
当代数据管理系统
作业 2: 书店 (Bookstore)
实验报告

龚敬洋 10195501436
段晗祈 10195501404
刘明熹 10194500031

指导教师: 周烜
完成日期: 2021/12/18

Contents

1 实验要求	4
1.1 功能	4
1.2 要求	4
2 项目实现	5
2.1 系统构架	5
2.2 数据库设计	6
2.2.1 Entity-Relation 图	6
2.2.2 关系数据库	6
2.2.3 文档数据库	9
2.2.4 键值对数据库	11
2.2.5 冗余	11
2.2.6 事务处理	11
2.2.7 数据库选型	12
2.2.8 数据库调优	14
2.3 接口设计	15
2.3.1 基础接口	15
2.3.2 额外接口	21
2.4 前端实现	37
2.4.1 前端技术介绍	37
2.4.2 前端结构设计	38
2.4.3 前端接口设计	39
2.4.4 开发流程	40
2.4.5 页面展示	41
2.4.6 特点	46
2.4.7 后续开发方向	46
2.5 后端实现	46
2.5.1 Springboot 服务构架	46
2.5.2 JWT 与登录验证	48
2.5.3 ORM 框架的使用	51
2.5.4 用户服务	52
2.5.5 买家服务	53
2.5.6 卖家服务	54
2.5.7 搜索服务	55
2.5.8 ElasticSearch 与全文搜索	56
2.6 高可用	60
2.6.1 Redis 与缓存数据库	61
2.6.2 布隆过滤器与缓存防穿透	64
2.6.3 Caffeine 与多级缓存	66
2.6.4 Kafka 与消息队列	68
2.6.5 Nginx 与负载均衡	73
3 项目部署	75

4 项目测试	76
4.1 接口测试	76
4.1.1 单元测试规范	76
4.1.2 JUnit5	76
4.1.3 基础接口测试代码移植	76
4.1.4 基础接口测试	77
4.1.5 额外接口测试	78
4.2 代码覆盖率	79
4.3 吞吐量测试	80
5 开发说明	83
5.1 版本控制	83
5.2 项目分工	84
6 后记	84

1 实验要求

1.1 功能

实现一个提供网上购书功能的网站后端。网站支持书商在上面开商店，购买者可能通过网站购买。买家和卖家都可以注册自己的账号。一个卖家可以开一个或多个网上商店，买家可以为自己的账户充值，在任意商店购买图书。支持下单 → 付款 → 发货 → 收货的流程。

1. 实现对应接口的功能，见 doc 下面的.md 文件描述 (60% 分数)

- (a) 用户权限接口，如注册、登录、登出、注销
- (b) 买家用户接口，如充值、下单、付款
- (c) 卖家用户接口，如创建店铺、填加书籍信息及描述、增加库存

通过对对应的功能测试，所有 test case 都 pass。测试下单及付款两个接口的性能（最好分离负载生成和后端），测出支持的每分钟交易数，延迟等

2. 为项目添加其它功能：(40% 分数)

- (a) 实现后续的流程
发货 → 收货

(b) 搜索图书

用户可以通过关键字搜索，参数化的搜索方式；如搜索范围包括，题目，标签，目录，内容；全站搜索或是当前店铺搜索。如果显示结果较大，需要分页（使用全文索引优化查找）

(c) 订单状态，订单查询和取消订单。

用户可以查自己的历史订单，用户也可以取消订单。

取消订单（可选项，加分 +5 ~ 10），买家主动地取消订单，如果买家下单经过一段时间超时后，如果买家未付款，订单也会自动取消。

1.2 要求

1. 允许向接口中增加或修改参数，允许修改 HTTP 方法，允许增加新的测试接口，请尽量不要修改现有接口的 url 或删除现有接口，请根据设计合理的拓展接口（加分项 +2 ~ 5 分）。测试程序如果有问题可以提 bug（加分项，每提 1 个 BUG +2，提 1 个 Pull Request +5）。
2. 核心数据使用关系型数据库（PostgreSQL 或 MySQL 数据库）。blob 数据（如图片和大段的文字描述）可以分离出来存其它 NoSQL 数据库或文件系统。
3. 对所有的接口都要写 test case，通过测试并计算代码覆盖率（有较高的覆盖率是加分项 +2 ~ 5）。
4. 尽量使用正确的软件工程方法及工具，如：版本控制，测试驱动开发（利用版本控制是加分项 +2 ~ 5）。
5. 后端使用技术，实现语言不限；不要复制这个项目上的后端代码（不是正确的实践，减分项 -2 ~ 5）。
6. 不需要实现页面。
7. 最后评估分数时考虑以下要素：

- (a) 实现完整度, 全部测试通过, 效率合理
 - (b) 正确地使用数据库和设计分析工具, ER 图, 从 ER 图导出关系模式, 规范化, 事务处理, 索引等
 - (c) 其它 …
8. 3 个人一组, 做好分工, 量化每个人的贡献度

2 项目实现

2.1 系统构架

作为一个经典的电商平台项目, 我们采用了前后端分离和微服务的构架对整个系统进行了设计。整个系统的构架如下图所示 (Figure 1)。其中, Nginx 支撑起一个前端维生容器, 并向用户提供一个的由 Vue.js 搭建的前端服务 (正向代理)。根据网页中的 UI 元素, 用户可以根据通过配置好的请求方式向后端的对应接口发送 Http 请求 (Axios)。请求会首先经过网关, 并发送到一个同样由 Nginx 维护的负载均衡服务器上 (反向代理)。该服务器维护了当前系统内所有可用的服务节点, 并根据系统负载将所有的请求均摊到对应的服务上。在服务运行的过程中一共涉及到 4 个数据库: **PostgreSQL 主数据库**、**MongoDB 扩展数据库**、**Redis 缓存数据库** 和 **ElasticSearch 搜索服务数据库**, 这些数据库通过分离部署的方式被所有服务容器所共享。此外, 一个由 Zookeeper 维护着的分布式 Kafka 消息队列连接着所有的组件, 为所有微服务和数据库间提供消息同步的机制。

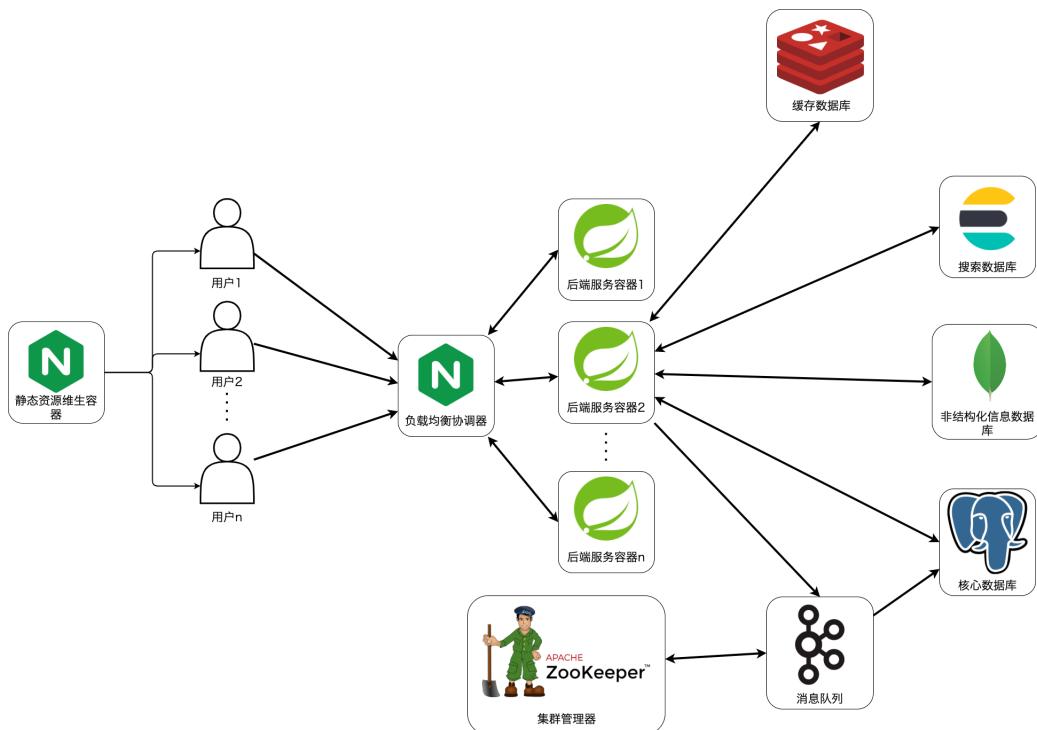


Figure 1: 书店系统的系统构架图

2.2 数据库设计

2.2.1 Entity-Relation 图

本项目的 Entity-Relation 图如下 (Figure 2)。书店主要包含用户、商店、商品、订单和购物车五个实体。其中用户和商店构成经营的关系，用户和购物车构成操作的关系，用户与订单依据买家和卖家的角色构成下单和处理的关系；商品和商店构成出售的关系，商品和购物车构成包含的关系，商品和订单构成组成的关系。

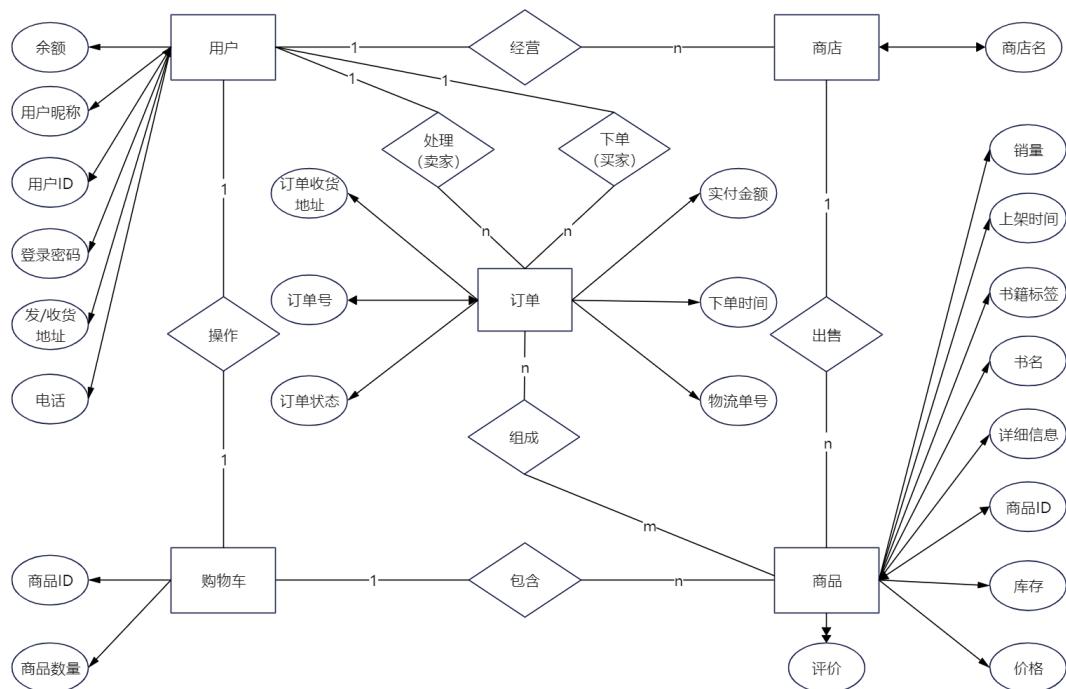


Figure 2: 书店系统的 ER 图

2.2.2 关系数据库

本项目的大部分数据存储在关系数据库中，一共包含 6 张表。

2.2.2.1 用户表 (user)

user 表存储用户的基本信息，包括用户名（name）、地址（address）、余额（money）、昵称（nickname）、密码（password）和手机号（phone）。

以用户名为主键，不额外设置自增主键，简化表结构，易于维护。

```
bookstore=# \d "user"
          数据表 "public.user"
  栏位 |      类型      | 校对规则 | 可空的 | 预设
-----+-----+-----+-----+-----+
  name | character varying(255) |          | not null |
address | character varying(255) |          |          |
  money | integer |          | not null |
nickname | character varying(255) |          |          |
  password | character varying(255) |          | not null |
  phone | character varying(255) |          |          |
索引:
  "user_pkey" PRIMARY KEY, btree (name)
```

Figure 3: user 表结构

2.2.2.2 商品表 (book)

book 表存储商品信息，包括商品 id(id)、详细信息的 id(info_id)、价格 (price)、销量 (sale)、库存 (stock_level)、商店名 (store_name) 和上架时间戳 (time)。

以自增 id 为主键，唯一确定一件商品，同时 store_name 与 info_id 的组合也可以唯一确定一件商品，在 store_name 与 info_id 上建立联合索引，该索引和主键索引可以分别用于不同的场景以提高查询效率。在 sale、price 和 time 上分别建立普通索引，用于提高带有对销量、价格或者时间字段排序的查询的效率。

```
bookstore=# \d "book"
          数据表 "public.book"
  栏位 |      类型      | 校对规则 | 可空的 | 预设
-----+-----+-----+-----+-----+
  id | bigint |          | not null | generated by default as identity
info_id | character varying(255) |          | not null |
  price | integer |          | not null |
  sale | integer |          | not null |
stock_level | integer |          | not null |
store_name | character varying(255) |          | not null |
  time | bigint |          | not null |
索引:
  "book_pkey" PRIMARY KEY, btree (id)
  "book_info_and_store" btree (info_id, store_name)
  "book_price" btree (price)
  "book_sale" btree (sale)
  "book_time" btree ("time")
```

