片	商品描述	商品价格	商品库存
Iphone XR		第十代iPhone	4299	50

图3.13 商品列表界面

商品详情页的实现：基于创建商品的界面来完成商品详情页的制作。先将列表页增加跳转，在详情界面显示全部信息。在详情页定义一个 getParam 方法，在其中通过 location.search 解析 url 并获得 itemId。然后使用 get 请求，将获取的 id 传入。获取商品信息的方法与商品列表类似。

3.2.5 交易下单功能的实现

首先从订单功能入手，

Service 层：在 Service 层定义一个 OrderService 的接口，里面包含一个 createOrder 创建订单方法来处理创建订单的事务，需传入 userId,itemId，以及下单数量 amount，返回一个订单模型 OrderModel。创建订单的行为同样也是一个事务，要使用@Transactional 标签。在 OrderService 的实现类 OrderServiceImpl 中的 create 方法中需要实现以下功能：校验，减库存。

校验功能：首先调用 ItemService 中的 getItemById 方法通过 itemId 获取到下单商品的模型 ItemModel，如果该模型为空则直接抛出自定义的异常类，显示“商品不存在”；同样的，调用 UserService 中的 getUserId 方法对用户进行校验，如果不存在该 userId 对应的用户，则抛出异常类显示“用户信息不存在”；对下单数量校验则是判断传入的 amount 是否小于等于 0 或者大于 99，异常信息显示“下单数量不正确”。

减库存操作：减库存方法一般有两种，下单减库存与付款减库存。下单减库存即用户一旦下单商品就会减少库存，而付款减库存则是在下单并且付款结束以后库存才会减少。在对商品库存不敏感的场景中，选择两种方法都可以，然而在秒杀活动中如果使用付款减库存的策略，则很容易出现“超卖”现象：即因为没有及时减库存而导致用户总下单数量大于总库存量，这种现象的出现对正常业务的干扰极大，所以本系统采用下单减库存的策略。

首先在 ItemService 层定义一个减库存的方法 decreaseStock，传入下单商品的 it

emId 以及下单数量 amount，返回一个布尔值来表示减库存行为是否成功。在 ItemStockDOMapper 文件中自定义一个减库存的方法代码如下：

```
<update id="decreaseStock"> update item_stock
set stock = stock - #{amount}
where item_id = #{itemId} and stock >= #{amount}
</update>
```

当库存量 stock 大于下单数量 amount 时，更新 stock 为 stock 减去 amount 的值。

在 Service 层中调用该方法返回一个整型值 affectRow 表示该次操作受到影响的条目数，当 affectRow 大于 0 时则代表减库存成功，返回 true；反之则返回 false 并抛出异常显示库存不足。新建一个 OrderModel，将之前获取的订单信息 set 进去，再将该 OrderModel 转化出一个 DataObject，调用对应 orderDOMapper 文件中的 insertSelective 和 increaseSales 插入方法将数据对象入库，返回订单模型。

Controller 层：在 OrderController 中的 CreateOrder 方法中先校验用户的登录状态是否合法，使用到之前在用户登录中保存在 session 中的登录凭证，如果登录状态是 false 则抛出通用异常类型，使用用户未登录的错误码提示用户还未登录；如果为 true 则调用 createOrder 方法，接收 itemId、amount 以及 session 中的登录的用户模型中的 userId，调用 Service 层的方法创建 OrderModel 订单模型后返回 CommonReturnType 即可完成订单入库的操作。

前端页面：在前端商品详情页面增加一个下单的按钮，Ajax 请求中响应数据类型为商品 id 以及下单数量，请求成功的回调函数提示下单成功；请求失败的回调函数提示下单失败，并且判断错误码如果是未登录的错误，则自动跳转到用户登录界面。

3.2.6 促销活动功能的实现

促销活动并不需要在代码中创建，而是选择直接在促销表中填入数据：

id	promo_name	start_date	item_id	promo_item_price	end_date
1	年终促销	2022-12-27 17	1	2500	2022-12-31 17:00:00

图3.14 促销活动信息表

Service 层：

(1) 获取促销信息：在 PromoDOMapper 文件中定义一个查询方法 selectByItemId，通过 itemId 来获取 PromoModel 的促销信息，代码如下：

```
<select id="selectByItemId"
parameterType="java.lang.Integer" resultMap="BaseResultMap">
    select
```

```
<include refid="Base_Column_List" /> from item_stock
where item_id = #{itemId,jdbcType=INTEGER}
</select>
```

在 PromoService 中定义一个 getPromoByItemId 方法，通过输入商品 id 来获取促销信息，在 PromoServiceImpl 的 getPromoByItemId 方法中调用 PromoDOMapper 的 selectByItemId 获取 PromoDO 促销数据模型，如果该模型为空则直接返回。然后将 PromoDO 转化为对应的 PromoModel。

(2) 判断当前时间是否处于活动期间：先为 PromoModel 促销模型添加一个 Integer 类型的 status 属性来表示三种活动状态，1 表示还未开始，2 表示活动正在进行中，3 则表示活动已经结束，然后为其添加 get 和 set 方法。在 Service 层的实现类中先用 DateTime 类获取当前的时间，通过比较当前时间与促销模型的促销开始时间来为 status 属性赋值。

(3) 调整商品模型：加入了促销模块以后商品模块的后端功能要进行一系列修改。第一：为商品模型添加促销信息，在促销期间商品模型需要添加一些特殊信息，比如促销还有多久开始，促销的优惠价格等。由此在商品模型中也引入促销模型并为其添加 get 和 set 方法，然后在 Item 的 Service 层的 getItemById 中，在获取商品模型时也获取对应的促销信息模型，如果该模型不为空并且其状态 status 不等于 3，则将该模型 set 进商品模型中。同时考虑到促销信息应该是对用户可见的，也将促销相关的属性添加到 ItemVO 的可视模型中。然后在 Controller 也要做相应的修改，如果商品有促销活动则将促销模型 set 进去。

第二：修改下单操作。首先对订单模型作出调整如果用户是以秒杀价格买到商品，则需要为模型添加促销的属性。为 OrderMode 添加一个 Integer 的属性 promoid，若该属性不为空，则表示用户是以秒杀的优惠价格下单该商品的，同样的也需要一个 itemPrice，如果 promoid 不为空则表示秒杀价格，为空则表示原价。对 DataObject 以及 Mapper 也作出相应的修改。

其次是订单的 Service 层，应先接收前端 url 上传过来的秒杀活动 id，然后在下单接口内校验对应 id 是否属于对应商品且活动已经开始。在 Service 层的 createOrder 方法的入口参数中添加一个 promoid 然后用 promoid 判断该商品是否属于活动商品。如果商品正处于促销期间，则放入 OrderModel 中的价格应该是促销模型中的促销价格。判断结束后需要将 promoid 给 set 进 OrderModel。

最后是 Controller，在控制器层的创建订单方法的入口参数添加 promoid，将其 required 属性设为 false，表示如果没有 promoid 的值则认定是以平销的价格买入。

(4) 调整前端页面：有了促销活动相应的也要对商品详情页做调整。在详情页的顶端处添加一个 promoStartContainer 容器，用于显示秒杀开始时间；在常规价格标

签底下添加一个 PriceContainer 的容器，在其中添加一栏显示促销价格并为了更醒目将其样式调整为红色。在 Ajax 请求体中获取 itemVO 的 status，为 1 时使用 jQuery 的 id 选择器改动 promoStart 容器的的显示内容，为其添加上促销开始倒计时，在 Price 容器中添加上促销价格。并且当促销快要开始的时候下单功能是关闭的，将下单按钮 disable 掉；倒计时的实现首先要将活动开始时间以及现在的时间，获取出来然后用正则表达式转化为 JavaScript 能够解析的格式，然后计算差值。当差值小于 0 的时候表示促销正在进行，将 status 设置为 2，同时使能下单按钮。

status 为 2 时则将 promoStart 添加上“秒杀活动进行中”的显示信息，Price 容器中添加上促销价格。添加了促销模块的商品详情页如图：



图3.15 促销商品界面

3.3 本章小结

在本章通过使用 SpringBoot+MyBatis 完成了一个 JavaWeb 系统的搭建，在该系统中实现了包括注册，登录，下单，秒杀等基础功能的闭环交易流程。

首先使用前后端分离的设计思路，前端的 UI 界面使用了 html，CSS，jQuery 与图片完成了用户注册、登录、商品展示、下单交易以及秒杀倒计时一系列前端界面。接着在控制层定义了 ViewObject 对象并结合通用的异常处理以及通用的返回对象返回了 data+status 模型；之后在业务层引入了 MyBatis 以及领域模型的概念完成了用户服务，商品服务，交易服务以及活动服务；在数据库层面使用了事务标签来完成事务的切面，使用了数据库的 DAO 来完成对数据的操作。

通过上述对系统总体控制方案的分析和设计，可以构成如图所示的系统总体结构框图 3.16：

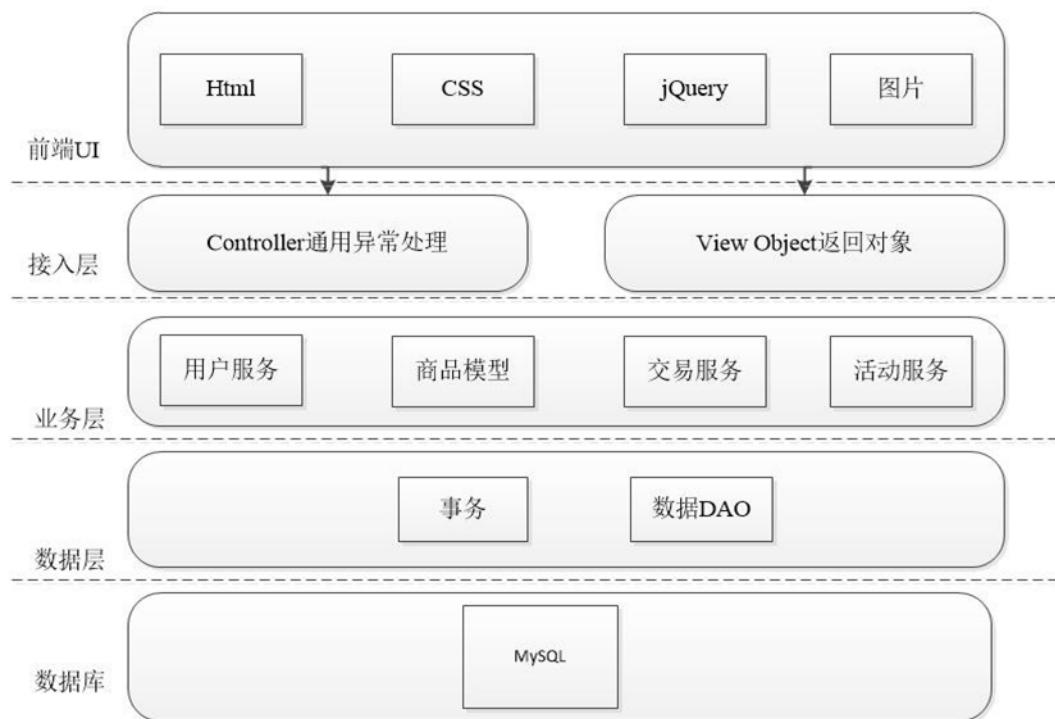


图3.16 系统总体结构框图

第四章 交易与查询性能优化

考虑到秒杀商城的业务逻辑，有两个环节尤为重要：查询与下单。就查询而言，在秒杀活动开始前必然肯定伴随着大量的查询商品页面和刷新的操作，这样的情况对商城的性能来说是个极大的挑战；在秒杀活动开始后，大量集中式的访问也会对数据库造成巨大的压力，甚至致使整个业务宕机。因此对查询性能进行优化十分有必要；而对于交易操作而言，整个过程最核心环节就是库存的扣减的部分，特别是涉及到高并发的使用背景，那就必然要考虑库存控制的问题，如何解决缓存与库存信息一致，不出现超卖少卖的情况也是本章要重点讨论的内容。