Figure 4: book 表结构

2.2.2.3 商店表 (store)

store 表存储商店信息，包括商店名 (name)、销量 (sale) 和卖家名 (seller_name)。

以商店名为主键，同时在 sale 和 seller_name 上分别建立普通索引，以提高相关查询的效率。

```
bookstore=# \d "store"
          数据表 "public.store"
  栏位      |      类型      | 校对规则 | 可空的 | 预设
-----+-----+-----+-----+-----+
  name   | character varying(255) |      | not null |
  sale    | integer                |      | not null |
  seller_name | character varying(255) |      | not null |
索引:
  "store_pkey" PRIMARY KEY, btree (name)
  "store_sale" btree (sale)
  "store_seller" btree (seller_name)
```

Figure 5: store 表结构

2.2.2.4 订单表 (order)

order 表存储订单信息,包括订单号 (uuid,使用雪花算法生成)、买家用户名 (buyer_name)、发货地址 (from_address)、物流单号 (logistic_id)、价格 (price)、状态 (status)、商店名 (store_name)、下单时间戳 (time) 和收货地址 (to_address)。

以 uuid 为主键,同时在 buyer_name 和 store_name 分别建立普通索引,以提高相关查询的效率。在 time 上建立普通索引,提高需要对时间排序的查询的效率。

```
bookstore=# \d "order"
          数据表 "public.order"
  栏位      |      类型      | 校对规则 | 可空的 | 预设
-----+-----+-----+-----+-----+
  uuid   | character varying(255) |      | not null |
  buyer_name | character varying(255) |      | not null |
  from_address | character varying(255) |      |      |
  logistic_id | character varying(255) |      |      |
  price    | integer                |      | not null |
  status    | integer                |      | not null |
  store_name | character varying(255) |      | not null |
  time      | bigint                 |      | not null |
  to_address | character varying(255) |      |      |
索引:
  "order_pkey" PRIMARY KEY, btree (uuid)
  "order_buyer" btree (buyer_name)
  "order_store" btree (store_name)
  "order_time" btree ("time")
```

Figure 6: order 表结构

2.2.2.5 订单商品表 (order_book)

order_book 表存储订单里的商品信息,order 与 order_book 是一对多的关系。order_book 表包括订单 id(order_id)、图书详情 id(book_info_id)、数量 (count) 和单价 (single_price)。

以 order_id 和 book_info_id 作为联合主键，唯一确定某一订单里的某件商品。

```
bookstore=# \d "order_book"
      数据表 "public.order_book"
栏位 | 类型 | 校对规则 | 可空的 | 预设
-----+-----+-----+-----+-----+
order_id | character varying(255) | not null |
book_info_id | character varying(255) | not null |
count | integer | not null |
single_price | integer | not null |
索引:
"order_book_pkey" PRIMARY KEY, btree (order_id, book_info_id)
```

Figure 7: order_book 表结构

2.2.2.6 购物车商品表 (shopping_cart_book)

shopping_cart_book 表存储用户购物车里的商品信息，这里不单独设置购物车表，因为用户和购物车存在一对多的关系，购物车与购物车中的商品存在一对多的关系，因此可以把用户名作为购物车商品表的一个属性，这样即可确定某一商品是在哪个用户购物车里的。shopping_cart_book 表包括用户名 (buyer_name)、商品 id(book_id)、数量 (count)、单价 (single_price)、商店名 (store_name)。

以 buyer_name 和 book_id 作为联合主键，唯一确定某一用户购物车里的某件商品。

```
bookstore=# \d "shopping_cart_book"
      数据表 "public.shopping_cart_book"
栏位 | 类型 | 校对规则 | 可空的 | 预设
-----+-----+-----+-----+-----+
buyer_name | character varying(255) | not null |
book_id | bigint | not null |
count | integer | not null |
single_price | integer | not null |
store_name | character varying(255) | not null |
索引:
"shopping_cart_book_pkey" PRIMARY KEY, btree (buyer_name, book_id)
```

Figure 8: shopping_cart_book 表结构

2.2.3 文档数据库

本项目中，文档数据库主要用于存储大段的文字信息和图片信息。

2.2.3.1 图书详细信息集合 (book_info)

book_info 集合存储图书的详细信息，所有对该集合的访问都通过 id 来进行，因此除了 id 之外不再单独构建索引。

```
1  {
2      id,          // string 图书ID
3      title,       // string 书籍题目
4      author,      // string 作者
5      publisher,   // string 出版社
6      original_title, // string 原书题目
7      translator,  // string 译者
8      pub_year,    // string 出版年月
9      pages,       // int 页数
10     price,       // int 价格(以分为单位)
11     binding,     // string 装帧/精状/平装
12     isbn,        // string ISBN号
13     author_intro, // string 作者简介
14     book_intro,   // string 书籍简介
15     content,     // string 样章试读
16     tags: [       // array 标签
17         "tags1",
18         "tags2",
19         "tags3",
20         ...
21     ],
22     pictures: [   // array 照片
23         "$Base 64 encoded bytes array1$",
24         "$Base 64 encoded bytes array2$",
25         "$Base 64 encoded bytes array3$",
26         ...
27     ]
28 }
```

Figure 9: book_info 文档结构

2.2.3.2 图书详细信息索引集合 (book_info_index)

这一部分存储在 Elastic Search 数据库中，关于 Elastic Search，详见本文的[2.5.8](#)节。

2.2.3.3 评论集合 (comment)

comment 集合存储商品的评价信息，所有对该集合的访问都通过 id 来进行，因此除了 id 之外不再单独构建索引。

```

1  {
2      id,                      // long    商品id
3      count:[...],            // array   长度是5的数组, 表示从1-5星的数量
4      comments:[
5          {
6              user_id,        // string  用户名
7              star,           // int    星级(1-5)
8              text,           // string  文本评价
9              time            // long   评论时间(时间戳)
10         },
11         ...
12     ]
13 }

```

Figure 10: comment 文档结构

2.2.4 键值对数据库

本项目采用 redis 这一键值对缓存数据库, 将一部分查询频率较高的数据存储到 redis 中, 以提升查询效率。关于 redis, 详见本文的2.6.1节。

2.2.5 冗余

在本项目中, redis 可以看作是关系数据库中某些数据的冗余, 当一个查询到来时, 如果在 redis 的缓存中命中, 便可不用访问关系数据库, 直接返回 redis 缓存中的结果, 以此来提高查询效率。

Elastic Search 中存储的图书详细信息索引集合可以看作是 mongDB 中图书详细信息集合的冗余, 这里采用冗余是因为 Elastic Search 能够更好地进行搜索操作, 提高搜索效率。

从 2.2.2 节可以看出, 在关系数据库内部的表之间也存在一些冗余信息, 这些冗余都是为了提高查询效率。

2.2.6 事务处理

本项目对某些操作采用了“事务处理”, 如下单操作和付款操作等, 以保证数据的一致性和完整性。

在 SpringBoot 中, 我们可以在方法上添加 @Transactional 注解来表明该方法为需要进行“事务处理”的方法。当外部调用带有 @Transactional 注解的方法时, 其访问数据库的过程会当作一个事务来完成。当在方法执行过程中抛出 RunTimeException 异常时, 会触发事务回滚 (Rollback), 以保证数据的一致性, 具体代码如下。

```

1 /* BuyerServiceImpl.java */
2 @Override
3 @Transactional // 事务注解
4 public ResponseEntity<OrderMessage> newOrder(NewOrderBody newOrderBody) {
5     String buyerName = newOrderBody.getBuyerName();
6     String storeName = newOrderBody.getStoreName();
7     //Irrelevant codes
8 }
9
10 @Override

```

```
11  @Transactional // 事务注解
12  public ResponseEntity<Message> payment(PayBody payBody) {
13      String md5Password = DigestUtils.md5DigestAsHex(payBody.getPassword() .
14          getBytes());
15      String userRedisKey = "user_" + payBody.getBuyerName();
16      //Irrelevant codes
17  }
```

2.2.7 数据库选型

本实验要求我们使用关系型数据库作为系统的核心数据库。现阶段，市面上两大主流的传统关系型数据库即为 **MySQL** 和 **PostgreSQL**（SQL Server 主要服务于 Windows 及其衍生系统，而本项目设计的目标系统为 Linux 相关系统，故不作考虑）。下面我们来对这两大关系型数据库进行分析，以确定在该项目中所要选用的核心数据库。

作为 Oracle 旗下的老牌关系型数据库，MySQL 已经在无数的应用上发挥了其威力。然而，作为一个以商用为主的软件，MySQL 分为免费的 **Community**（社区版）和收费的 **Enterprise**（商业版），这就导致了其社区版本在许多功能上都受到了限制。此外，尽管 MySQL 的社区版本开放了源代码，但其使用的 **GPL 协议**使得二次开发受到了许多的限制。相比之下，PostgreSQL 是一款完全由社区（PostgreSQL Global Development Group）维护的开源关系型数据库，这就意味着其开发和改进相比 MySQL 更为方便和敏捷。其使用的 **类 BSD/MIT 协议**也使得二次开发变得容易。

PostgreSQL 的宣传标语就是“世界上最先进的开源关系型数据库”，这也意味着其拥有着更新的特性。从功能上来说，PostgreSQL 相比 MySQL 增加了大量的数据类型，如 Array 和 JSON，且在部分原始数据类型上相比 MySQL 也有更小的限制（如 TEXT 类型）。这一特性使得 PostgreSQL 甚至可以作为 NoSQL 来使用。此外，从运维上来说，PostgreSQL 拥有许多实用的特性，如：DDL 操作能够放入事务中、能够并发地创建或删除索引、多样化的复制和提交方式等。这些特性能够帮助我们快速完成对数据库中结构的修改，这对于类似本实验这样的小型项目是十分重要的。

从 DB Engines 网站上我们可以看到近几年几大关系型数据库的流行程度（Figure 11）。可以发现，尽管 MySQL 仍然占据着主导的数据库比例，但其市场占比已经开始有下降的趋势；而 PostgreSQL 作为一款新兴的关系型数据库，其市场占有率自 2014 年起就始终以一个极高的速度增长着，这也充分表明了 PostgreSQL 在实际使用场景下的优势。

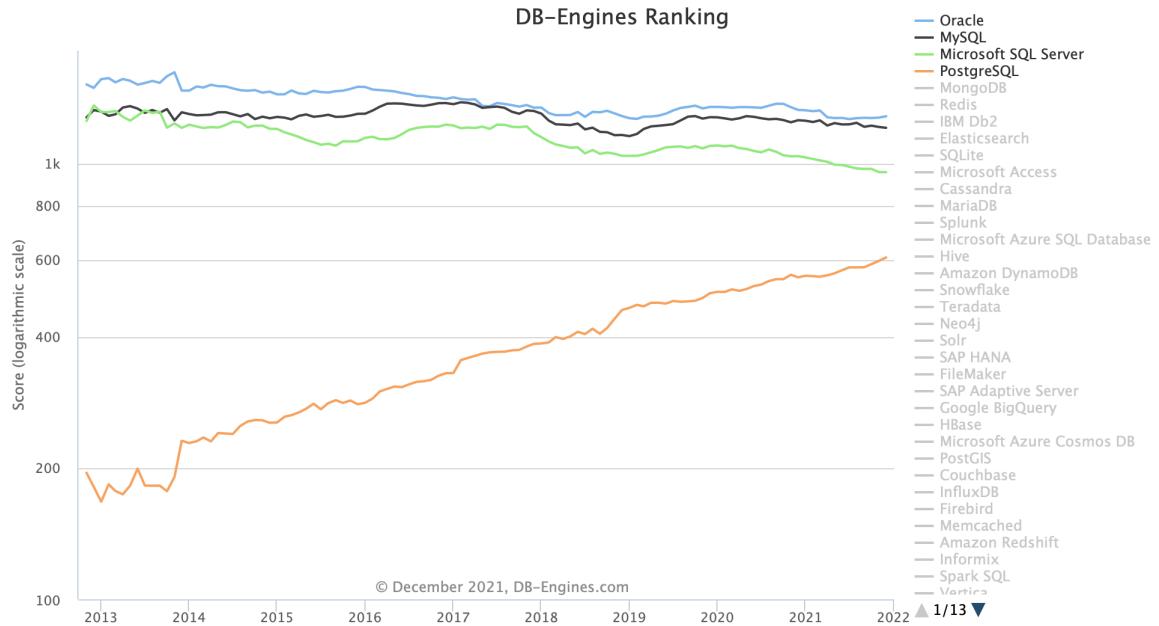


Figure 11: 近年来几大主流关系型数据库的热度趋势变化 [Eng21]

MySQL 使用了 InnoDB 作为其主要存储引擎（新版本下的 MySQL 支持如 MyISAM 等其他存储，但由于其不支持完整的事务处理功能，故不常被使用），尽管自起发明以来已经取得了长足的进步，但仍然有不少案例报告系统在运行过程中出现了服务器级别的数据库丢失现象。相比之下，在实际使用过程中，PostgreSQL 的稳定性更强。从性能上来说，MySQL 使用多线程技术来提升系统性能的利用率，而 PostgreSQL 使用了多进程技术。在现代处理器上，后者相比前者有着更好的优化空间。在实际的使用过程中，PostgreSQL 在面对高并发访问请求时相比 MySQL 也的确拥有着更好的表现。除此之外，当数据库负载逼近极限下，PostgreSQL 的性能指标仍可以维持双曲线甚至对数曲线，到顶峰之后不再下降，而 MySQL 会明显出现一个波峰后下滑。

为了实际测试 MySQL 和 PostgreSQL 的性能表现，我们使用 **Sysbench** 基准测试工具分别对同一台部署了 MySQL 和 PostgreSQL 的服务器进行了性能测试，如下图所示（Figure 12）。测试结果表明，同等配置下，PostgreSQL 的综合性能表现达到了 MySQL 的近两倍。

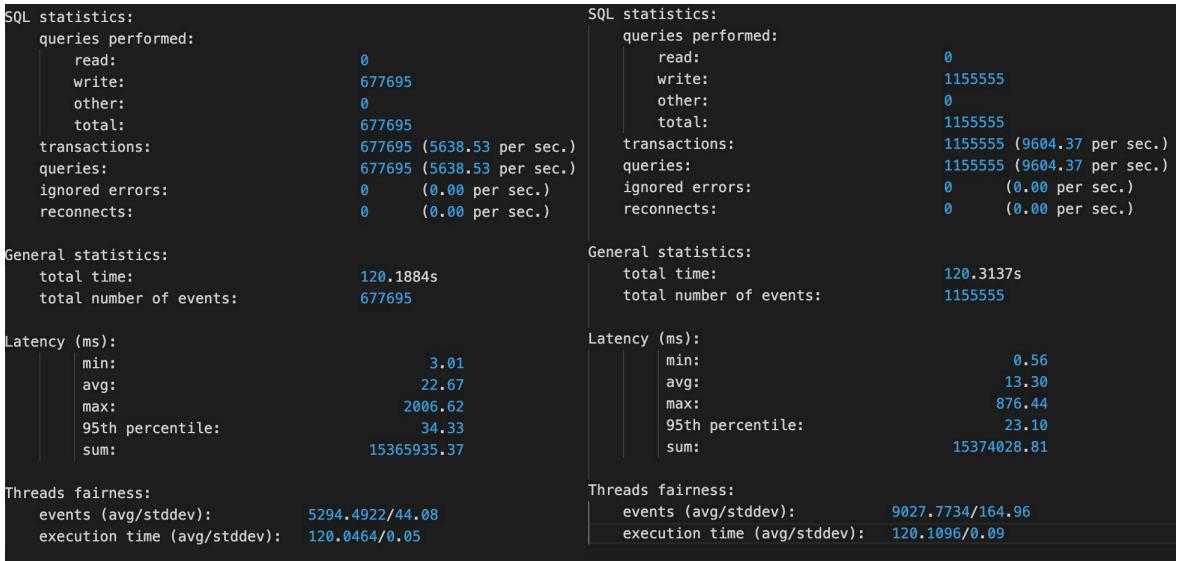


Figure 12: 阿里云同配置下 MySQL 和 PostgreSQL 性能比较 (左为 MySQL, 右为 PostgreSQL)

综合上面的结果，在本项目中，我们使用了 **PostgreSQL** 作为系统的核心数据库。

除此之外，在书店系统中，一本书本商品通常还会带有大量的 **BLOB** 数据，如图书简介、作者简介、图书照片等。若直接这些数据放在关系型数据库中，会导致一个字段过长，且不方便存储和扩展。对于这些数据，**文档型数据库 (Document Database)** 是一种极好的存储方式。在本应用中，我们使用 **MongoDB** 来作为系统的扩展数据存储数据库。

2.2.8 数据库调优

除了前文中所讨论的如索引、冗余、事务处理等数据访问优化，对于数据库本身而言，其当前配置也是一个需要考虑的问题。通常来说，一个数据库的默认配置是较为保守的，在很多场景下无法完全利用系统中的资源。为了提升设备利用效率，我们需要对数据库的相关配置进行调优。由于 MongoDB 等 NoSQL 本身已经具有了极强的性能分配机制，因此我们主要针对传统关系型数据库进行调优。

对于 PostgreSQL 数据库，我们使用如下配置对参数进行了优化 (Table 1)

参数名	默认值	参数说明	优化值
max_connections	100	允许客户端连接的最大数目	10000
fsync	on	强制把数据同步更新到磁盘	off
shared_buffers	24MB	PostgreSQL 能够用于缓存数据的内存大小	1024MB
work_mem	1MB	内部排序和一些复杂的查询在这个 Buffer 中完成	10MB
wal_buffer	768KB	日志缓存区的大小	4096KB
checkpoint_segments	3	设置 Wal Log 的最大数量 (一个 Log 的大小为 16M)	10
commit_delay	0	事务提交后，日志写到 Wal Log 上到 wal_buffer	2
commit_siblings	5	写入到磁盘的时间间隔	20

Table 1: PostgreSQL 的参数优化

2.3 接口设计

本实验共设计并实现了 33 个接口，其中包括 11 个基础接口以及 22 个额外接口。额外接口一部分用于给前端提供必要的展示数据，对数量较大的数据进行了分页处理，另一部分则用于用户信息修改、购物车、订单的后续处理（发货、收货、评价及取消）、商品搜索和商品评论等功能。

2.3.1 基础接口

2.3.1.1 用户注册

URL: [http://\[address\]/auth/register](http://[address]/auth/register)

Method: POST

Request Body

变量名	类型	描述	是否可空
user_id	string	用户名	N
password	string	密码	N

Response Status Code

状态码	描述
200	ok
512	注册失败，用户名已存在

Response Body

变量名	类型	描述	是否可空
message	string	错误消息，成功时为“ok”	N

2.3.1.2 用户注销

URL: [http://\[address\]/auth/unregister](http://[address]/auth/unregister)

Method: POST

Request Body

变量名	类型	描述	是否可空
user_id	string	用户名	N
password	string	密码	N

Response Status Code

状态码	描述
200	ok
401	注销失败，用户名不存在或密码错误

Response Body

变量名	类型	描述	是否可空
message	string	错误消息，成功时为“ok”	N

2.3.1.3 用户登录

URL: [http://\[address\]/auth/login](http://[address]/auth/login)

Method: POST

Request Body

变量名	类型	描述	是否可空
user_id	string	用户名	N
password	string	密码	N
terminal	string	终端代码	N

Response Status Code

状态码	描述
200	ok
401	登录失败，用户名不存在或密码错误

Response Body

变量名	类型	描述	是否可空
message	string	错误消息，成功时为“ok”	N
token	string	访问 token	成功时不为空

2.3.1.4 用户修改密码

URL: [http://\[address\]/auth/password](http://[address]/auth/password)

Method: POST

Request Body

变量名	类型	描述	是否可空
user_id	string	用户名	N
old_password	string	旧登录密码	N
new_password	string	新登录密码	N

Response Status Code

状态码	描述
200	ok
401	更改密码失败

Response Body

变量名	类型	描述	是否可空
message	string	错误消息，成功时为“ok”	N

2.3.1.5 用户登出

URL: [http://\[address\]/auth/logout](http://[address]/auth/logout)

Method: POST

Request Headers

key	类型	描述
token	string	访问 token

Request Body

变量名	类型	描述	是否可空
user_id	string	用户名	N

Response Status Code

状态码	描述
200	ok
401	登出失败，用户名或 token 错误

Response Body

变量名	类型	描述	是否可空
message	string	错误消息，成功时为“ok”	N

2.3.1.6 卖家创建商铺

URL: [http://\[address\]/seller/create_store](http://[address]/seller/create_store)

Method: POST

Request Headers

key	类型	描述
token	string	访问 token

Request Body

变量名	类型	描述	是否可空
user_id	string	卖家用户 ID	N
store_id	string	商铺 ID	N

Response Status Code

状态码	描述
200	ok
514	创建失败，商铺 ID 已存在

Response Body

变量名	类型	描述	是否可空
message	string	错误消息，成功时为“ok”	N

2.3.1.7 卖家添加书籍信息

URL: [http://\[address\]/seller/add_book](http://[address]/seller/add_book)

Method: POST

Request Headers

key	类型	描述
token	string	访问 token

Request Body

变量名	类型	描述	是否可空
user_id	string	卖家用户 ID	N
store_id	string	商铺 ID	N
book_info	object	书籍信息	N
stock_level	int	初始库存	N

book_info

变量名	类型	描述	是否可空
id	string	图书 ID	N
title	string	题目	N
author	string	作者	Y
publisher	string	出版社	Y
original_title	string	原书题目	Y
translator	string	译者	Y
pub_year	string	出版年月	Y
pages	int	页数	Y
price	int	价格 (以分为单位)	N
binding	string	装帧/精状/平装	Y
isbn	string	ISBN 号	Y
author_intro	string	作者简介	Y
book_intro	string	书籍简介	Y
content	string	样章试读	Y
tags	array	标签	Y
pictures	array	照片	Y

Response Status Code

状态码	描述
200	ok
511	添加失败, 卖家用户 ID 不存在
513	添加失败, 商铺 ID 不存在
516	添加失败, 图书 ID 已存在

Response Body

变量名	类型	描述	是否可空
message	string	错误消息, 成功时为 “ok”	N

2.3.1.8 卖家添加书籍库存

URL: [http://\[address\]/seller/add_stock_level](http://[address]/seller/add_stock_level)

Method: POST

Request Headers

key	类型	描述
token	string	访问 token

Request Body

变量名	类型	描述	是否可空
user_id	string	卖家用户 ID	N
store_id	string	商铺 ID	N
book_info_id	string	书籍 ID	N
add_stock_level	int	增加的库存量	N

Response Status Code

状态码	描述
200	ok
513	添加失败, 商铺 ID 不存在
515	添加失败, 图书 ID 不存在
520	添加失败, 卖家用户 ID 错误

Response Body

变量名	类型	描述	是否可空
message	string	错误消息, 成功时为“ok”	N

2.3.1.9 买家下单

URL: [http://\[address\]/buyer/new_order](http://[address]/buyer/new_order)

Method: POST

Request Headers

key	类型	描述
token	string	访问 token

Request Body

变量名	类型	描述	是否可空
user_id	string	买家用户 ID	N
store_id	string	商铺 ID	N
books	object array	书籍购买列表	N

book

变量名	类型	描述	是否可空
id	string	书籍 ID	N
count	string	购买数量	N

Response Status Code

状态码	描述
200	ok
511	下单失败, 买家用户 ID 不存在
514	下单失败, 商铺 ID 不存在
515	下单失败, 购买的图书不存在
517	下单失败, 商品库存不足

Response Body

变量名	类型	描述	是否可空
message	string	错误消息, 成功时为 “ok”	N
order_id	string	订单号	成功时不为空

2.3.1.10 买家付款

URL: [http://\[address\]/buyer/payment](http://[address]/buyer/payment)

Method: POST

Request Body

变量名	类型	描述	是否可空
user_id	string	买家用户 ID	N
order_id	string	订单 ID	N
password	string	密码	N

Response Status Code

状态码	描述
200	ok
519	支付失败, 账户余额不足
551	支付失败, 无效参数
401	支付失败, 授权失败

Response Body

变量名	类型	描述	是否可空
message	string	错误消息, 成功时为 “ok”	N

2.3.1.11 买家充值

URL: [http://\[address\]/buyer/add_funds](http://[address]/buyer/add_funds)

Method: POST

Request Body

变量名	类型	描述	是否可空
user_id	string	买家用户 ID	N
password	string	密码	N
add_value	int	充值金额 (以分为单位)	N

Response Status Code

状态码	描述
200	ok
551	充值失败, 无效参数
401	充值失败, 授权失败

Response Body

变量名	类型	描述	是否可空
message	string	错误消息, 成功时为 “ok”	N

2.3.2 额外接口

2.3.2.1 用户获取个人信息

URL: [http://\[address\]/auth/{user_id}](http://[address]/auth/{user_id})

Method: GET

Request Headers

key	类型	描述
token	string	访问 token

Path Variable

变量名	类型	描述	是否可空
user_id	string	用户 ID	N

Response Status Code

状态码	描述
200	ok
511	获取失败, 用户 ID 不存在

Response Body

变量名	类型	描述	是否可空
message	string	错误消息, 成功时为 “ok”	N
user	object	用户信息	成功时不为空

user

变量名	类型	描述	是否可空
user_id	string	用户 ID	N
nick_name	string	用户昵称	N
money	int	用户余额	N
phone	string	手机号	N
address	string	地址	N

2.3.2.2 用户提现

URL: [http://\[address\]/buyer/cash](http://[address]/buyer/cash)

Method: POST

Request Headers

key	类型	描述
token	string	访问 token

Request Body

变量名	类型	描述	是否可空
user_id	string	用户 ID	N
cash	int	提现金额	N

Response Status Code

状态码	描述
200	ok
511	提现失败，用户 ID 不存在
519	提现失败，余额不足

Response Body

变量名	类型	描述	是否可空
message	string	错误消息，成功时为“ok”	N

2.3.2.3 用户修改昵称、手机号、地址

URL: [http://\[address\]/auth](http://[address]/auth)

Method: PATCH

Request Headers

key	类型	描述
token	string	访问 token

Request Body

变量名	类型	描述	是否可空
user_id	string	用户 ID	N
nickname	string	用户昵称	Y
phone	string	手机号	Y
address	string	地址	Y