4.1 分布式扩展

本文的设计侧重于应对高并发访问的适用场景，现有系统的架构为：商城项目部署在一台服务器上，由前端直接访问该服务器。而服务器的容量有限，在大量访问下很容易达到性能瓶颈，因此需要对系统进行分布式扩展。分布式扩展又分为两种，分布式垂直扩展与分布式水平扩展。垂直扩展主要是提升单机处理请求的能力，例如使用性能更好的服务器，升级速度更快的网卡以及对硬盘扩容等方法，提升硬件性能往往是最容易想到的方法，但是垂直扩展有一个致命的不足，即单机的物理性能总是有极限的，随着用户的不断增多，需要垂直扩展的次数也就越多，这样对业务的发展和扩张极为不便^[19]，所以水平扩展才应对分布式高并发应用场景的正解。该部分内容首先对前端资源进行了动静分离，然后使用 Nginx 对系统进行了分布式水平扩展，并调整了原有的会话管理方式，使之能够适应分布式架构。

4.1.1 Nginx 的动静分离

在本文的系统架构中，Nginx 是用户请求到达的第一个服务器，而用户的请求并不都是动态的，静态请求没必要访问到水平服务器以及数据库中。因此将静态资源部署到 Nginx 上，让 Nginx 同样用作一台动静分离的服务器，如果用户访问的是一些静态的网页，则直接返回 Nginx 上存放的资源；如果用户进行的是 Ajax 的动态请求，则再将请求代理分发到水平服务器上。从而一定程度上提升了响应速度并且减轻底层数据库的访问压力。

为了便于对 Nginx 进行开发，使用 OpenResty 平台，OpenResty 是由章亦春发起的，将 Nginx 核心、LuaJIT 虚拟机、ngx_ua 模块，和第三方 Nginx 模块组合在一起的开源 Web 平台^[20]，将 OpenResty 部署到 Nginx 服务器上。进入 Nginx 的配置文件

nginx.conf该文件包含 5 个代码块，如图 4.1 所示：



图4.1 Nginx 配置文件内容

修改配置文件中的 location 块，如图：

```
server {
    listen      80;
    server_name localhost;

    #charset koi8-r;

    #access_log  logs/host.access.log  main;

    location /resources/ {
        alias  /usr/local/openresty/nginx/html/resources/;■
        index index.html index.htm;
    }
}
```

图4.2 Nginx 前端资源路由

当访问路径命中/resources/时，将访问路径替换为 alias 属性中的路径，然后将所有网页资源部署到 alias 中设置的路径中。

4.1.2 Nginx 反向代理

在应用集群系统中，通常由各种不同硬件性能的服务器组成，如果反向代理时负载均衡算法不合适，服务器集群中的各服务器应对请求时极易出现服务器负载压力的

不均衡,不但使服务器资源得不到充分的利用,还会导致集群系统整体性能的下降^[20]。为了提升整体系统的性能,一般选用 Nginx 作为反向代理服务器,根据实际需求调整均衡策略向后端服务器集群转发客户端的请求以及使后端服务器均匀的完成响应。Nginx 作为代理服务器本身并不处理业务,对用户来说 Nginx 的域名是已知的用户访问 Nginx,然后它将请求反向代理给下游的服务器,这些下游服务器才是真正处理商城业务的部分。这样的做法也保护了后端集群。本文使用了两台下游服务器。

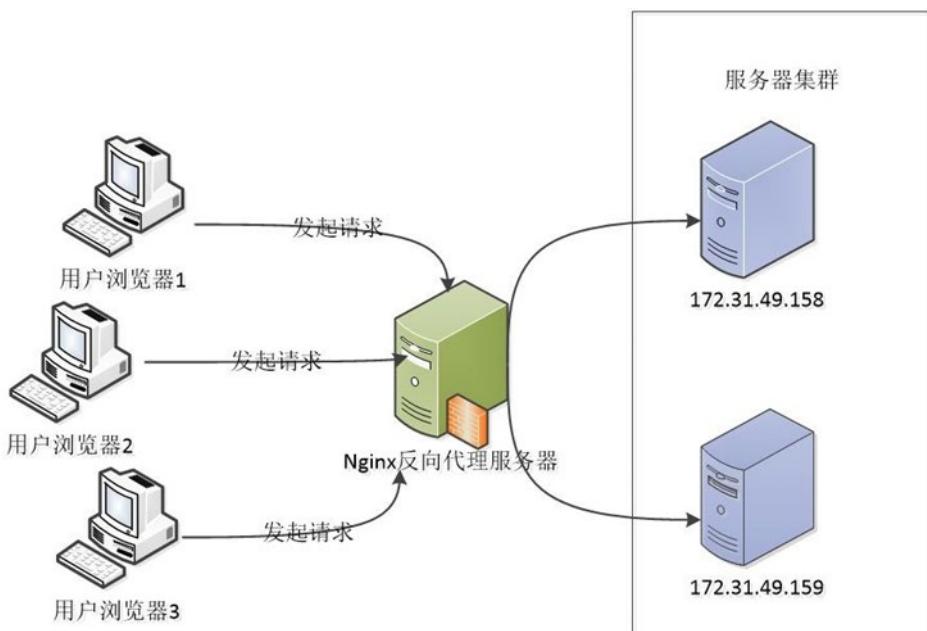


图4.3 Nginx 反向代理

作为均衡负载分配服务器, Nginx 会将客户端的请求通过 `upstream server`, 又称上游服务器, 它可以提供负载均衡功能, 同时支持多种负载均衡算法, 例如轮询、轮询权值等, 通过该功能将请求反向代理到下游的两台服务器上。修改 `nginx.conf` 中的 `server` 块, 新建一个节点命名为 `backend_server`, 里面包含两台下游服务器, 两台服务器的比重均设置为 1, 即采用轮询策略, 意为每个请求按照时间顺序逐一分配到不同的两台下游服务器, 如图 4.4 所示; 然后启用该 `backend_server`, 在 `nginx.conf` 文件中的 `location` 块中使用 `proxy_pass` 设置代理服务器的路径, 除去/resources 路径下的访问全部视为动态请求, 经由 Nginx 反向代理到两台下游服务器上, 如图 4.5 所示:

```

upstream backend_server{
    server 172.31.49.158 weight=1;
    server 172.31.49.159 weight=1;
}

```

图4.4 Server 块的配置

```

location / {
    proxy_pass http://backend_server;
    proxy_set_header Host $http_host:$proxy_port;
    proxy_set_header X-Real-IP $remote_addr;
    proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
}

```

图4.5 Location 块的配置

4.1.3 基于 token 的分布式会话管理

http 请求会有一个 Session 会话管理机制，用来标识用户会话的过程。上一章使用的是 springboot 内嵌的 HttpSession 的实现机制，这个 tomcat 协议实现是基于 cookie 传输 sessionId 的方式来完成容器的实现，其原理如图 4.6 所示：

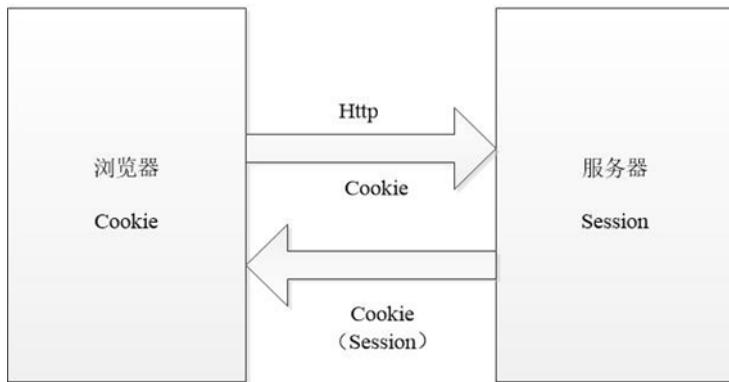


图4.6 基于 cookie 传输 session 原理图

在用户登录的模块中，将登录凭证和 UserModel 加入到 session 中然后在需要用到用户登录状态的方法中，通过登录凭证判断当前用户是否已经登录。使用 cookie 管理会话方法简单，但是它有以下缺点：第一，不适用于分布式架构。在引入了分布式架构之后，虽然每个下游服务器执行的都是同样的业务逻辑，但是 session 内的数据是不共享的，假设用户第一次登陆后访问请求被 Nginx 分发到服务器 1，当下一次访问很可能被分发到服务器 2，此时服务器 2 没法获取登陆的 session 便会判定用户没有登陆。第二，占用内存。每个用户通过认证之后都会将 session 数据保存在服务器的内存中，可能会给服务器带来不小的压力。

考虑到以上缺陷，改用 token 进行会话管理。首先 token 是无状态的机制，在登陆认证成功后会对当前用户数据进行加密生成一个加密字符串 token 并返还给用户，由用户保存，服务器端只保存该加密字符串的密钥，并不存储 token，只负责通过密钥验证用户的 token。

使用 Redis 来存储会话，安装 Redis 数据库，然后在代码中实现 Redis 对 token

的会话管理。

先在 pom 文件中导入 SpringBoot 对 Redis 和 session 管理的依赖并在配置文件中设置 Redis 连接信息。

在工程中新建一个 RedisConfig 类用来管理 Redis 的配置，在该类上添加 @EnableRedisHttpSession 注解，并设置 maxInactiveIntervalInSeconds=1800，Session 失效时间为 1800 秒。经过以上配置后，Session 就会自动去 Redis 中进行存取。

在 UserController 的登陆方法 login 中，在用户登录并验证成功后使用通用唯一识别码 uuid 作为 token 登陆凭证，确保该登陆凭证是全局唯一的。引入一个 RedisTemplate 来建立该 token 与登录用户模型的联系。RedisTemplate 是 Spring 提供给用户的简易 Redis 模板，用户可直接通过 RedisTemplate 进行多种操作，例如设置 key 和 value，删除单个或者多个 key，设置过期时间等。将 uuid 与登陆进来的 UserModel 映射存入 Redis 中。然后为该登陆态设置过期时间一个小时，即登陆过后一个小时需要重新登陆。

接着调整需要验证登陆凭证的部分。进入前端登陆界面，将登陆成功用户的 token 存储在 localStorage 中。localStorage 是 HTML5 中新增的特性，可以在其中存储键值对数据，相较于 cookie，它拥有更大的存储空间，如图 4.7：

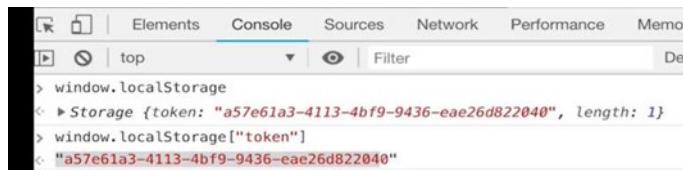


图4.7 localStorage 存储 token

同样需要调整的还有商品详情页的下单界面，如果没有获取到该 token 则报错，不能下单，然后跳转到登录界面；如果有登录凭证，则在 Ajax 的请求体的 url 路径中拼接上该 token。

类似的，还需要在 OrderController 中的创建订单 createOrder 方法中通过 HttpServletRequest.getParameterMap().get() 方法获取 token 并验证，如果 token 为空则表示用户还未登陆，抛出 BusinessException(EmBusinessError.USER_NOT_LOGIN,"用户还未登陆")。