Response Status Code

状态码	描述
200	ok
511	修改失败，用户 ID 不存在

Response Body

变量名	类型	描述	是否可空
message	string	错误消息，成功时为“ok”	N

2.3.2.4 卖家删除商店

URL: [http://\[address\]/seller/store/{seller_id}/{store_id}](http://[address]/seller/store/{seller_id}/{store_id})

Method: DELETE

Request Headers

key	类型	描述
token	string	访问 token

Path Variable

变量名	类型	描述	是否可空
seller_id	string	用户 ID	N
store_id	string	商店 ID	N

Response Status Code

状态码	描述
200	ok
511	删除失败，商店不存在

Response Body

变量名	类型	描述	是否可空
message	string	错误消息，成功时为“ok”	N

2.3.2.5 卖家获得自己的商店列表

URL: [http://\[address\]/seller/store/{seller_id}](http://[address]/seller/store/{seller_id})

Method: GET

Request Headers

key	类型	描述
token	string	访问 token

Path Variable

变量名	类型	描述	是否可空
user_id	string	用户 ID	N

Response Status Code

状态码	描述
200	ok

Response Body

变量名	类型	描述	是否可空
message	string	错误消息, 成功时为“ok”	N
stores	object array	商店列表	成功时不为空

store

变量名	类型	描述	是否可空
store_id	string	商店 ID	N
sale	int	销量	N

2.3.2.6 卖家下架商品

URL: [http://\[address\]/seller/book/{seller_id}/{book_id}](http://[address]/seller/book/{seller_id}/{book_id})

Method: DELETE

Request Headers

key	类型	描述
token	string	访问 token

Path Variable

变量名	类型	描述	是否可空
seller_id	string	用户 ID	N
book_id	long	商品 ID	N

Response Status Code

状态码	描述
200	ok
516	删除失败, 商品不存在
520	删除失败, 用户 ID 错误

Response Body

变量名	类型	描述	是否可空
message	string	错误消息, 成功时为“ok”	N

2.3.2.7 获得商品详细信息

URL: [http://\[address\]/search/book_info/{book_id}](http://[address]/search/book_info/{book_id})

Method: GET

Request Headers

key	类型	描述
token	string	访问 token

Path Variable

变量名	类型	描述	是否可空
book_id	long	商品 ID	N

Response Status Code

状态码	描述
200	ok
515	获取失败, 商品信息不存在

Response Body

变量名	类型	描述	是否可空
message	string	错误消息, 成功时为“ok”	N
book_info	object	商品详细信息	成功时不为空

book_info

变量名	类型	描述	是否可空
id	string	图书 ID	N
title	string	题目	N
author	string	作者	Y
publisher	string	出版社	Y
original_title	string	原书题目	Y
translator	string	译者	Y
pub_year	string	出版年月	Y
pages	int	页数	Y
price	int	价格 (以分为单位)	N
binding	string	装帧/精状/平装	Y
isbn	string	ISBN 号	Y
author_intro	string	作者简介	Y
book_intro	string	书籍简介	Y
content	string	样章试读	Y
tags	array	标签	Y
pictures	array	照片	Y

2.3.2.8 买家获得购物车列表

URL: [http://\[address\]/buyer/cart/{user_id}](http://[address]/buyer/cart/{user_id})

Method: GET

Request Headers

key	类型	描述
token	string	访问 token

Path Variable

变量名	类型	描述	是否可空
user_id	string	用户 ID	N

Response Status Code

状态码	描述
200	ok

Response Body

变量名	类型	描述	是否可空
message	string	错误消息, 成功时为“ok”	N
items	object array	购物车中的商品列表	成功时不为空

item

变量名	类型	描述	是否可空
book_id	long	商品 ID	N
store_id	string	商店 ID	N
book_info	object	商品详情	N
single_price	int	单价	N
count	int	数量	N

book_info

变量名	类型	描述	是否可空
id	string	图书 ID	N
title	string	题目	N
author	string	作者	Y
picture	string	图片	Y

2.3.2.9 买家把商品加入到购物车

URL: [http://\[address\]/buyer/cart](http://[address]/buyer/cart)

Method: POST

Request Headers

key	类型	描述
token	string	访问 token

Request Body

变量名	类型	描述	是否可空
buyer_id	string	用户 ID	N
book_id	long	商品 ID	N
count	int	数量	N

Response Status Code

状态码	描述
200	ok
515	添加失败，商品不存在

Response Body

变量名	类型	描述	是否可空
message	string	错误消息，成功时为“ok”	N

2.3.2.10 买家修改购物车中商品的数量

URL: [http://\[address\]/buyer/cart](http://[address]/buyer/cart)

Method: PATCH

Request Headers

key	类型	描述
token	string	访问 token

Request Body

变量名	类型	描述	是否可空
buyer_id	string	用户 ID	N
book_id	long	商品 ID	N
count	int	数量	N

Response Status Code

状态码	描述
200	ok
515	修改失败，商品不存在

Response Body

变量名	类型	描述	是否可空
message	string	错误消息，成功时为“ok”	N

2.3.2.11 买家从购物车下单

URL: [http://\[address\]/buyer/cart_order](http://[address]/buyer/cart_order)

Method: POST

Request Headers

key	类型	描述
token	string	访问 token

Request Body

变量名	类型	描述	是否可空
user_id	string	买家用户 ID	N
store_id	string	商铺 ID	N
books	object array	书籍购买列表	N

book

变量名	类型	描述	是否可空
id	string	书籍 ID	N
count	string	购买数量	N

Response Status Code

状态码	描述
200	ok
511	下单失败，买家用户 ID 不存在
514	下单失败，商铺 ID 不存在
515	下单失败，购买的图书不存在
517	下单失败，商品库存不足

Response Body

变量名	类型	描述	是否可空
message	string	错误消息，成功时为“ok”	N
order_id	string	订单号	成功时不为空

2.3.2.12 分页获取订单列表

URL:

用户作为买家:

[http://\[address\]/buyer/order/{user_id}/{status}/{current_page}/{num_per_page}](http://[address]/buyer/order/{user_id}/{status}/{current_page}/{num_per_page})
用户作为卖家:

[http://\[address\]/seller/order/{user_id}/{status}/{current_page}/{num_per_page}](http://[address]/seller/order/{user_id}/{status}/{current_page}/{num_per_page})

Method: GET

Request Headers

key	类型	描述
token	string	访问 token

Path Variable

变量名	类型	描述	是否可空
user_id	string	用户 ID	N
status	int	订单状态	N
current_page	int	当前页	N
num_per_page	int	每页数量	N

status

status	说明
-1	全部
0	待支付
1	待发货
2	待收货
3	待评价
4	已完成
5	已取消

Response Status Code

状态码	描述
200	ok

Response Body

变量名	类型	描述	是否可空
message	string	错误消息, 成功时为“ok”	N
count	int	数量	N
orders	object array	订单列表	Y

order

变量名	类型	描述	是否可空
uuid	string	订单号	N
buyer_id	string	买家用户 ID	N
store_id	string	商店 ID	N
status	int	订单状态	N
price	int	总价	N
time	long	时间戳	N
from_address	string	发货地址	Y
to_address	string	收货地址	Y
logistic_id	string	物流单号	Y
items	object array	订单商品列表	N

item

变量名	类型	描述	是否可空
book_id	long	商品 ID	N
store_id	string	商店 ID	N
book_info	object	商品详情	N
single_price	int	单价	N
count	int	数量	N

book_info

变量名	类型	描述	是否可空
id	string	图书 ID	N
title	string	题目	N
author	string	作者	Y
picture	string	图片	Y

2.3.2.13 卖家发货

URL: [http://\[address\]/seller/order_send](http://[address]/seller/order_send)

Method: POST

Request Headers

key	类型	描述
token	string	访问 token

Request Body

变量名	类型	描述	是否可空
uuid	string	订单号	N
seller_id	string	卖家用户 ID	N
logistic_id	string	物流单号	N

Response Status Code

状态码	描述
200	ok
518	发货失败, 订单无效
520	发货失败, 用户 ID 错误

Response Body

变量名	类型	描述	是否可空
message	string	错误消息, 成功时为 “ok”	N

2.3.2.14 买家收货

URL: [http://\[address\]/buyer/order_receive](http://[address]/buyer/order_receive)

Method: POST

Request Headers

key	类型	描述
token	string	访问 token

Request Body

变量名	类型	描述	是否可空
uuid	string	订单号	N
buyer_id	string	买家用户 ID	N

Response Status Code

状态码	描述
200	ok
518	收货失败，订单无效

Response Body

变量名	类型	描述	是否可空
message	string	错误消息，成功时为“ok”	N

2.3.2.15 买家评价

URL: [http://\[address\]/buyer/comment](http://[address]/buyer/comment)

Method: POST

Request Headers

key	类型	描述
token	string	访问 token

Request Body

变量名	类型	描述	是否可空
user_id	string	用户 ID	N
uuid	string	订单号	N
books	object array	商品评价列表	N

book

变量名	类型	描述	是否可空
book_id	long	商品 ID	N
comment	object	商品评价	N

comment

变量名	类型	描述	是否可空
star	int	星级	N
text	string	文字评价	Y

Response Status Code

状态码	描述
200	ok
518	评价失败，订单无效

Response Body

变量名	类型	描述	是否可空
message	string	错误消息，成功时为“ok”	N

2.3.2.16 分页获取商品评论

URL: [http://\[address\]/search/comment/{book_id}/{current_page}/{num_per_page}](http://[address]/search/comment/{book_id}/{current_page}/{num_per_page})

Method: GET

Request Headers

key	类型	描述
token	string	访问 token

Path Variable

变量名	类型	描述	是否可空
book_id	long	商品 ID	N
current_page	int	当前页	N
num_per_page	int	每页数量	N

Response Status Code

状态码	描述
200	ok

Response Body

变量名	类型	描述	是否可空
message	string	错误消息，成功时为“ok”	N
count	int	数量	N
content	object	评论	Y

content

变量名	类型	描述	是否可空
book_id	long	商品 ID	N
count	int array	长度为 5，表示从 1-5 星的评论数	N
comments	object array	评论数组	N

comment

变量名	类型	描述	是否可空
user_id	string	用户 ID	N
star	int	星级	N
text	string	文字评价	Y
time	long	时间戳	N

2.3.2.17 获取商品星级

URL: [http://\[address\]/search/star/{book_id}](http://[address]/search/star/{book_id})

Method: GET

Request Headers

key	类型	描述
token	string	访问 token

Path Variable

变量名	类型	描述	是否可空
book_id	long	商品 ID	N

Response Status Code

状态码	描述
200	ok

Response Body

变量名	类型	描述	是否可空
message	string	错误消息, 成功时为“ok”	N
count	int array	长度为 5 的数组, 表示从 1-5 星的评论数	N

2.3.2.18 取消订单

URL: [http://\[address\]/buyer/order_cancel](http://[address]/buyer/order_cancel)

Method: POST

Request Headers

key	类型	描述
token	string	访问 token

Request Body

变量名	类型	描述	是否可空
uuid	string	订单号	N
buyer_id	string	买家用户 ID	N

Response Status Code

状态码	描述
200	ok
518	取消失败，订单无效

Response Body

变量名	类型	描述	是否可空
message	string	错误消息，成功时为“ok”	N

2.3.2.19 查看商店详情

URL: [http://\[address\]/search/store/{store_id}](http://[address]/search/store/{store_id})

Method: GET

Request Headers

key	类型	描述
token	string	访问 token

Path Variable

变量名	类型	描述	是否可空
store_id	string	商店 ID	N

Response Status Code

状态码	描述
200	ok
513	获取失败，商店不存在

Response Body

变量名	类型	描述	是否可空
message	string	错误消息，成功时为“ok”	N
store	object	商店详细信息	成功时不为空

store

变量名	类型	描述	是否可空
store_id	string	商店 ID	N
seller_id	string	卖家用户 ID	N
address	string	地址	N

2.3.2.20 获取商店列表

URL: [http://\[address\]/search/store/{num}](http://[address]/search/store/{num})

Method: GET

Request Headers

key	类型	描述
token	string	访问 token

Path Variable

变量名	类型	描述	是否可空
num	int	数量	N

Response Status Code

状态码	描述
200	ok

Response Body

变量名	类型	描述	是否可空
message	string	错误消息, 成功时为“ok”	N
stores	object array	商店详细信息	成功时不为空

store

变量名	类型	描述	是否可空
store_id	string	商店 ID	N
seller_id	string	卖家用户 ID	N
address	string	地址	N

2.3.2.21 搜索商品, 分页返回

URL: [http://\[address\]/search/book/{store_id}/{current_page}/{num_per_page}/{strategy}?keywords=...](http://[address]/search/book/{store_id}/{current_page}/{num_per_page}/{strategy}?keywords=...)
 Method: GET

Request Headers

key	类型	描述
token	string	访问 token

Path Variable

变量名	类型	描述	是否可空
store_id	string	商店 ID(如果为 0 则表示全局搜索)	N
current_page	int	当前页	N
num_per_page	int	每页商品数量	N
strategy	int	搜索策略	N

strategy

strategy	说明
0	相关性 (相关性高的在前面)
1	时间 (新发布的在前面)
2	销量 (销量高的在前面)
3	价格升序
4	价格降序

Request Param

变量名	类型	描述	是否可空
keywords	string	用户搜索输入	Y

Response Status Code

状态码	描述
200	ok

Response Body

变量名	类型	描述	是否可空
message	string	错误消息, 成功时为 “ok”	N
count	int	数量	N
books	object array	商品列表	成功时不为空

book

变量名	类型	描述	是否可空
id	string	商品 ID	N
price	int	价格	N
stock_level	int	库存	N
store_id	string	商店 ID	N
sale	int	销量	N
time	long	上架时间	N
book_info	object	书籍信息	N

book_info

变量名	类型	描述	是否可空
id	string	图书 ID	N
title	string	题目	N
author	string	作者	Y
picture	string	图片	Y

2.4 前端实现

2.4.1 前端技术介绍

2.4.1.1 Vue.js 3

渐进式 *JavaScript* 框架

- Vue.js 是一套构建用户界面的渐进式框架。
- Vue 只关注视图层，被设计为可以自底向上逐层应用。
- Vue 的目标是通过尽可能简单的 API 实现响应数据的绑定以及视图组件的组合使用。
- 另外，当与现代化的工具链以及各种支持类库结合使用时，Vue 也完全能够为复杂的单页应用提供驱动。

2.4.1.2 Quasar v2

高性能高质量的 *Vue.js 3* 用户界面框架

- Quasar 是一个高性能的 *Vue.js* 前端用户界面设计框架。
- Quasar CLI 模式，对每种构建模式（SPA、SSR、PWA、移动应用程序、桌面应用程序和浏览器扩展）提供了一流的支持，并提供了最佳的开发人员体验，可以高效、灵活地进行高级开发者设计。
- Quasar v2 基于 *Vue3*，提供高性能的响应式前端组件，可以对页面的风格特性、布局网络、静态和动态组件进行高效的设计。
- Quasar 是最注重性能的框架之一。

2.4.1.3 Axios

易用、简洁且高效的 *http* 库

- Axios 是一个基于 promise 的 HTTP 库，可以用在浏览器和 *node.js* 中。
- Axios 从浏览器中创建 XMLHttpRequests，从 *node.js* 创建 http 请求，并且支持 Promise API
- Axios 可以高效地处理或拦截请求和响应，转换请求数据和响应数据。
- 我们将 Axios 进行封装成为 api，简化调用过程、突出功能代码、便捷接口使用

```
1 // Axios请求使用例
2 let url = location.protocol + "://" + ( testurl || location.hostname ) + ":" +
3     backend_port + "/buyer/order_cancel";
4 let body = {
5     "buyer_id": objData.user_id,
6     "uuid": order_id,
7 };
8 api.post(url, body).then((response) => {
9     if(response.status == 200){
10         alert("取消成功！")
```

```
10  }
11 }
12 .catch((error) => {
13   alert(error.response.data.message);
14});
```

2.4.2 前端结构设计

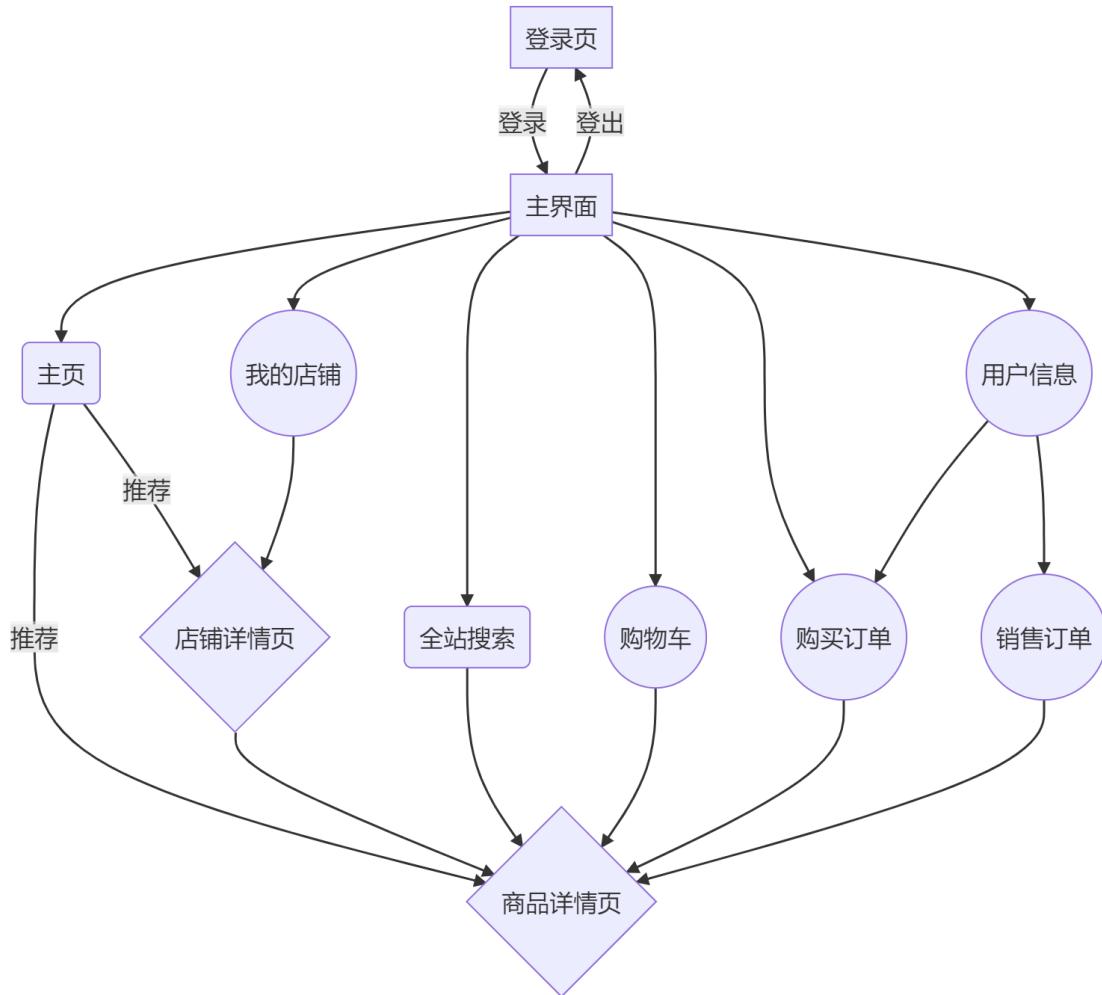


Figure 13: 前端页面结构设计示意图

网页结构以登录页和主界面为主，不同的功能页面在主界面导航栏下显示，提供主页、用户信息、商品搜索页、店铺详情页、商品详情页等页面的跳转接口。

2.4.3 前端接口设计

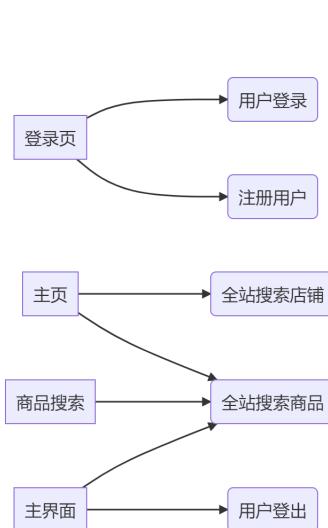


Figure 14: 页面接口示意图 1

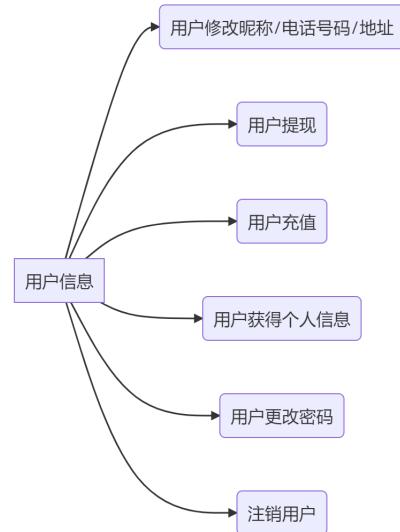


Figure 15: 页面接口示意图 2

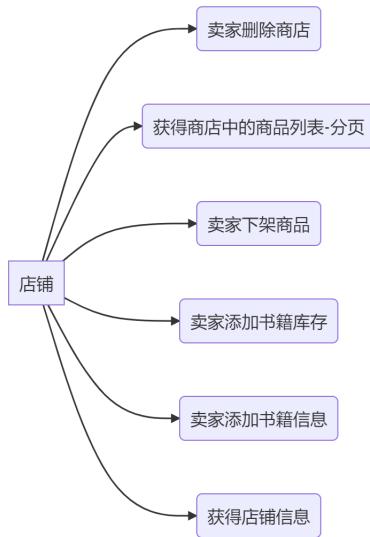


Figure 16: 页面接口示意图 3

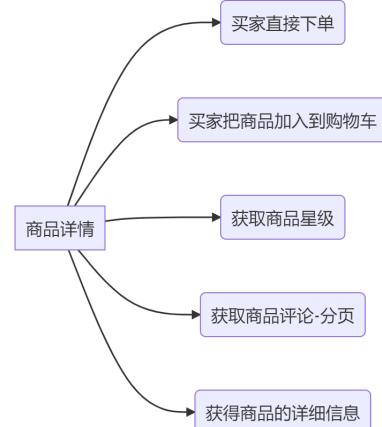


Figure 17: 页面接口示意图 4

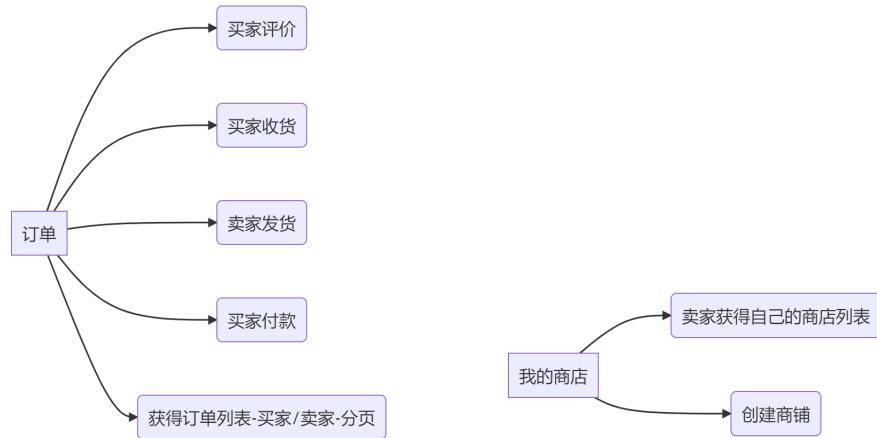
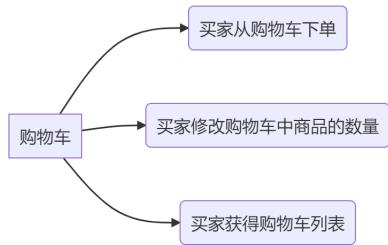