4.2 多级缓存

在现有的秒杀商城的结构中，前端请求经由 Nginx 反向代理到两台轮询的下游服

务器上，然后两台服务器对数据库直接进行访问。这样的设计有一个很大的弊端，当面对秒杀促销的大流量访问时，所有请求均直接到达数据库，可能出现响应时间过长的情况甚至造成系统宕机。在基于计算与存储分离架构的分布式数据库中，本地缓存容量对数据库系统的整体性能均有着极大的影响。为了应对因大流量访问而引起的缓存容量不足的问题，有必要为计算节点和存储节点提供容量大、成本低的多级缓存系统^[21]。为此，本文设计了集中缓存+本地热点数据缓存+Nginx 缓存的多级缓存架构，一方面极大提升了查询能力，提高了该系统的综合性能。另一方面也避免了使用单级缓存时缓存数据过期或者故障带来的隐患。

4.2.1 Redis 缓存商品详情页

使用 Redis 作为缓存的第一级。缓存数据是 Redis 的典型应用之一，使用 Redis 作为缓存中间件有以下优点。第一：丰富的存储数据类型，Redis 支持 list, hash, String, set 与 sortedset 等数据结构的储存；第二点也是最重要的一点，Redis 支持持久化操作且提供了以下持久化方法，一种是快照，另一种是追加文件^[22]。该部分内容实现的是 Redis 的接入以及使用 Redis 完成对商品模型的缓存。由于上文中已经引入了 Redis 需要的依赖，所以直接使用即可。放入 Redis 中的对象必须要实现序列化，将 PromoMode 和 ItemModel 实现 Serializable 接口实现序列化。

Redis 缓存将在 Controller 层中接入，先在 ItemController 中引入 RedisTemplate。然后修改 getItem 方法，根据传入的商品 id，使用 RedisTemplate.opsForValue().get() 方法先去 Redis 中去获取，如果 Redis 中没有，则从下游 Service 层，使用 ItemService.getItemIdById() 方法获取数据，并且使用 RedisTemplate.opsForValue().set() 将数据放入 Redis 中以便于下一次再有相同的请求时直接返回，减少对数据库的依赖。

缓存必须要有失效时间，一方面是为了 Redis 内存考虑，另一方面是为了业务能力做考量。采用被动过期的方式，使用 RedisTemplate.opsForValue() 方法设置过期时间为 15 分钟。

完成设置后发送获取商品的请求，进入 Redis 查看存入的商品模型如图 4.8：

```
127.0.0.1:6379[10]> get "item 1"
"["com.miaoshaproject.service.model.ItemModel", {"id":1,"title":"Iphone XR","price":50,"description":"\xe7\xac\xac\xec\x8d\x81\xe4\xbb\xaa3iPhone","sales":1000,"imgUrl":"32000230&fm=253&fmt=auto&app=120&f=JPEG?w=500&h=500","promoModel":["com.miaoshaproject.ss":1,"promoName":"\xe5\xb9\xb4\xe7\xbb\x88\xe4\xbf\x83\xe9\x94\x80","startDate":"2017-02-00","itemId":1,"promoItemPrice":[\u0024java.math.BigDecimal\u201c,2500]]}]]"
127.0.0.1:6379[10]>
```

图4.8 Redis 商品详情页

4.2.2 本地热点数据缓存的接入

热点数据顾名思义是指多数业务请求都命中的一份缓存数据，电商的热门秒杀商

品便可以视作热点数据。这些热门商品数据量不大，但是访问频繁，只是使用现有的 Redis 的集中式缓存很难满足需求，因此需要增加一种新的缓存方案来存放热点数据。解决缓存热点的常用方法是复制多份缓存副本，将请求分散到多个缓存服务器以减轻缓存单台缓存服务器的压力^[23]，但是再优秀的缓存服务器也会有网络开销，其性能容易被网络因素所限制，所以本系统使用本地内存存放热点缓存，这样的做法彻底解放了缓存服务器，不会对其造成任何负荷，同时相较于缓存，访问本地内存中的数据会更高效。

传统的本地缓存方案是使用 ConcurrentHashMap，它是一种类似于 HashMap 并且支持并发读写的数据结构，同时也是线程安全的。使用 ConcurrentHashMap 提供的键值对的存储方式，将商品 id 作为 key，商品模型作为 value 即可完成本地缓存的设计。但是 ConcurrentHashMap 作为本地缓存有两个缺点：第一，ConcurrentHashMap 是将整个数据结构分为多个段，每个段分配一把分段锁。在进行写操作加上写锁后会影响到读操作的效率；第二，本地内存空间十分宝贵，而 ConcurrentHashMap 缺乏有效的控制内存大小的手段，存储的键值对过多会直接影响到程序的运行。

本文使用谷歌提供的 Guava Cache 来管理本地缓存，Guava Cache 是谷歌所开发的一款开源本地缓存工具库，相较于传统的 ConcurrentHashMap，它有以下优点

（1）可控制的键值大小和超时时间。

在初始化 Guava Cache 时可以设置初始键值对容量和最大容量。

（2）可配置的 LRU 策略。

LRU 即 Least Recently Used，在键值对超出设置容量后，会自动删除掉访问次数最少的键值。

Guava Cache 可以完成弥补 ConcurrentHashMap 的缺点更适合作为本地缓存。以下是具体的实现方式：

先在工程的 pom 文件中引入 com.google.guava 的依赖，使用的是 19.0 的版本。在 Service 层中新建一个 CacheService，其中包含两个方法 setCommonCache () 和 getFromCommonCache () 分别用来存和取；在其实现类 CacheServiceImpl 中引入一个 Cache 类，以键值对的形式储存。

然后为其定义一个初始化 init () 方法，使用谷歌提供的 CacheBuilder 类。调用 CacheBuilder.newBuilder().initial () 方法，设置初始容量为 20 个键值；maximumSize 最大容量设为 100 个键值，超过该容量后会按照 LRU 的策略移除缓存；缓存过期的时间方面 Guava Cache 提供了三种选择：

expireAfterAccess:指定缓存在一定时间没有访问，会将该 key 过期。 expireAfterWrite:指定缓存在写入后一定的时间会被过期。 refreshAfterWrite:缓存项更新操作之后的一定时间被刷新。

考虑到该缓存中存放的是热点数据，很可能会长期存在，如果设置为访问后过一段时间过期，则可能永远无法过期，所以选择 refreshAfterWrite 的过期策略，设置为写入缓存后两分钟过期，因为是热点数据，所以过期时间应比 Redis 缓存短很多。

进入 Controller 层中，引入 CacheService。在其中的 getItem 查询商品方法里先调用 CacheService.getFromCommonCache()方法，传入商品 id；若该层缓存中没有命中再去 Redis 缓存中获取商品；如果 Redis 中也没有命中，则访问下游 Service 层从数据库中读取，接着将获取到的商品模型 set 到 Redis 和 Guava Cache 中。

4.2.3 Nginx 缓存的接入

目前的架构中，Nginx 是离用户最近的节点，本着将缓存设置的离用户越近就可以更好的提升性能，越高效的使用服务器的理念，将最后一级缓存架设在 Nginx 服务器上。以下是一些在设置在 Nginx 服务器上的缓存方案。

第一种方案，使用 Nginx 代理缓存。该缓存功能是集成在代理模块中的，Nginx 将缓存数据当做文件持久化在服务器磁盘中，当收到客户端请求时 Nginx 会在磁盘中靠文件系统搜索缓存信息^[24]。进入 Nginx.conf 配置文件中进行配置：

```
proxy_cache_path /usr/local/openresty/nginx/cache_temp levels=1:2;
keys_zone=tmp_cache:100m inactive=2d max_size=20g;
```

图4.9 proxyCache 配置

申明一个 cache 缓存节点的路径，cache_temp 为自定义的缓存名称，proxy_cache_path 为缓存路径；levels 在 /usr/local/openresty/nginx/tmp_cache 目录下设置了一个两级层次结构的目录。之所以设置两级目录，是因为考虑到将大量文件设置在单个目录中会导致文件访问缓慢，所以大多数情况下都会设置至少两级目录，将文件分散在多个目录中会一定程度上减少寻址的消耗；keys_zone 设置一个共享内存区，该内存区用于存储键值，将键的拷贝放入内存可以使 Nginx 在不检索磁盘的情况下快速决定一个请求是否命中，这样大大提高了检索速度。本文为键的存储空间设置为 100m；inactive 指定了项目在不被访问情况下能够在内存中保持的时间，区别于将数据过期的操作 inactive 会将数据直接删除，时间设置为 2 天；max_size 即最大缓存空间，设置了缓存最大值。如果缓存空间已满，处理器便会调用 cache manager 来移除最近最少被使用的文件，默认覆盖掉缓存时间最长的资源。

然后将该 cache_temp 添加到 Nginx.conf 文件的 location 代码块中。再次发出查询请求可以看到第二级目录中新生产的 tmp_cache 以及其中的缓存在 Nginx 本地的商品详情。如图 4.10：

```
[root@iZbp19yk0c2fd9azh4bifz nginx]# ls
client_body_temp  conf  fastcgi_temp  html  logs  proxy_temp  sbin  scgi_temp  tmp_cache  uwsgi_temp
[root@iZbp19yk0c2fd9azh4bifz nginx]# cd tmp_cache/
[root@iZbp19yk0c2fd9azh4bifz tmp_cache]# ls
[root@iZbp19yk0c2fd9azh4bifz tmp_cache]# ls
[root@iZbp19yk0c2fd9azh4bifz tmp_cache]# ls
[root@iZbp19yk0c2fd9azh4bifz tmp_cache]# cd 8
[root@iZbp19yk0c2fd9azh4bifz 8]# ls
86e4dkjg35431eeck465
[root@iZbp19yk0c2fd9azh4bifz 8]# cat 86e4dkjg35431eeck465
{"status": "success", "data": {"id": 1, "title": "Iphone XR", "price": 4299, "stock": 50, "description": "第十代iPhone", "sales": 1000, "imgUrl": "https://img.baidu.com/u=3181983035,3732002305&fm=253&fmt=auto&app=120&f=JPEG?w=500&h=500", "promoStatus": 1, "promoPrice": 2500, "promoId": 1, "startDate": "2022-12-21 17:01:33"}}

[root@iZbp19yk0c2fd9azh4bifz 8]# 
```

图4.10 Nginx proxy cache 缓存

但是使用 Nginx 代理缓存有以下缺陷：虽然 Nginx 没有将请求转发给后端服务器直接在 Nginx 上做了缓存返回，但是该缓存本质上读取的仍然是本地的文件，并没有将文件的内容缓存在 Nginx 的内存中，每次访问的都是从磁盘上进行读写，难以跟缓存的读取速度相比；同时，Nginx 代理缓存是以文件的方式储存的，对内存的消耗较大。

第二种方案，使用 Nginx 共享内存字典，该字典是一种跨越多个 worker 进程的，基于字典的数据共享方式，该功能由 NginxLuaModule 提供。字典数据结构采用键值对的存储结构，读取速度快，而且可以指定 LRU 的淘汰规则，相较于 Nginx proxy cache 性能更优。

同样使用 Openresty 平台开发 Nginx，开发语言使用 Lua，Lua 是一种轻量级的脚本语言，其设计的目的是通过灵活的嵌入应用程序中，从而为应用程序提供灵活的扩展和定制功能，具有功能强大、代码简洁、小巧灵活、易于集成、运行速度快、易于扩展等特点^[25]；同时 Nginx 为 Lua 留下了许多插载点。这些特点使得 Lua 成为一种用于开发 Nginx 的最好的脚本语言^[26]。