Figure 18: 页面接口示意图 5

Figure 19: 页面接口示意图

2.4.4 开发流程

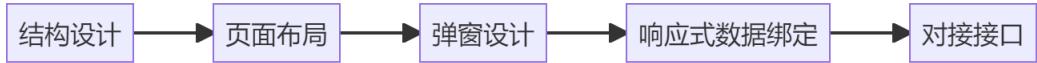


Figure 20: 开发流程图

2.4.5 页面展示



Figure 21: 登录页

登录页提供用户注册和登录功能

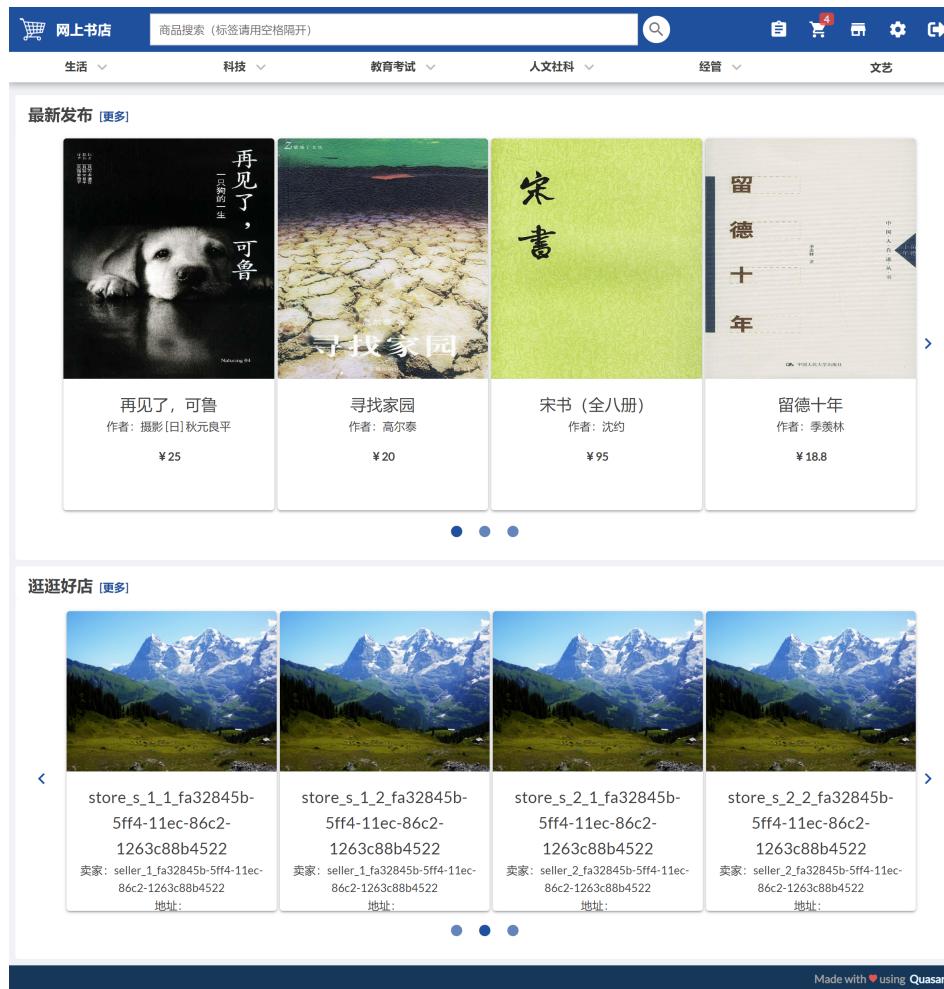


Figure 22: 主页

主页显示部分商品和店铺推荐



Figure 23: 全站搜索页

全站搜索页依据用户搜索的文本对标题、内容等进行全站的商品搜索，依据发布时间、销量或价格等，分页展示。



Figure 24: 店铺详情页

店铺详情页展示店铺信息和店铺内的商品，同时也可以进行关键字搜索、排序等，分页展示。

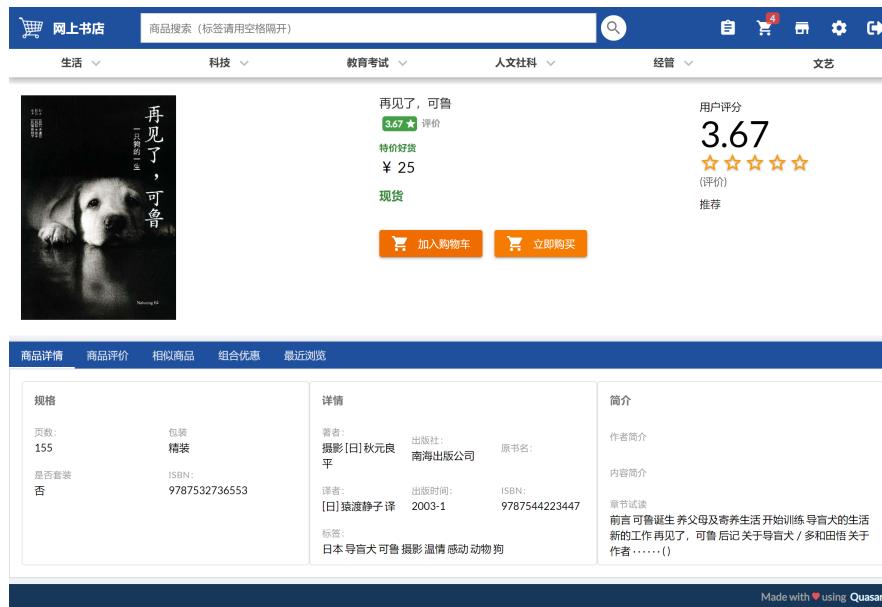


Figure 25: 商品详情页

商品详情页详细展示书籍的信息，并且提供加入购物车或者直接下单的功能。

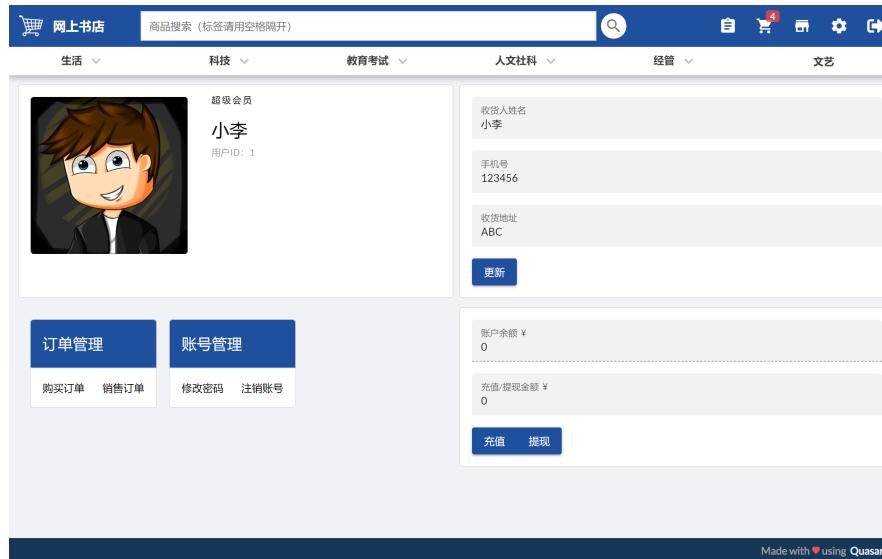


Figure 26: 用户详情页

用户详情页详细展示用户的信息，并且提供修改收货地址，修改密码，注销账号，余额充值与提现等功能。

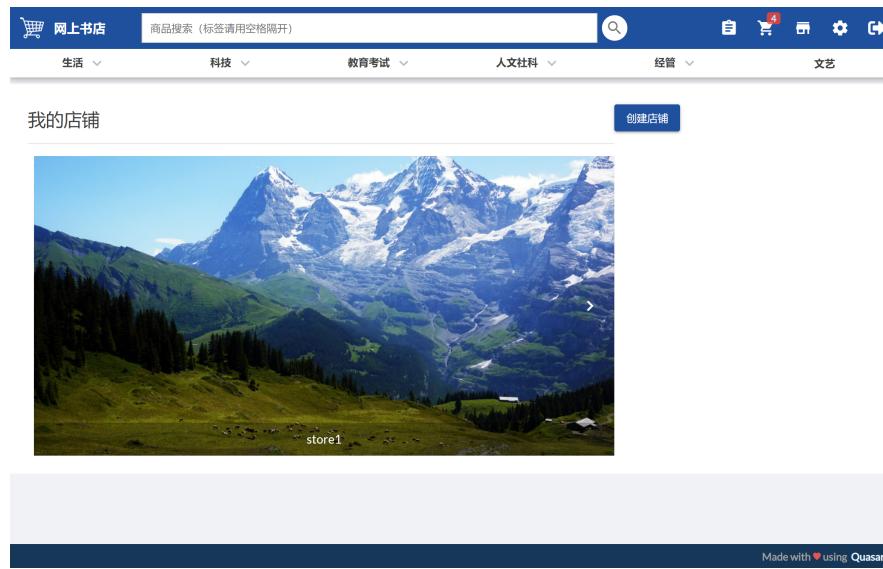


Figure 27: 我的商店页

我的商店页可以查看和进入用户拥有的商店，也提供创建新的商店的功能。

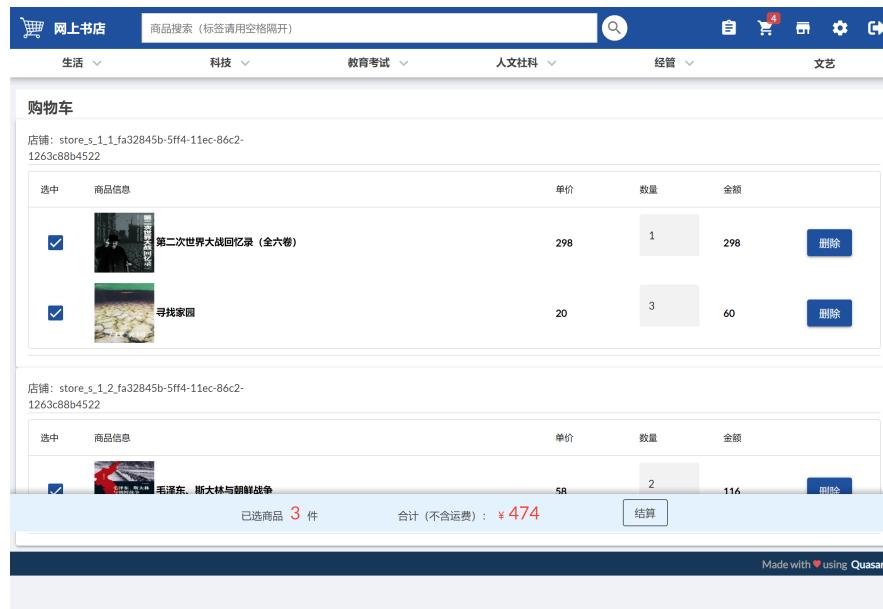


Figure 28: 购物车页

购物页可以查看购物车内的商品，修改加购数量，以及从购物车进行下单。

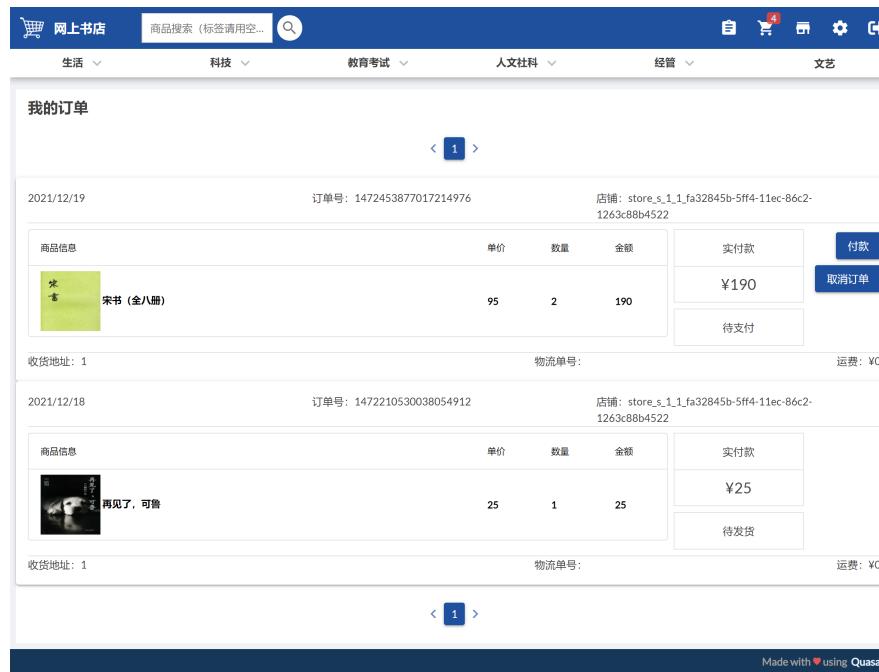


Figure 29: 订单页

订单页可以查看购买或销售订单的详细信息，并且依据订单所处的状态进行付款、确认收货、发布评价等功能。

2.4.6 特点

- 数据响应式绑定，一方面简化开发人员对页面内容的操作方式，另一方面使用户实时直观地看到自己正在修改的内容对页面其他数据的影响。
- 采用高性能框架，减少前端页面内容对用户设备的页面渲染开销，并降低带宽压力。
- 网页模块化设计，有利于前端框架的维护工作，简化对前端进行进一步开发的流程。

2.4.7 后续开发方向

- 优化功能：调整页面布局和浏览器本地存储策略，优化用户的信息获取方式，减少用户重复填入信息的操作。
- 响应式支持：进一步优化响应式布局设计，构建 PC、平板电脑、手机端都能获得优秀使用体验的前端设计。

2.5 后端实现

2.5.1 Springboot 服务构架

为了实现一个完整的电商服务平台并能够应对各种可能的情况，我们必须使用一个足够方便且功能全面的后端开发框架。作为当前世界上最流行的后端服务框架，Spring 拥有着后端开发中最全的生态系统。为了方便开发，在本项目中，我们使用了 **Springboot** 作为后端开发框架。

书店系统的后端构架如下图所示 (Figure 30)。整个系统共分为 5 层，**拦截层 (Interceptor)** 负责验证请求的 Token 信息是否合法，**控制层 (Controller)** 负责接口的声明和参数传递，**服务层 (Service)** 负责各种服务的实现，**持久化层 (DAO, Data Access Operator)** 负责与数据库的 CRUD 及其他各种操作，**模型层 (Model)** 用于声明各种应用中存在的数据实体。

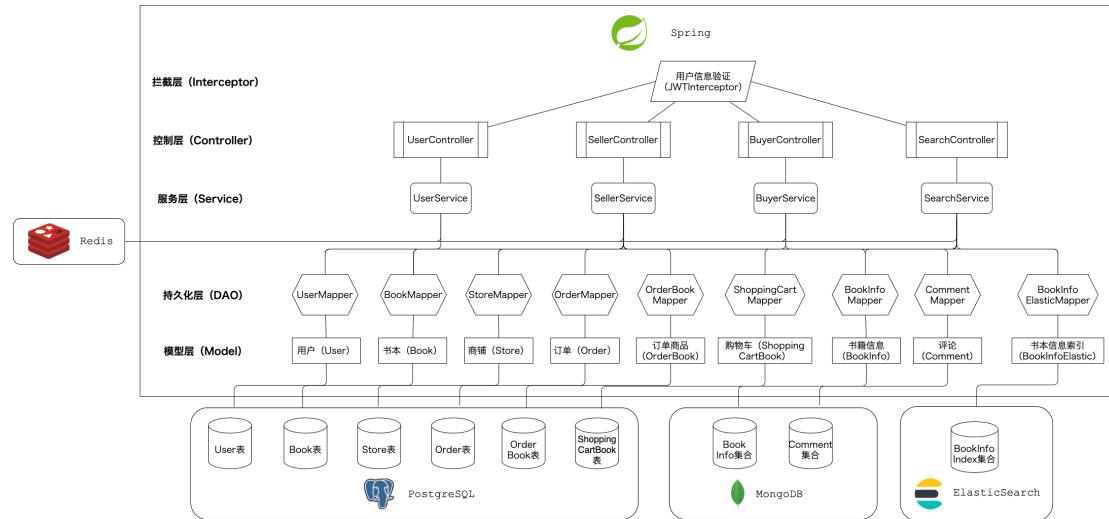


Figure 30: 书店系统后端服务构架图

为了方便开发和统一接口，我们使用了 **ORM (Object Relational Mapping)** 和 **ODM (Object Document Mapping)** 的方式将 Java 对象和数据库中的实体建立联系。这一功能是由 Spring 生态中强大的驱动器 (Driver) 组件所保证的。具体来说，本项目用到的全部模块依赖如下表所示 (Table 2)。

模块名	版本	功能
Spring Boot Starter Web	2.6.1	Spring Web 服务
Mybatis Plus Boot Starter	3.4.3.4	ORM 模型及标准 CRUD 操作
Spring Boot Start Data Redis	2.6.1	Redis 驱动器
Redisson	3.16.6	Redis 扩展功能 (布隆过滤器)
Spring Kafka	2.8.0	Kafka 驱动器
Caffeine	3.0.5	高性能进程内缓存
PostgreSQL	42.3.1	PostgreSQL 驱动器
Hystrix Core	1.5.18	服务降级组件
Lombok	1.18.22	实体模型方法补全器
Spring Boot Starter Test	2.6.1	单元测试组件
JAVA JWT	3.10.3	登录验证组件
Spring Boot Starter Data MongoDB	2.6.1	MongoDB 驱动器
Spring Data ElasticSearch	4.3.0	ElasticSearch 驱动器

Table 2: 书店项目后端用到的全部依赖模块

在实际处理请求的过程中，各功能层会被逐层调用，以做到业务的隔离。以买家服务控制器 (BuyerController) 为例，其加载流程如下图所示 (Figure 31)。可以看到，控制器层首先调用各服务接口层 (Service Interface)，并由服务实现层 (Service Implementation) 完成所有

的请求处理逻辑。对于其中的数据库操作，服务层再接着调用持久化层和模型层完成相应的操作。

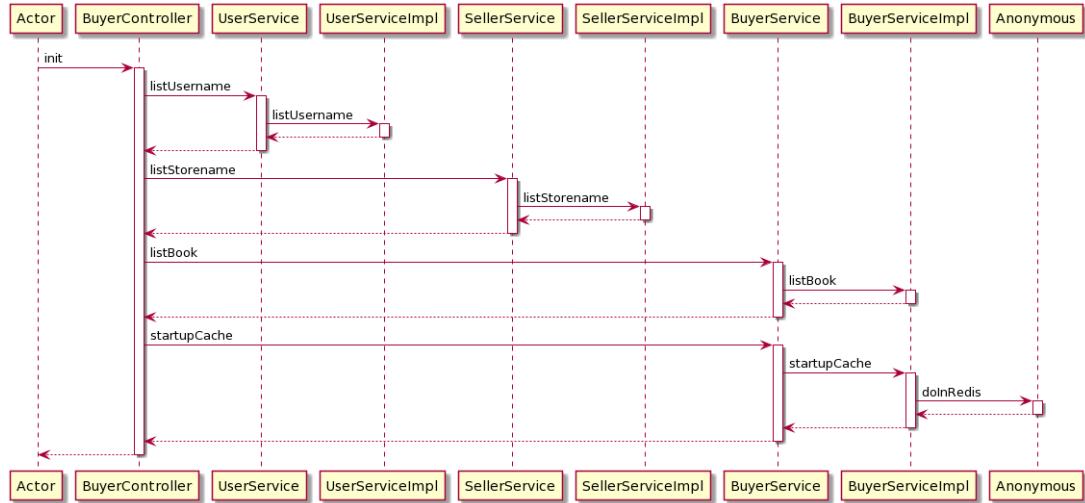


Figure 31: BuyerController 加载时序图

2.5.2 JWT 与登录验证

2.5.2.1 JWT 介绍

JSON Web Token (JWT) 是一个开放标准 (RFC7519)，它定义了一种紧凑的、自包含的方式，用于作为 JSON 对象在各方之间安全地传输信息。该信息可以被验证和信任，因为它是数字签名的。

2.5.2.2 JWT 组成

JWT 由三部分组成。第一部分为头部 (header)，第二部分为载荷 (payload)，第三部分是签名 (signature)。头部包含 token 的类型 (“JWT”) 和加密算法名称 (如 SHA256)，载荷包含用户的有效信息 (不能是敏感信息)，签名用于验证消息在传递过程中是否被更改，要得到签名，需要仅存在于服务端的密钥 (secret)。

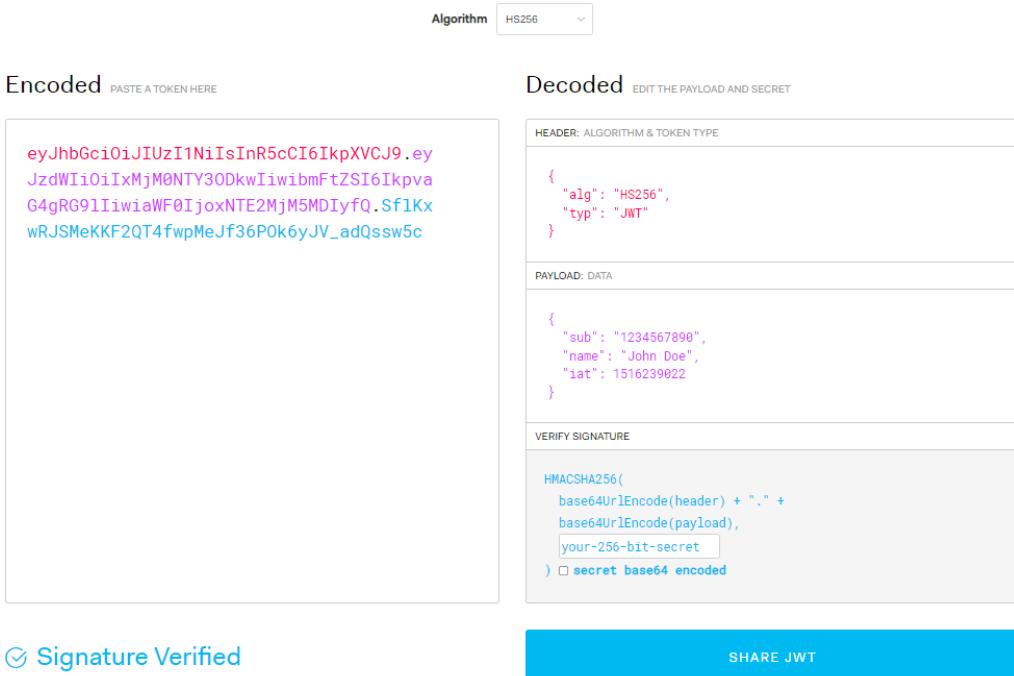


Figure 32: JWT 结构

2.5.2.3 JWT 工作流程

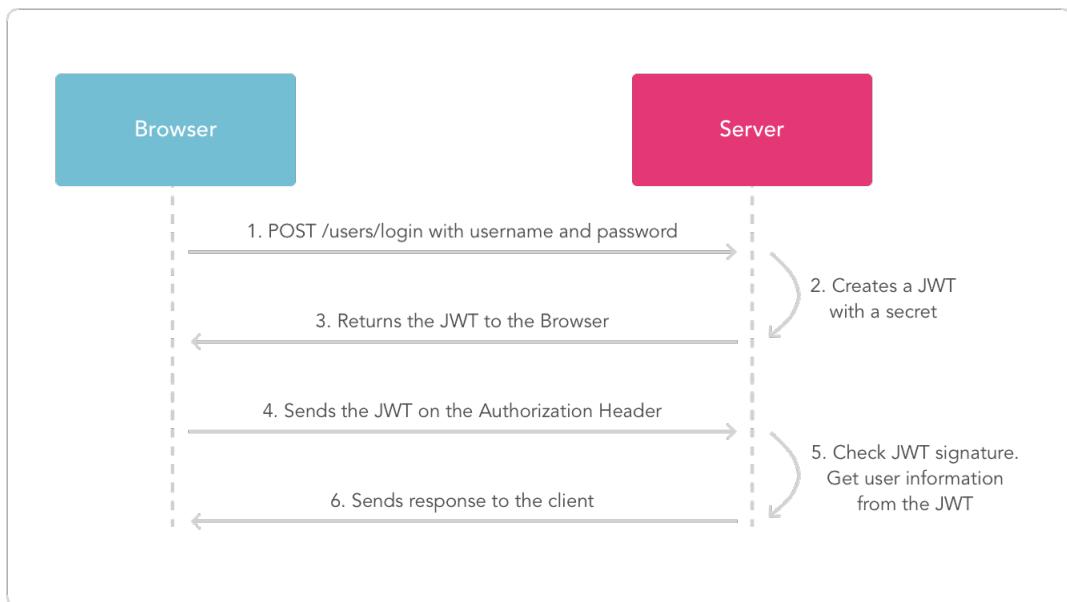


Figure 33: JWT 工作流程图

2.5.2.4 JWT 优点

- 由于 JSON 的通用性，JWT 是跨语言支持的。
- 便于传输，JWT 的构成简单，字节占用小。
- 不需要在服务端保存会话信息，易于应用的扩展。

2.5.2.5 JWT 应用

配置 JWT 拦截器 (JWT Interceptor)，编写 JWT 生成和验证方法，除了特定接口直接放行之外 (如登录、注册不需要 token 验证)，其余接口的请求都需要通过 JWT 验证才能继续进行。具体代码如下。

```
1  /* JWTInterceptor.java */
2  // JWT拦截器
3  public class JWTInterceptor implements HandlerInterceptor {
4
5      @Override
6      public boolean preHandle(HttpServletRequest request, HttpServletResponse
7          response, Object handler) throws Exception {
8          String token=request.getHeader("token");
9          Message message=new Message();
10         try{
11             JWTUtils.verify(token);
12             message.setMessage("ok");
13             return true;
14         }catch (Exception e){
15             message.setMessage("token错误");
16             response.setStatus(HttpStatus.SC_UNAUTHORIZED);
17             response.setContentType("application/json; charset=UTF-8");
18             response.getWriter().write(new ObjectMapper().writeValueAsString(
19                 message));
20             return false;
21         }
22     }
23
24  /* JWTInterceptorConfig.class */
25  // JWT拦截器配置
26  @Configuration
27  public class JWTInterceptorConfig implements WebMvcConfigurer {
28
29      @Override
30      public void addInterceptors(InterceptorRegistry registry) {
31          registry.addInterceptor(new JWTInterceptor())
32              .addPathPatterns("/**")
33              .excludePathPatterns( // 不需要JWT验证直接放行的接口
34                  "/auth/register",
35                  "/auth/unregister",
36                  "/auth/login",
37                  "/auth/password",
38                  "/buyer/payment",
39                  "/buyer/add_funds"
```

```

39
40
41
42
43 /* JWTUtils.class */
44 // JWT工具类
45 public class JWTUtils {
46     private static final String SECRET="gld-bookstore-*%#@*!&"; // 密钥
47
48     // JWT生成
49     public static String generateToken(LoginBody loginBody){
50         JWTCreator.Builder builder= JWT.create();
51         builder.withClaim("name",loginBody.getName());
52         builder.withClaim("terminal",loginBody.getTerminal());
53         Calendar instance=Calendar.getInstance();
54         instance.add(Calendar.HOUR,24);
55         builder.withExpiresAt(instance.getTime());
56         return builder.sign(Algorithm.HMAC256(SECRET));
57     }
58
59     // JWT验证
60     public static void verify(String token) {
61         JWT.require(Algorithm.HMAC256(SECRET)).build().verify(token);
62     }
63 }

```

2.5.3 ORM 框架的使用

在 SpringBoot 中，常用的 ORM 框架有两个，一个是 JPA，另一个是 MyBatis。JPA 可以实现自动创建数据库表结构，而 Mybatis 却不行。但 Mybatis Plus 封装的 CURD 接口要比 JPA 的接口使用起来更加方便，因此本项目即使用了 JPA 用于自动建表，又使用了 Mybatis Plus 的 CURD 接口。

2.5.3.1 JPA 自动建表

JPA 的自动建表功能十分方便，只需在实体类及其属性上加上相应的注解即可，具体代码如下。

```

1 /* Book.class */
2 @Data
3 @AllArgsConstructor
4 @NoArgsConstructor
5 @Entity // 建表注解
6 @Table(name = "book", indexes = { // 表名、索引注解
7     @Index(name = "book_info_and_store", columnList = "infoId"),
8     @Index(name = "book_info_and_store", columnList = "storeName"),
9     @Index(name = "book_time", columnList = "time"),
10    @Index(name = "book_sale", columnList = "sale"),
11    @Index(name = "book_price", columnList = "price")
12 })
13 public class Book {
14     @Id // 主键注解