先在 Nginx 上部署 OpenResty；在 Nginx 的配置文件 Nginx.conf 中对共享字典进行配置。使用 shared_dict 代码新建一个名为 my_cache 的共享字典，内存上限设置为 128m。再编写 lua 程序，完成对商品详情的存取，代码如下：

```
unction get_cache(key)
local cache_nginx = ngx.shared.my_cache
local value = cache_nginx:get(key)
return value
end

function set_cache(key,value,exptime)
if not exptime then
exptime = 0
end
local cache_nginx = ngx.shared.my_cache
local succ,err = cache_nginx:set(key,value,exptime)
```

```

return succ
end
local args = ngx.req.get_uri_args()
local id = args["id"]
local item_model = get_from_cache("item_..id")
if item_model == nil then
local resp = ngx.location.capture("/item/get?id=..id")
item_model = resp.body
set_to_cache("item_..id",item_model,2*60)
end
ngx.say(item_model)

```

`get_cache` 方法定义一个变量 `cache_nginx` 来获取 `my_cache` 的内容，通过传入 key 值来返回对应的 value；

`set_cache` 方法传入键值和过期时间，过期时间默认为 0，将键值 set 进 `my_cache` 中，lua 语言支持多重返回结果，`set` 方法返回结果可以有 `succ` 成功或者 `err` 失败，只返回 `set` 数据成功的情况；

访问逻辑为：通过 Nginx 的 get 请求中携带的参数，即 `id` 来获取对应的商品，如果共享字典中没有，则将请求转发到反向代理的 `backend server`，从后端获取到后放入字典中，设置缓存时间为 2 分钟，如果字典中有则直接返回即可。

以上完成了三级缓存的设计整体访问流程如图：

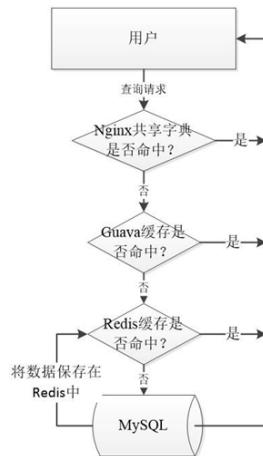


图4.11 多级缓存访问流程图

4.3 缓存数据一致性

在引入缓存以后，会先将商品信息保存在缓存中，用户下单会直接在缓存中扣减

库存，此时容易导致数据库与缓存的库存信息不一致，产生脏数据，如何确保缓存与数据库的数据一致是本节要研究的内容：

4.3.1 CAP 理论的介绍

在计算机科学中,CAP 理论又称之为布鲁尔定理(Brewer's theorem),目前已成为分布式系统设计与构建的重要理论基石^[27]。该理论在 1998 并在 2001 得到了证明。该理论包含三个指标：一致性(Consistency)，使用性(Availability)以及分块容忍性(Partition Tolerance)。

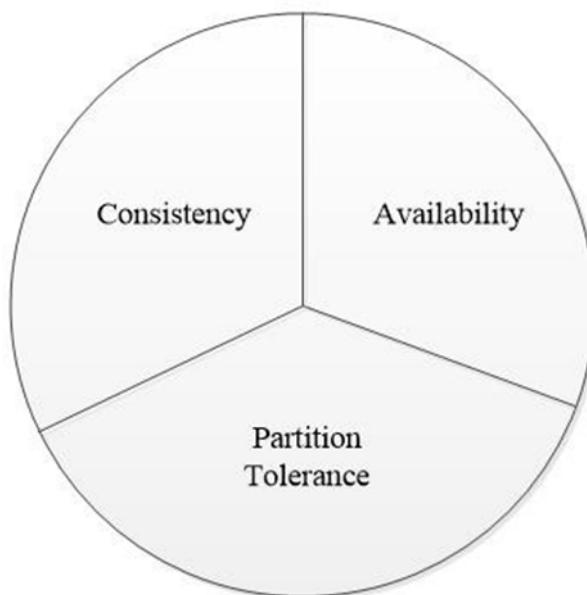


图4.12 CAP 理论

Consistency, 强一致性，也称原子一致性或线性一致性。强一致性有两个要求：第一，要求任何一次读取数据都能读到某个数据最近一次写入的最新数据；第二，在系统中的所有进程，看到的操作顺序都和全局时钟下的顺序一致。如果满足以上两点那么这样的系统就被认为具有强一致性。不难看出，强一致性强调的是数据复制必须是同步的也因为如此，强一致性的实现很多时候需要牺牲掉响应速度。

Availability, 即可用性，是指系统提供的服务必须一直处于可用的状态，对于用户的每一个请求都能够在一定的时间内做出响应。即对用户的请求必须在一个可以接受的时间内，返回一个明确的响应结果。

Partition Tolerance, 分区容忍性，指分布式系统在遇到网络故障等原因导致，仍然能够对外提供完整的正确的服务。

CAP 理论指的是一个分布式体系无法同时满足强一致性，可用性与分区容忍性，最多只能满足其中的两种^[28]。

4.3.2 数据一致性方案

数据一致性通常表示在系统运行过程中缓存中数据与源数据库中的数据是否保持一致^[29]。考虑到使用了多级缓存，一定会有大量商品信息的副本，如果数据库中信息发生更新，就会产生脏数据，因此保证数据一致性的至关重要的。而数据一致性也分为以下几种：

(1)强一致性：又称原子一致性。强一致性表示系统在任何时刻提供给所有用户的数据都是最新的，缓存中的数据和源数据库中的数据保持相同^[30]。当更新操作完成之后的每一次访问，读取的都是更新后的最新的数据。要实现强一致性需要满足两个要求：第一，任何一次读数据都能读到该数据最近一次更新的数据；第二：保证系统的所有进程的顺序一致。也是最难实现的一种。

(2)弱一致性：弱一致性指在某数据更新操作之后，存在系统的源数据库和缓存数据不一致的状态^[31]，不保证任何一次都能读到最近一次更新过的值，但可以保证最终可以读到正确的数据^[32]。在数据更新后，如果能容忍后续的访问只能访问到部分，即是弱一致性。弱一致性的实现方法较多，如何选出一种能满足要求的是问题的关键。

(3)最终一致性与 BASE 理论：基于 CAP 理论，从强一致性，可使用性与分区容忍性中进行取舍。首先考虑到分布式架构中，不同的组件一定会被分开部署到不同的服务器上，如果连分区容忍性都无法保证，那分布式的实现就毫无意义；其次，如果一款互联网应用没有了可用性，那么任何的架构设计都是空谈，所以只有一定程度上牺牲 Consistency，即强一致性。当然，数据的一致性依然要保持，但是不过分追求数据库数据与副本数据的瞬时跟随，只要求两者最终一致即可。由此将 CAP 演变为 BASE 理论，即最终一致性，又称 Basically Available Soft-state Eventual Consistency 柔性事务^[33]。在数据库更新操作中，会有一些副本处于一个 Soft-state 软状态中，与数据库不同步，这种情况被视为是可以容忍的，因为这些副本在经过到一定时间，等到系统不再发生更新事件后，最终会恢复到正常状态，与数据库最终保持一致。

基于以上理论，提出以下最终一致性的实现方案。

更新缓存方案。包括先更新缓存后更新数据库以及先更新数据库再更新库存，如图 4.12 和 4.13：

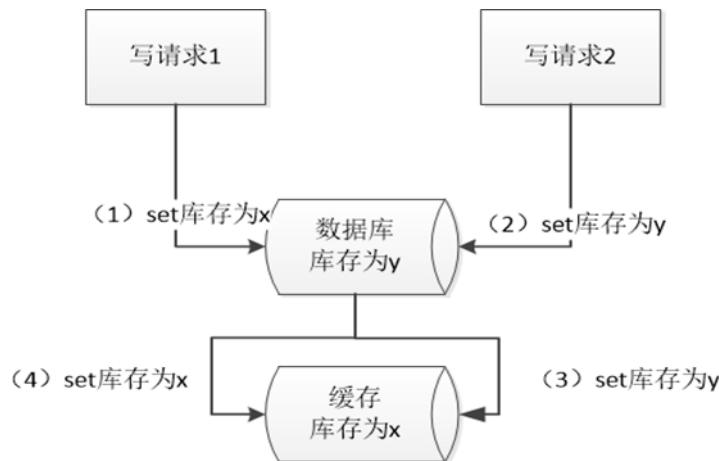


图4.13 先更新数据库再更新缓存方案

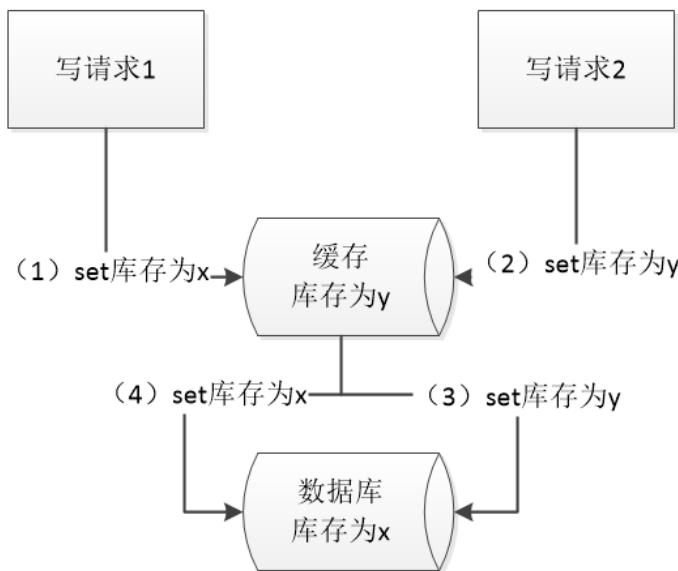


图4.14 先更新缓存再更新数据库方案

可以从图中看出更新缓存的两种方案有明显的缺点，即两个写请求很可能因为网络原因导致执行顺序不同，一个更新操作被另一个更新操作覆盖最终出现数据不一致的情况；除此之外，更新缓存本身就是一个效率较差的选择，相较于直接删除缓存，更新操作更复杂，而且缓存都是有过期时间的，更新后的数据到了过期时间本来就会被删除掉，所以更新缓存的方案不仅没法确保最终一致性，还会因为更新操作浪费 IO，CPU 资源。

由此提出第二类方案，删除缓存方案。

方案一，先删除缓存再更新数据库如图 4.15：

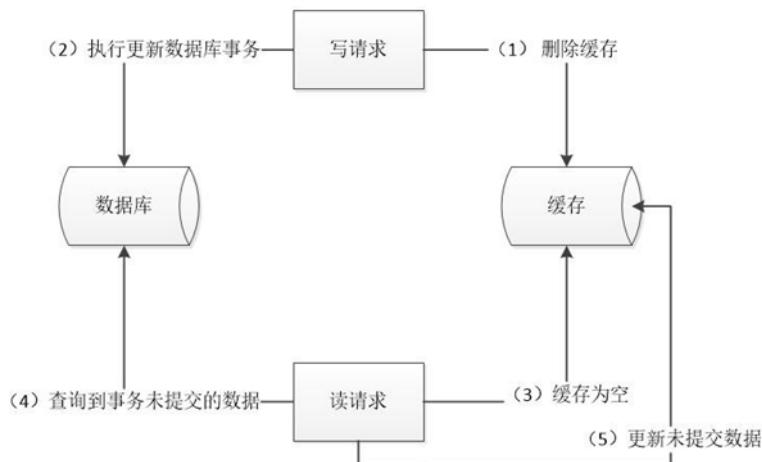


图4.15 先删除缓存再更新数据库方案

种方法的缺点较为明显，在写线程删除缓存去更新数据库，但更新还未完成之前，读线程因为缓存被删进入数据库读取到了未更新的脏数据并放回缓存，还是会出现数据不一致的情况。

方案二：先更新数据库再删除缓存，但是这个方法也可能造成数据不一致，如图：

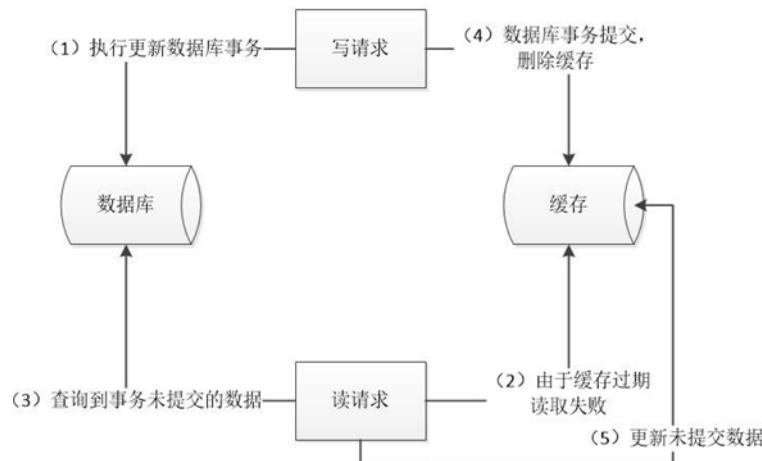


图4.16 先更新数据库再删除缓存方案

写线程先执行更新数据库的事务，如果更新还没完成时缓存数据刚好过期，那么读线程会去数据库中读取未提交的数据，然后数据库才提交事务并删除缓存，读线程再将读到的未提交的数据并写入缓存。如果以这样的顺序执行，那么仍然会出现数据不一致的情况。这种情况发生几率较低。在对并发性要求不高的使用场合，先更新数据库再删除缓存的方案几乎可以胜任，但是对于秒杀商城来说，这样的方法还不足以应对。

以上几种方案都有几率出现数据不一致的情况，并且也无法确保在数据库更新后下一次用户获取到的是最新的数据，最根本的原因是缓存与数据库本就是两个不同的

系统，缺乏有效手段来保证两个系统中扣减库存操作的原子性。

4.4 消息队列异步扣减库存

4.4.1 异步消息队列的介绍

显然，只是简单地调整删除缓存跟更新数据库的顺序不足以满足高并发应用场景，为了更好的解决上文中的问题，引入异步消息队列的方案。

消息队列即 MessageQueue，顾名思义是由消息组成的队列，是分布式系统中的重要中间件，通过该中间件将缓存与数据库联系起来^[34]。就像其他队列一样，MQ 也遵循“先入先出”的规则，消息队列主要用作消息传输过程中保存消息的容器，在高并发大流量访问的应用场景下，由于系统性能不足，造成请求堵塞，例如大量的插入，更新操作几乎同时到达，直接导致产生无数的行锁导致请求堆积过多，极有可能会导致数据出错，甚至系统宕机，严重影响业务。使用异步消息队列区别于串行的方式，生产者生产消息存放到队列中，写入队列的时间很快，消费者从中取走并处理，两种操作是解耦的、异步的，不会产生请求堆积的情况^[35]。将发送信息跟消费信息的操作解耦，这将使各个系统可以独立开发。

MQ 的实现可以使用 RabbitMQ, ActiveMQ, 以及 Kafka 等，本文选择 RocketMQ。其原理如图 4.17 所示：

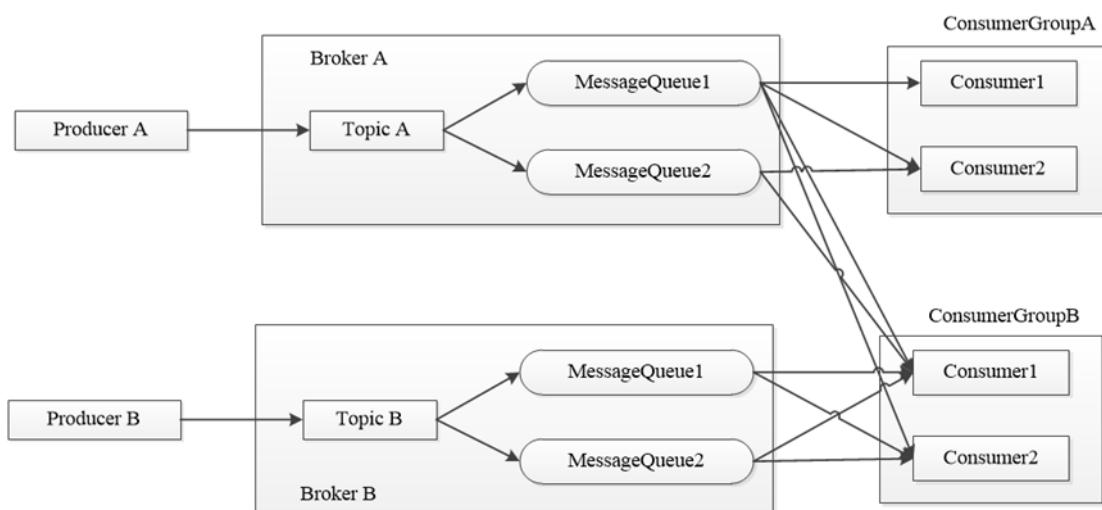


图4.17 RocketMQ 原理

其中包含以下概念：

Producer:消息生产者，主要的作用就是将信息发送到消息队列中。生产者本身既可以产生消息。也可以对外提供接口，由外部应用来调用接口，再由生产者将收到的消息发送到消息队列^[36]。

Producer group: 生产者组，简单来说就是多个 producer 的组群。Producer 可以发送多个 Topic。处理分布式事务时，也需要 producer 集群提高可靠性。

Topic: 消息主题，不同的消息有不同的 topic。例如商城有订单类的消息，也有库存类的消息，那么就需要进行分类。一个 topic 下可以有多个消息队列。

Consumer: 消息消费者，简单来说，消费 MQ 上的消息的应用程序就是消费者，至于消息是否要消费掉，还是存储到数据库等取决于业务需要。

Consumer Group: 消费者组，和生产者组类似，消费同一类消息的多个 consumer 实例组成一个消费者组实例的集合。同一个 Group 中的实例，在集群模式下，以均摊的方式消费；在广播模式下，每个实例都全部消费。

Message: Message 是消息的载体。一个 Message 必须指定 topic。RocketMQ 提供了一个实现了序列化的 Message 类，该封装类包括 Properties 和 Body，Properties 可以对消息进行修饰，如消息的优先级、传输格式（如 JSON）、延迟等高级特性，Body 则是消息体内容。Message 类有三种状态：待消费状态、待应答状态和总数状态。Broker: Broker 是 RocketMQ 系统的主要角色，其实就是前面一直说的 MQ。Broker 接收来自生产者的消息，储存以及为消费者拉取消息的请求做好准备。

NameServer: NameServer 为 producer 和 consumer 提供路由信息，所有的 broker，consumer 以及 producer 都要在 NameServer 上做注册。Producer 是从 Name Server 中拉取 broker 的信息，consumer 也是如此。

整体流程为，当 broker1 启动时会向 NameServer 发生一个注册的请求并主动告知 NameServer 自己的 ip 地址，负责的 topicA 以及消息队列；producer 同样连接到 NameServer 并发现注册在其中的 broker1，如果该 producer 要发布一条 topic 为 A 的消息的时候就会往 broker1 中投递；如果 topicA 下有多个消息队列的话就轮询投递。接着 consumer 启动，去 NameServer 中以 topic 为单位抓取，如果对应 topic 下有消息队列，则与这些队列建立长轮询并获取其中的消息进行消费，在确认消息被消费掉后将消息删除，如果队列中没有消息则会等待并不断开连接，当 producer 再次投递消息时 consumer 的这些连接会被唤醒再次试图消费。Consumer 的消费方式有两种，一种是广播消费，意为一个 Topic 中的每一条消息都会被一类 Consumer 中的每个 Consumer 消费；第二种是集群式消费，意为一类 Consumer 共同完成对一个 Topic 的消费。

4.4.2 库存缓存异步化的设计与实现

在服务器上下载并安装 RocketMQ，使用 nohup sh bin/mqnamesrv & 命令启动 NameServer 服务器；nohup sh bin/mqbroker - n localhost:9876 & 命令启动 Broker 服务器。剩下的 Producer 与 Consumer 主要以代码实现。

首先在 pom 文件中引入 RocketMQ 需要的 jar 包，使用的是 4.3.0 的版本。

然后在 application.properties 文件中设置 MQ 的远端地址 mq.nameserver.addr 以及该 MQ 对应的主题 mq.topicname。

定义两个类分别作为 Producer 以及 Consumer。如图 4.18 所示：

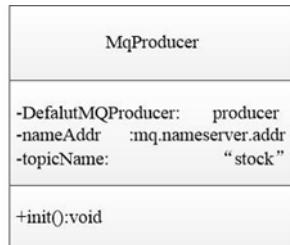


图4.18 MqProducer 类图

MqProducer 中首先引入一个默认消息队列生产者 DefaultMQProducer 作为消息中间件的消息队列生产者使用。然后通过@Value 注解将 application.properties 中的 nameserver 地址和主题名注入进来；定义 init () 方法对 producer 进行初始化，设置 nameserver 的地址，设置 topic 名为 stock 后直接调用 start 方法来启动。

然后定义一个 asyncReduceStock () 方法真正实现库存扣减信息的发送。该方法传入两个参数商品 id 以及下单数量 amount，返回一个布尔值类型的值用来表示消息发放是否成功。考虑到 Redis 键值对的储存形式，选择使用使用 HashMap 将需要发布的消息储存起来。发送时直接调用 send 方法，先指定发布消息的 Topic，然后将 HashMap 中保存的消息转化为 Json 格式一起发送。

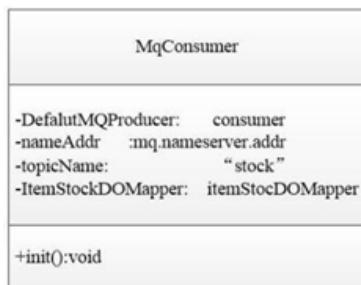


图4.19 MqProducer 类图

类似的，设计 MqConsumer 如图，同样也引入默认消息队列消费者 DefaultMQConsumer，在初始化方法 init () 中设置 nameserver 的地址后。调用 subscribe(topicName, “*”) 监听发布的消息，“*”则表示监听 stock 主题下的全部消息，接着实现一个 registerMessageListener 的回调函数用来设置获取到消息后如何去消费：同样将传入的消息放在一个 Hashmap 中，获取其中的商品 id 以及下单数量 amount。然后调用商品 Mapper 文件中的减库存方法 decreaseStock，传入商品 id 以及下单数量，真正完成了数据库库存的扣减。

完成了 producer 与 consumer 后, 与在 Service 层中引入 mqProducer, 在 ItemService 的实现类中的 DecreaseStock 减库存方法中, 如果扣减库存成功, 则调用 producer 的 asyncReduceStock () 方法发送一条消息, 让异步消息队列感知到。如果消息发送失败或者抛出异常, 则用 RedisTemplate 将缓存中的库存补回去。接入消息队列后扣减库存的流程如图 4.20 所示:

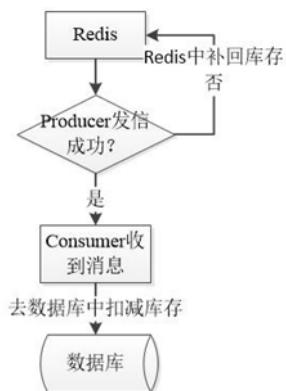


图4.20 异步扣减库存流程图

这样的设计将消息的发送和消费全部封装在方法中, 只需直接调用即可。以上结合下单操作, 系统会先修改 Redis 缓存中的数据, 调用 MqProducer 中的减库存方法, 然后发出消息。MQConsumer 接收到消息以后再完成真正的数据库减库存的操作。若消息发送失败或者抛出异常则调用 RedisTemplate 的 increment 方法将缓存库存再加回去。完成缓存与数据库的最终一致。

4.5 事务型消息

4.5.1 事务型消息的引入

之前的内容使用了异步消息队列的方式, 使得 Redis 缓存中的库存与数据库中的库存达成了最终一致。目前为止系统的扣减库存的流程为: 在 ItemService 中的 decreaseStock () 方法中, 先在 Redis 缓存中扣除库存, 如果扣减成功则发送异步消息给对应的异步回调, 在其中完成数据库中的扣减。如果异步消息的发送出现任何异常, 则将 Redis 中的库存加回去。

然而这样的做法依然有问题, 在 Service 层中的创建订单方法 CreateOrder () 中调用了 decreaseStock () 方法, 而这两者全部都用@Transactional 申明了事务, 关系如图 4.21 所示:

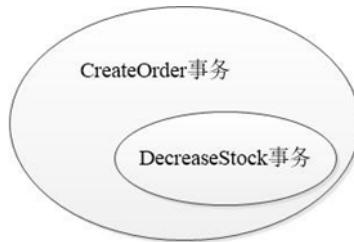


图4.21 创建订单事务与扣减库存事务关系图

扣减库存这一事务被包裹在创建订单这一大事务之内，如果扣减库存这一事务已经执行成功，Redis 中的库存已经扣减，但是创建订单这一大事务中的其他操作出现错误，那么整个创建订单的行为就会失败回滚掉，用户下单失败，真实的库存没有减少但是库存显示却已经扣减掉了。这样虽然不会导致超卖，但是却会出现少卖的情况。导致该问题出现的根本原因是，在扣减库存的方法发送异步消息之前无法确保创建订单的其他操作是否会成功。

由此可以提出第一种解决的方法：等到创建订单的事务执行完成以后再进行 Redis 扣减库存和发送异步消息的操作。新定义一个异步更新库存方法 `asyncDecreaseStock ()`，与一个回补库存方法 `increaseStock ()` 方法。等到创建订单中所有的操作都完成之后再进行异步更新库存，如果该更新失败，则回补库存。流程如图 4.22 所示：

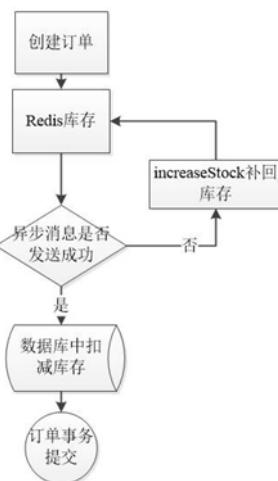


图4.22 扣减库存方案一

但是这种方案也有缺点：如果在整个创建订单事务执行成功准备提交时，因为外部网络原因或者磁盘容量满了导致该事务提交失败，那么还是会出现少卖的现象。

在此基础上提出一种改进方案：在创建订单事务完成并提交以后再异步更新库存，同样的，更新失败则回补库存。使用`@TransactionSynchronizationManager` 注解，该注解是 Spring 提供的事务同步管理器，用于自定义实现 `TransactionSynchronization` 类来

监听 Spring 的事务操作。可以在事务提交之后，回调 TransactionSynchronization 类的方法。通过定义一个 afterCommit() 方法，将异步更新库存的操作放在创建订单事务执行之后进行。

然而该方案也有隐患，即将异步消息的投递放在创建订单之后进行，那么该条消息就必须要投递成功，如果失败就再没有机会去回滚库存。这种情况一旦出现，只能通过回源数据库来补救。考虑到秒杀商城的应用背景，这种问题是绝对无法容忍的。

从根本来看，以上方案都有一个问题没有解决，就是数据库事务与消息队列事务两者的一致性问题。简而言之，数据库的事务跟普通 MQ 事务的消息发送无法直接绑定。为了解决该问题，引入另一种方案：事务性消息，也叫两阶段提交协议（Two-phase Commit, 2PC）。事务消息主要通过消息的异步处理保证本地事务和消息发送的同时执行成功或失败^[37]。

使用事务性消息的流程主要如图 4.23 所示：

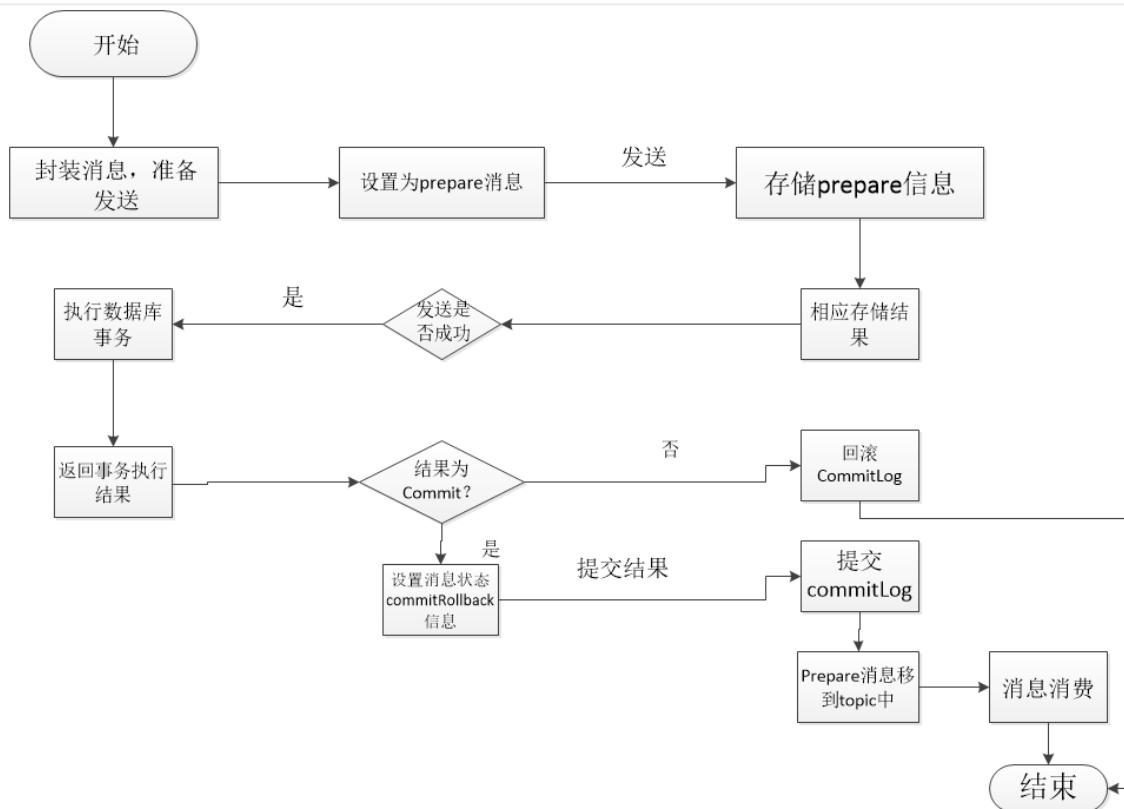


图4.23 事务型消息方案流程图

- (1) 在 Producer 发送消息时，先发送一个 perpare 消息又称半消息给 MQ，该 perpare 消息对 Consumer 是不可见的。
- (2) perpare 消息发送成功后，应用模块执行数据库事务。
- (3) 根据数据库事务执行的结果进行判断，将执行成功 Commit，执行失败 RollBack。

(4) 如果是 Commit, 消息队列把该 prepare 消息下发到 Consumer 端; 如果是 Rollback, 直接删除 prepare 消息; 如果是 Unknown 则再视情况具体处理。

(5) 第三步的执行结果如果没响应或是超时, 启动定时任务回查事务状态。

由以上流程不难看出, 事务消息会根据数据库事务的执行成功与否再做出判断, 能很好的弥补之前两种方案的缺陷。目前能支持事务消息的 MQ 产品越来越多, 比如 RabbitMQ, Kafak 等, 选择 RocketMQ 来实现。

4.5.2 RocketMQ 实现事务消息

首先在之前的 MqProducer 中需要实现一个 TranscationListener 的事务监听器的接口。该接口包含一个方法: executeLocalTransaction(), 用于执行本地事务, 该方法会返回一个 LocalTransactionState 用于表示本地事务的执行结果。在前者中实现创建订单的操作, 如果创建失败, 会将 LocalTransactionState 设置为 ROLLBACK_MESSAGE 表示事务提交失败需要回滚, Broker 端会删除该半消息; 如果创建成功, 则设置为 COMMIT_MESSAGE 表示本地事务提交成功, 半消息可以被消费。新增一个 transactionReduceStock 方法用于实现事务性扣减库存。使用 TransactionMQProducer 事务消息生产者去发送消息。消费端与之前一致。

进入 Controller 层中的, 在创建订单方法中判断用户已经登录的情况下直接调用 transactionReduceStock 方法去扣减库存。

以上使用了 RocketMQ 实现了事务型消息, 通过判断数据库事务的执行结果来决定 Redis 缓存的行动。相较于之前的方案, 该方法把 producer 发送信息分为两个阶段:

prepare 阶段与确认阶段, 通过异步方式处理消息, 保证了本地事务和消息发送的原子性, 从而最终保证了数据的最终一致性, 并且有效避免了少卖, 超卖的现象。

4.6 本章小结

本章基于查询性能进行了一系列的优化, 首先将 Nginx 做了动静分离, 使前端静态页面不必发送 Ajax 请求来向后端获取, 提升了查询与响应时间; 接着用 Nginx 做了分布式的扩展, 当请求发送到 Nginx 上时, Nginx 会将请求反向代理到两台下游服务器中, Nginx 作为代理服务器本身并不处理业务, 对用户来说 Nginx 的域名是已知的, 用户访问 Nginx, 然后它将请求反向代理给下游的服务器, 这些下游服务器才是真正处理商城业务的部分。这样的做法也保护了后端集群。两台下游服务器采用了轮询的策略; 然后区别于传统的使用 Cookie 的方案, 改用了 token 进行了会话管理。然后设置了多级缓存的策略, 分别在 Redis, 本地以及 Nginx 服务器上设置了缓存。极大程度上提升了查询速度, 同时三层缓存的引入也很好地保护了数据库免于大流量的

冲击。又因为引入多级缓存的原因，必须要对数据一致性进行处理，引入了 RocketMQ 的消息队列并用 RocketMQ 完成了事务消息，用来完成减库存的操作，在保证了数据最终一致性的基础上有效的避免了超卖少卖现象的出现。

第五章 性能测试

本章主要针对高并发下应用场景下商城应用的访问操作进行测试，通过 Jmeter 压测软件创建线程组模拟高并发场景，对商城项目发出查询商品的请求，来测试商城性能并且分析生成的聚合报告来验证上文的优化措施的有效性。

5.1 硬件测试环境

本文中使用的服务器共有 Nginx 反向代理服务器一台，网络应用服务器两台，数据库服务器一台。其配置如表 5.1 所示：

表5.1 硬件配置表

Server	配置
Nginx反向代理服务器	1核 CPU, 1GB内存, 处理器主频2.5GHZ, 内网带宽最高1.5Gbps
网络应用服务器1	1核CPU, 2GB内存, 处理器主频2.5GHZ, 内网带宽最高1.5Gbps
网络应用服务器2	1核CPU, 2GB内存, 处理器主频2.5GHZ, 内网带宽最高1.5Gbps
数据库服务器	2核CPU, 2GB内存, 处理器主频2.5GHZ, 内网带宽最高1.5Gbps

以上服务器均部署在阿里云弹性计算服务器上，使用的操作系统是 CentOS 8.3 的 64 位版本。将项目打包成 jar 包部署到服务器上。

5.2 压测工具介绍

Apache Jmeter 是 Apache 组织开发的完全基于 Java 的压力测试工具^[38]，用于对软件做压力测试。旨在加载测试功能行为和测量性能。Jmeter 是一个纯 java 程序。它的工作原理是向服务器提交请求，并接收服务器提交的请求。通过分析数据或图形结果完成对 http、ftp、junit、jdbc 等性能的测试^[39]。它可以用于测试静态和动态资源，例如静态文件和动态资源、网络动态应用程序的性能、Java 小服务程序、CGI 脚本、Java 对象、数据库、FTP 服务器等等^[40]，也可以用来模拟一个服务器、一组服务器、网络或对象上的重负载，以测试其强度或分析不同负载类型下的整体性能。有能力对

许多不同的应用程序/服务器/协议类型进行负载和性能测试。

Jmeter 的界面如图 5.1 所示：

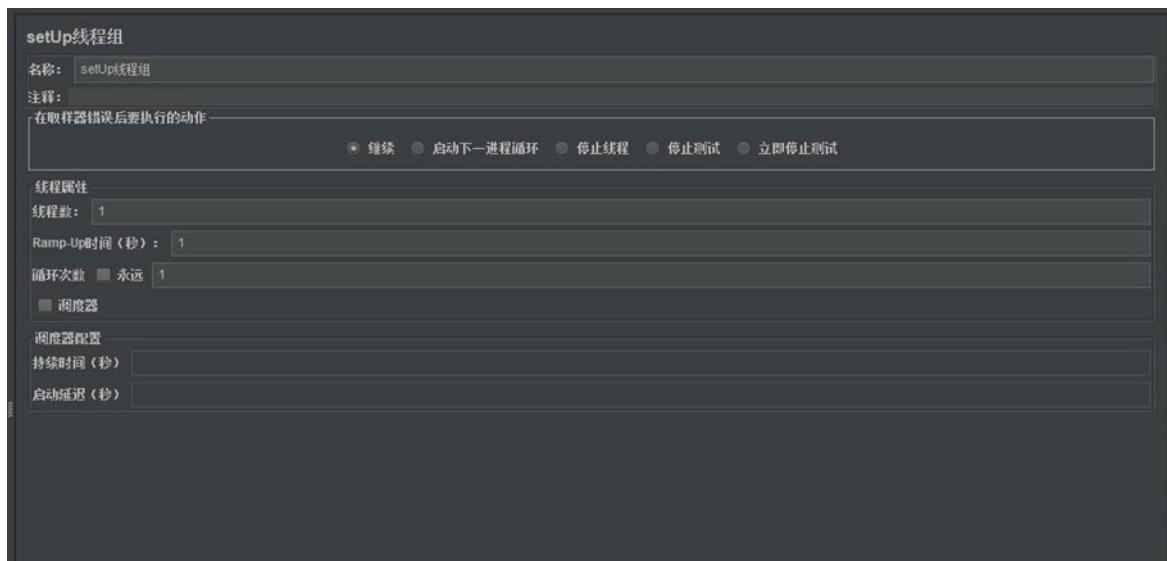


图5.1 Jmeter 线程组设置界面

- (1) 创建一个测试计划，为其中添加一个线程组。
- (2) 对该线程组进行设置，线程数即并发线程数；Ramp-Up 时间表示启动并发线程的时间；循环次数表示循环并发线程的次数。举例：如果线程数设为 100，Ramp-Up 时间设为 10，循环次数设为 10。表示启用 100 个线程，在 10 秒内启动，然后让这 100 个线程循环调用接口 10 次。
- (3) 为该线程组设置请求方式，选择使用的是 http 请求。
- (4) 为该线程组添加结果树，该结果树会将对应的 http 请求的返回结果全部打印出来。
- (5) 为该线程组添加聚合报告，该报告中包含以下性能指标：
Samples: 该次测试中发出的总请求数。
Average: 平均响应时间，默认情况下是指单个请求的平均响应时间，单位为毫秒。
Median: 响应中位数，50% 请求的响应时间，单位为毫秒。
90%Line: 俗称 90 线，90% 请求响应的时间，单位为毫秒。
95%Line: 95% 请求响应的时间，单位为毫秒。
99%Line: 99% 请求响应的时间，单位为毫秒。
Min: 最小响应时间，单位为毫秒。
Max: 最大响应时间，单位为毫秒。
Error%: 错误响应率，本次测试中出现的错误响应的请求数/请求总量。

TPS: Throughput Per Second 一般是指每秒钟完成的请求数，是测试高并发项目的重要指标。

5.3 并发查询压测实验

固定循环次数为 100 次，请求方式设置为 http 请求，请求路径为/item/get?id=1 的查询商品请求。分别测试在 100,500,800,1000 并发数下不同架构的性能表现，来验证上文中各种设计的有效性。不同的架构指的包括第三章中实现的直接从数据库中读数据的基础项目和基础项目加上分布式扩展和各级缓存的架构：

表5.2 查询实验表

架构	并发数	总请求数	平均响应时间 (ms)	95% (ms)	TPS (req/s)
基础项目	100	10000	34	50	322
	500	50000	665	1131	497
	800	80000	897	2214	602
	1000	100000	1073	3521	625
Redis	100	10000	26	39	501
	500	50000	43	87	925
	800	80000	183	325	1925
	1000	100000	221	997	2252
Redis+Guava Cache	100	10000	20	29	423
	500	50000	24	41	1998
	800	80000	37	50	2997
	1000	100000	45	61	3771
Redis+Guava Cache+Proxy Cache	100	10000	37	33	442
	500	50000	44	52	1987
	800	80000	56	57	3005
	1000	100000	73	71	3881
Redis+Guava Cache+Shared Dic	100	10000	16	22	419
	500	50000	19	38	2426
	800	80000	30	44	3701
	1000	100000	39	50	4973

由表中数据可以总结出以下结论：

(1)对商城的分布式扩展+多级缓存的性能优化方案在几项重要性能指标中均有亮眼的表现，平均响应时间相当稳定，并且仅在双机系统的条件下将 TPS 优化到了接近 5000req/s，性能提升显著。

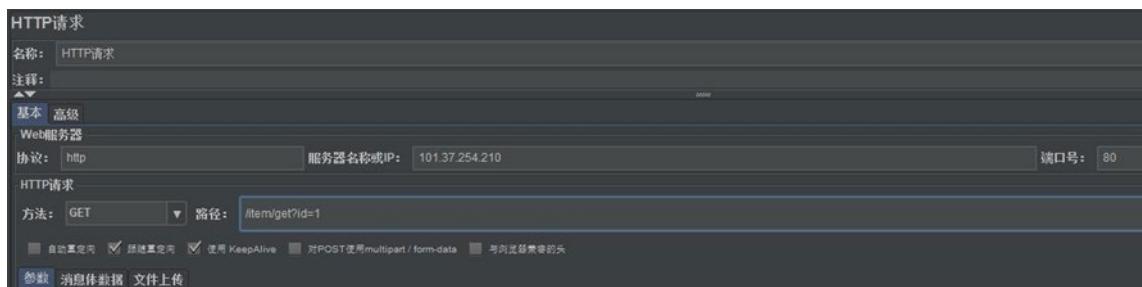
(2)增加了本地缓存 Guava Cache 后，相较于 Redis 的单层缓存在响应时间方面有了明显缩短，但是后续方案在响应时间上提升并不明显，推测原因为增加了本地缓存后为系统减少了多段网络开销，所以响应时间有明显提升；后续方案的响应时间提升有限推测是因为该项指标已经达到了现有硬件系统的性能瓶颈。对服务器进行扩容，物理性能提升或者增加水平扩展的服务器后还有提升空间。

(3)相较于 Redis+Guava Cache+Proxy Cache 的组合，Redis+Guava Cache+Share dDic 的多级缓存在 TPS 上的提升几乎相同，但是前者的响应时间要远远久于后者，这也印证了 4.2 节中关于共享字典的读写速度优于代理缓存的磁盘读写速度的理论。

5.4 数据库服务器查询压测实验

考虑到 Redis 和 MySQL 数据库是部署在同一服务器上的，高并发读写应用背景下数据库服务器的性能至关重要，所以现使用 JMeter 与 Linux 提供的 top 命令对数据库服务器的性能进行分析，由 JMeter 设置线程组模拟高并发的使用场景。

第一组实验，JMeter 线程组添加 1000 个线程，循环 30 组，仍然是/item/get?id=1 的商品详情的访问，直接访问数据库，请求路径为数据库与 Redis 的服务器地址 101.37.254.210。如图 5.2 所示：



开始压测，然后使用 top 命令查看数据库服务器的性能分析如图 5.3 所示：

```
top - 23:45:58 up 2:10, 1 user, load average: 1.95, 0.75, 0.45
Threads: 973 total, 3 running, 970 sleeping, 0 stopped, 0 zombie
%Cpu(s): 80.0 us, 12.1 sy, 0.0 ni, 3.5 id, 0.0 wa, 0.0 hi, 4.5 si, 0.0 s
KiB Mem : 8010456 total, 6021644 free, 1636820 used, 351992 buff/cache
KiB Swap: 0 total, 0 free, 0 used. 6121988 avail Mem
```

图5.3 直接访问数据库的性能检测图

其中有几项重要数据需要重点关注：

(1) %Cpu(s): 80.0us, 12.1sy 表示用户空间程序的 CPU 使用率一度已经达到 80%，系统程序的 cpu 使用率也达到了 12.1%。

(2) load average: 1.95, 0.75, 0.45 分别表示 1 分钟，5 分钟, 15 分钟的 CPU 平均负载，数据库服务器采用的是 2 核 CPU，而在第一分钟 CPU 平均负载的峰值已经达到了 1.95，说明在这种情况下数据库的物理性能几乎已经达到极限。

第二组实验，添加同样的线程组，访问同样的商品，这次访问的路径是 Nginx 反向代理服务器，通过分布式架构与多级缓存到达数据库，再用 top 命令查看数据库服务器 CPU 性能如图 5.4 所示：

```
top - 23:51:09 up 2:15, 1 user, load average: 0.37, 0.31, 0.35
Threads: 992 total, 4 running, 110 sleeping, 0 stopped, 0 zombie
%Cpu(s): 5.5 us, 6.2 sy, 0.0 ni, 76.2 id, 0.0 wa, 0.0 hi, 12.1 si, 0.0 s
KiB Mem : 8010456 total, 2992196 free, 172604 used, 4845656 buff/cache
KiB Swap: 0 total, 0 free, 0 used. 7513480 avail Mem
```

图5.4 优化后的数据库服务器性能图

可以看到使用了现在的架构后 CPU 使用率的峰值只有 5.5%，CPU 平均负载的峰值也只有 0.37，由此可见整个系统的优化行之有效，极大程度上缓解了服务器的巨大压力。

5.5 库存缓存数据一致性实验

除了高并发场景下的查询商品的操作，还要对高并发场景下的扣减库存的操作进行实验。考虑到实际应用场景，将 itemId 为 1 的商品库存设置为 10000 件，然后在 JMeter 中创建线程组对商城发出下单扣减库存的请求模拟秒杀活动中的多人下单的场景；通过比对 Redis 缓存中的库存数量与数据库中的实际库存数量来判断缓存一致性。

分别对上文中提及到的延时双删以及实现了事务消息的消息队列异步扣减缓存这两种扣减库存方案进行对照试验，其中延时双删方案的延时时间设置为 500ms。

表5.3 缓存一致性实验表

扣减库存方案	总请求数	平均响应时间 (ms)	Redis 库存数量	数据库库存数量
先删除缓存 再更新数据库	1000	64	9000	9011
	5000	124	5000	5094
	10000	351	0	107

先更新数据库 再删除缓存	1000	70	9000	9000
	5000	116	5000	5000
	10000	327	19	19
实现了事务消息 的异步消息队列	1000	25	9000	9000
	5000	82	5000	5000
	10000	264	0	0

由上表可以得出以下结论

第一：先删除缓存再更新数据库的方案不可行，无法保证数据一致性

第二：先更新数据库再删除缓存的方案在并发量较低时可以同时保证数据一致性和正确性；但当并发量来到 10000 时，虽然保证了数据一致性，但是会出现因为更新数据库失败而出现少卖现象

第三：实现了事务消息的消息队列使用了消息中间件 RocketMQ 将缓存与数据库两个不同的系统关联起来，并且通过事务消息保证了在两个系统中扣减库存的操作的原子性，很好地保持缓存与数据库的一致性和正确性，同时拥有更快的响应速度，是解决分布式数据一致性问题的可行方案。

5.6 本章小结

本章使用 JMeter 模拟高并发场景，分别对不同的缓存和扣减库存方案进行查询和下单的测试，通过分析聚合报告中的吞吐量，平均响应时间等重要指标可以得出以下结论：Nginx 分布式扩展加上包括 Redis，GuavaCache 和 Nginx Shared Dic 的多级缓存和 RocketMQ 消息中间件的组合方案可以在保持数据一致性的基础上应对高并发的访问，并且目前的系统仍然有很强的扩展能力，可以根据业务需求随时进行扩展。

第六章 总结与展望

6.1 工作总结

本文使用 SpringBoot 框架设计了基于 SSM 架构的商城项目，完成了数据库和领域模型的设计，并且实现了注册，登陆，创建商品，交易下单和秒杀促销等基础功能。针对高并发秒杀商城的应用场景，提出了分布式扩展+多级缓存+消息中间件的优化思路。然后通过 JMeter 压测工具对不同的缓存和数据一致性方案进行压测，分析聚合报告中的关键数据指标，最终确立了 Nginx 反向代理和 Redis，Guava Cache 加上共享字典的多级缓存以及实现了事务消息的 RocketMQ 作为消息中间件的整体优化方案。通过压力测试实验也可以验证该方案的有效性。

6.2 研究展望

本文对于秒杀商城的应用研究提供了一套行之有效的解决方案，但是随着研究的深入，了解到还有许多研究方向与优化方案在本文中没有涉及到，包括：

- (1) 用户界面的优化。本文对于前端界面的设计较为简单，用户界面还不够美观，未来可以尝试使用更多 CSS 样式对网页进行美化，优化用户体验
- (2) 使用 phantosjs 全页面静态化技术，在服务端完成 html，css 的加载工作，渲染成纯静态 html 文件，然后将这些文件直接以静态资源的方式部署到内容分发网络 CDN 上。这样的做法可以极大程度上提升响应系统速度，也更利于网站的稳定性与安全性。
- (3) 引入流量削峰技术。使用秒杀令牌和队列泄洪等削峰技术来削弱秒杀活动刚开始时涌入系统的请求高峰，让商城系统吞吐量在高峰请求下依然保持可控。

参考文献

- [1] BOUTAH Sainamthip. Web 服务器性能测试研究[D]. 昆明: 昆明理工大学,2020.
- [2] 聂林海. 我国电子商务发展的特点和趋势[J]. 中国流通经济,2021(6):97-101.
- [3] 张守营. 我们需要一个怎样的“双十一” [J]. 中国经济导报,2021-12-29,23(07):33-35.
- [4] 周贺阳. 基于机器学习的电商秒杀商品点击率预估模型设计与实现[J]. 电子技术与软件工程,2021(23):150-152.
- [5] 吴自晨. 从美国看世界电子商务的发展[J]. 大经贸,2008,(04):13-15.
- [6] 张佳强. 基于领域模型的信息管理系统的研究与应用[D]. 无锡 :江南大学,2020
- [7] MOHAMMED ALI A,KADHIM FARHAN novel improvement with an effective expansion to enhance the MD5 hash function for verification of a secure E-document[J] IEEE Access,2020,8:80290-80304.
- [8] 苑思明, 郑晗, 李俊杰. 基于哈夫曼树压缩的加密技术[D]. 信息记录材料,2018.
- [9] 刘倩, 吕艳娜. 基于 MD5 的 IP 域域网动态 IGP 协议加密部署[D]. 中国联合网络通信有限公司河南省分公司.
- [10] 龚兰兰,凌兴宏. 基于敏捷开发 SSM Web 应用开发实践[J]. 苏州 :苏州大学计算机科学与技术学院,2019(123):29-33.
- [11] 陈峰. 基于 SSM 框架的 B2C 网上商城系统的设计与实现[D]. 湖南 :湖南大学,2018.
- [12] 王玉龙. 基于 Spring 的健康服务云平台医生端的设计与实现[D]. 北京 :北京交通大学,2019.
- [13] CL Smith,DA Hantula.Pricing effects on foraging in a simulated Internet shopping mall[D].Journal of Economic Psychology,2003.
- [14] 范新灿.基于 Struts、Hibernate、Spring 构架的 Web 应用开发[J]. 北京:电子工业出版社,2011(228):397-416.
- [15] 朱来雪. 基于 Nginx 技术的 Web 系统安全部署方案[J]. 信息与电脑(理论版),2019,31(17):172-173.
- [16] 姬渭孟,于雪莲.基于数据库代理实现数据库分表、分库访问的方案研究[J]. 数字通信世界,2019, (09): 238.
- [17] 张佳强. 基于领域模型的信息管理系统的研究与应用[D]. 无锡 :江南大学,2010.
- [18] 姬晓涛,刘建华.基于 CAP 理论的区块链共识机制的分析[J]. 计算机与数字工程,2020,48(12): 3008-3011+3026.
- [19] Efficient load balancing over asymmetric datacenter topologies[J]. Syed Mohammad Irteza, HafizMohsinBashir,TalalAnwar,IhsanAyyubQazi,FahadRafiqueDogar.ComputerCommunicat

- ions, 2018.
- [20] 孙青.分布式数据库多级缓存系统设计与实现[D].武汉 :华中科技大学,2020..
- [21] 陶辉. 深入理解 Nginx: 模块开发与架构解析第 2 版[M]. 机械工业出版社.2018.
- [22] 仲灵毓. Redis 存储系统在广电监测系统中的应用[J]. 现代电视技术,2022(02):139-142.
- [23] 河马. OpenResty 的现状、趋势、使用及学习方法[EB/OL]. [2019-01-22].
- [24] 熊军军, 彭晓刚, 郑晓勇,宣军法. 基于“Nginx+Lua”组件的应用系统灰度发布[J]. 金融电子化.2020,(05).
- [25] 程君. 基于 Lua 的移动互联网中间件系统的研究与实现[D]. 南京 :东南大学, 2017.
- [26] 阚远志. 命名数据网络中缓存一致性问题研究[D]. 合肥 :中国科学技术大学,2020.
- [27] 姬晓涛,刘建华.基于 CAP 理论的区块链共识机制的分析[J].计算机与数字工程,2020,48(12):30 08-3011+3026.
- [28] 彭亮, 胡忠顺, 邬来军. ITXC:一种基于事务型消息的分布式事务方案[J]. Telecommunications Techlogy,2017:5.
- [29] 赵庶林. 基于缓存一致性的高可靠性扩展研究[D]. 山东 :山东大学,2016.
- [30] 李国杰. 大数据研究的科学价值[J]. 中国计算机学会通讯,2021,8(9):8-15.
- [31] 李强. 消息中间件的生产端一致性设计与实战[J]. 电脑编程技巧与维护,2021,(07):67-71.
- [32] 李强,陈登峰. 改进 MD5 加密算法在系统密码存储中的研究及应用[J]. 信息记录材料,2021,22(1 0):157-159.
- [33] 吴莹,林聪. 消息队列技术在车联网综合信息平台系统的应用[J]. 大众标准化,2022(01):37-3 9.
- [34] 纪昌锋. 基于消息队列的多中心业务实例缓存同步系统设计研究[J]. 光源与照明,2021(10):45-47.
- [35] Qiao Tian,Jingmei Li,ShuoZhao,Fangyuan Zheng,Jiaxiang Wang. A Cache Consistency Proto col with Improved Architecture[J]. International Journal of Performability Engineering,2018,1 4(1).26-27.
- [36] 熊涵. 基于 Redis 分布式消息队列的报文过滤系统的设计与实现[D]. 成都: 电子科技大学,2021.
- [37] 徐树. 基于 Jmeter 对 Node 框架性能的测试研究[J]. 电子技术与软件工程,2018,(11):51-52.
- [38] 边耐政,赵东旭.基于 Jmeter 自动化测试集成框架[J]. 计算机应用与软件,2016,33(05):8-12.
- [39] 陈志勇. 全栈性能测试修炼宝典 Jmeter 实战[M]. 北京人民邮电出版社,2016.
- [40] 唐承玲,王虎,李光平,唐春蓬. 基于 Jmeter 的 Web 的性能测试研究 [J]. 电脑与电信 , 2021 (06):65-68.

致谢

三年研究生的学习生活转瞬即逝，随着毕业论文工作的完成我也即将告别自己的校园时光。在这离别的时刻，我的心中满怀感激之情。正是在各位老师、同学、朋友与亲人的关怀与帮助下，才使我取得了巨大的进步。

首先我最感谢的是我的校内导师赵明英老师以及校外导师王双平老师。在两位老师平时的言传身教之中，我不仅学到了丰富的理论知识，更提升了解决问题的能力，这都将会成为我今后工作与生活中的宝贵财富。在这三年的学习中，老师总是耐心的指导我们，为我们默默付出了许多。在最后的毕业论文中，老师也反复帮我审阅修改，给我提出了许多重要的建议与指导。老师严谨的学术态度与高尚的品格都给我留下了深刻的印象。

我还要感谢同级别的武林、郭雨佳、李斌和朱俊杰这几位同学。作为同届的学生，我们总是互相帮助，共同进步。同时要特别感谢高榕同学在三年的时间中对我的帮助和鼓励。

最后我要感谢我的家人，感谢他们多年来对我始终如一的支持与帮助。我今天所取得的所有成绩都离不开他们的辛苦与付出！

作者简介

1. 基本情况

李宜稼，男，陕西榆林人，1996年11月出生，西安电子科技大学机电工程学院控制工程专业2019级硕士研究生。

2. 教育背景

2015.09-2019.06 宝鸡文理学院，本科，专业：机械电子工程

2019.09-2022.06 西安电子科技大学，硕士研究生，专业：控制工程

3. 攻读硕士学位期间的研究成果

3.1 参与科研项目及获奖

- [1] 企、事业委托项目，智能门锁控制系统设计，2020.06~2020.09，已基本完成，
项目负责人：在之前项目的基础上，添加相关的附件，同时加入算法，实现
智能门锁控制系统的加密。
- [2] 企、事业委托项目，智能门锁前端 ECharts 展示图表的制作，2020.11~2021.4
已基本完成，制作用于展示智能门锁销售情况的前端可视图表，调整样式以
及动态显示。



西安电子科技大学
XIDIAN UNIVERSITY

地址：西安市太白南路2号

邮编：710071

网址：www.xidian.edu.cn