```

```

15  @GeneratedValue(strategy = GenerationType.IDENTITY) // 主键生成策略注解,
  此处为自增
16  private Long id;
17  @Column(nullable = false) // 字段不可空注解
18  private String infoId;
19  @Column(nullable = false)
20  private int price;
21  @Column(nullable = false)
22  private int stockLevel;
23  @Column(nullable = false)
24  private String storeName;
25  @Column(nullable = false)
26  private int sale;
27  @Column(nullable = false)
28  private long time;
29 }

```

2.5.3.2 基于 Mybatis Plus 的 CURD

使用 Mybatis Plus 封装的 CURD 接口可以方便地操作数据库，几乎不用写 SQL 语句。本项目中用到的 mapper 层接口如下。

```

1 int insert(T entity); // 插入, 返回插入条数
2 int deleteById(Serializable id); // 通过主键删除, 返回删除条数
3 int delete(@Param("ew") Wrapper<T> queryWrapper); // 删除, 返回删除条数
4 int update(@Param("et") T entity, @Param("ew") Wrapper<T> updateWrapper); // 更新, 返回更新条数
5 T selectById(Serializable id); // 通过主键查询, 返回查询结果
6 T selectOne(@Param("ew") Wrapper<T> queryWrapper); // 单个查询, 返回查询结果
7 Long selectCount(@Param("ew") Wrapper<T> queryWrapper); // 计数, 返回数量
8 List<T> selectList(@Param("ew") Wrapper<T> queryWrapper); // 多个查询, 返回查询结果
9 <P extends IPage<T>> P selectPage(P page, @Param("ew") Wrapper<T> queryWrapper); // 分页查询, 返回 IPage, IPage 中包含查询结果和查询总数

```

2.5.4 用户服务

2.5.4.1 注册

向数据库 user 表中插入一个新用户，如果捕获到 DuplicateKeyException 异常，则说明用户名已存在，反之，则说明注册成功。

2.5.4.2 注销

根据用户名和密码来删除 user 表中的用户，如果删除的条目数大于 0，则说明注销成功，否则注销失败。

2.5.4.3 登录

根据用户名和密码来查找 user 表中的用户，如果能够找到，则利用 user_id 和 terminal 字段来生成 JWT 返回给用户，登录成功，否则登录失败。

2.5.4.4 修改密码

根据用户名和新旧密码来更新 user 表中的用户，如果更新的条目数大于 0，则说明修改成功，否则修改失败。

2.5.4.5 登出

根据用户名来查找 user 表中的用户，如果能够找到，则登出成功，否则登出失败。

2.5.4.6 获取信息

根据用户名来查找 user 表中的用户，如果能够找到，则将查询结果中的密码置空后返回给用户（避免密码泄露），获取成功，否则获取失败。

2.5.4.7 更新信息

根据用户名来查找 user 表中的用户，如果找不到，则表示用户名不存在，如果找得到，则更新用户信息（昵称、手机号或者地址）。

2.5.5 买家服务

2.5.5.1 下单

由于要求中需要对下单接口的吞吐量进行测试，我们使用了多种优化和缓存技术。具体实现详见本文的[2.6.4](#)章节。

2.5.5.2 付款

由于要求中需要对付款接口的吞吐量进行测试，我们使用了多种优化和缓存技术。具体实现详见本文的[2.6.4](#)章节。

2.5.5.3 充值

首先根据用户名和密码来查找 user 表中的用户，判断能否找到。其次判断充值的金额是否大于 0。如果两次验证都通过，则更新用户余额。

2.5.5.4 提现

首先根据用户名来查找 user 表中的用户，判断能否找到。其次判断用户余额是否大于等于要提现的金额。如果两次验证都通过，则更新用户余额。

2.5.5.5 收货

根据买家用户名、订单号和订单状态来更新 order 表中的订单，如果更新条目数等于 0，则说明没有符合要求的订单。如果大于 0，则收货成功，并依次增加商品销量、店铺销量和卖家用户的余额。由于该过程涉及到对多个表的更新操作，因此采用事务处理。

2.5.5.6 获取购物车信息

首先根据用户名获取 shopping_cart_book 表中的购物车信息，然后根据商品 id 查找 book 表获取对应的商品信息，最后根据 info_id 查找 book_info 集合来获取图书的详细信息。把查询到的数据按照一定的结构返回给用户。

2.5.5.7 添加商品到购物车

首先根据商品 id 查询 book 表判断商品是否存在。若存在，则向 shopping_cart_book 表中插入相关信息。如果捕获到 DuplicateKeyException 异常，则说明该商品已经存在于该用户的购物车中，改为增加购物车中该商品的数量。

2.5.5.8 修改购物车中商品的数量

如果传入的商品数量小于等于零，则直接把该商品从 shopping_cart_book 表中删除，否则修改为指定数量。如果以上操作的条目数为 0，则说明用户的购物车中不存在该商品，修改失败。如果不为 0，则修改成功。

2.5.5.9 从购物车下单

首先把相关商品从 shopping_cart_book 表中删除，然后调用“下单”方法。由于该过程涉及到对多个表的更新操作，因此采用事务处理。

2.5.5.10 评论

根据买家用户名、订单号和订单状态来更新 order 表中的订单，如果更新条目数等于 0，则说明没有符合要求的订单。如果大于 0，则向 comment 集合中插入评论。

2.5.5.11 分页获取订单

首先使用 Mybatis Plus 提供的 selectPage 方法按照时间顺序在 order 表中选出订单列表，然后依次查询 order_book 表、book 表和 book_info 集合，把查询到的数据按照一定的结构返回给用户。

2.5.5.12 取消订单

根据买家用户名、订单号和订单状态来更新 order 表中的订单，如果更新条目数等于 0，则说明没有符合要求的订单。如果大于 0，则取消成功。

除了用户手动取消订单之外，我们还实现了基于 redis 的自动取消订单功能，详见[2.6.1](#)章节。

2.5.6 卖家服务

2.5.6.1 创建商店

向 store 表中插入商店，如果捕获到 DuplicateKeyException 异常，则说明商店名已存在。反之，则创建成功。

2.5.6.2 添加商品

首先在 user 表中查询判断用户是否存在，然后在 store 表中查询判断商店是否存在。如果以上验证通过，则在 book 表中插入商品，如果捕获到 DuplicateKeyException 异常，则说明商店已存在该商品，反之，则添加成功。

2.5.6.3 添加商品库存

首先在 store 表中查询判断商店是否存在，然后判断商店的卖家用户名是否与传入的一致。如果一致，则根据书籍信息 id、商店名和库存来更新 book 表中的商品，如果更新的条目数为 0，则说明不存在符合要求的商品。如果不为零，则添加库存成功。

2.5.6.4 删除商店

根据商店名和卖家用户名删除 store 表中的商店，如果删除的条目数为 0，则说明商店不存在。如果不为 0，则删除成功。

2.5.6.5 获得自己所有的商店信息

根据卖家用户名来查询 store 表，返回查询结果。

2.5.6.6 下架商品

首先在 book 表中查询判断商品是否存在，然后根据商店名查询 store 表，判断商店的卖家用户名是否与传入的一致。如果一致，则从 book 表中删除商品。

2.5.6.7 发货

首先根据订单号查询订单，判断订单是否存在及状态是否正确。然后依次查询 store 表和 user 表判断订单的卖家与传入的是否一致。如果一致，则给订单添加发货地址、收货地址和物流单号。

2.5.6.8 分页获取订单

首先获得卖家的所有商店，然后使用 Mybaits Plus 提供的 selectPage 方法按照时间顺序在 order 表中选出订单列表，然后依次查询 order_book 表、book 表和 book_info 集合，把查询到的数据按照一定的结构返回给用户。

2.5.7 搜索服务

2.5.7.1 分页商品搜索

首先用 Elastic Search 对用户的输入进行搜索（具体实现详见本文的2.5.8章节），返回书籍信息 id 列表。然后使用 Mybaits Plus 提供的 selectPage 方法按照 strategy 参数指定的顺序在 book 表中选出商品列表，然后查询 book_info 集合，把查询到的数据按照一定的结构返回给用户。

2.5.7.2 获取商品详细信息

根据商品 id 查询 book 表，然后根据图书信息 id 查询 book_info 集合，返回查询到的数据。

2.5.7.3 分页获取商品评论

根据商品 id 来查询 comment 集合中的评论信息。

2.5.7.4 获取商品星级

根据商品 id 来查询 comment 集合中的星级信息。

2.5.7.5 获取商店详情

根据商店名来查询 store 表，然后根据卖家用户名查询 user 表，把查询到的数据按照一定的结构返回给用户。

2.5.7.6 获取商店列表

按商店创建的时间顺序获取指定个数的商店信息。

2.5.8 ElasticSearch 与全文搜索

Elasticsearch 是一款基于 Apache Lucene 的分布式文档型数据库。Elasticsearch 的开发目的就是实现一个能够实时高效的对非结构化数据进行各种检索操作的搜索引擎，因此其被大量的用于各种 **OLAP (Online Analytical Processing)** 场景。

ElasticSearch 中使用了倒排索引 (**Inverted Index**) 技术为加入的每一个文档建立了倒排索引，这也是其能够实现高效全文搜索的重要基础之一。倒排索引的基本原理如下图所示 (Figure 34)。对于每一个加入的文档，系统会使用分词器对其进行分词。随后，系统会建立一个词项 → 文档编号的映射表。如此，当用户发起一个对应词项的文档查询请求时，系统就能快速查询出包含该词项的文档。

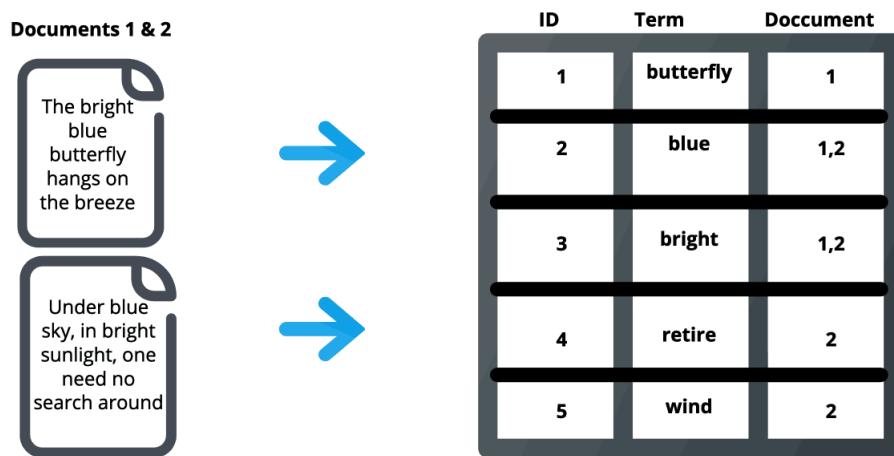


Figure 34: Elasticsearch 中倒排索引的基本原理 [Abu20]

Elasticsearch 默认的分词器对中文的支持并不理想，对于一个中文语句，其往往会将其逐字拆开作为词项，这显然是不合适的。为此，社区开发者 medcl 基于 Lucene IK analyzer 为 ElasticSearch 提供了一个名为 **Analyze IK** 的第三方分词。两种分词器对同一个中文语句的分词结果如下图所示 (Figure 35)。可以看到，使用 Analyze IK 对中文语句的分词效果远好于默认分词器。

默认分词器 (Left)	Analyze IK (Right)
<pre> "tokens": [{ "token": "我", "start_offset": 0, "end_offset": 1, "type": "<IDEOGRAPHIC>", "position": 0 }, { "token": "是", "start_offset": 1, "end_offset": 2, "type": "<IDEOGRAPHIC>", "position": 1 }, { "token": "一", "start_offset": 2, "end_offset": 3, "type": "<IDEOGRAPHIC>", "position": 2 }, { "token": "名", "start_offset": 3, "end_offset": 4, "type": "<IDEOGRAPHIC>", "position": 3 }, { "token": "学", "start_offset": 4, "end_offset": 5, "type": "<IDEOGRAPHIC>", "position": 4 }] </pre>	<pre> "tokens": [{ "token": "我", "start_offset": 0, "end_offset": 1, "type": "CN_CHAR", "position": 0 }, { "token": "是", "start_offset": 1, "end_offset": 2, "type": "CN_CHAR", "position": 1 }, { "token": "一", "start_offset": 2, "end_offset": 4, "type": "CN_WORD", "position": 2 }, { "token": "名", "start_offset": 4, "end_offset": 6, "type": "CN_WORD", "position": 3 }] </pre>

Figure 35: 默认分词器与 Analyze IK 对“我是一名学生”的分词结果对比 (左为默认分词器, 右为 Analyze IK 分词器)

在本项目中, Elasticsearch 索引文档的结构如下表所示 (Table 3)。其中, Elasticsearch 会对文本 (Text) 类型进行分词, 并将词项以倒排索引的形式存储在数据库中; 而对关键字 (Keyword) 类型, ES 直接将其视作一个整体存入数据库。这一区分也符合我们日常过程中对关键字的查询结果预期。

键名	值类型	值说明
title	文本 (Text)	图书标题
author	关键字 (Keyword)	图书作者
publisher	关键字 (Keyword)	出版商
originaltitle	文本 (Text)	图书原标题
translator	关键字 (Keyword)	图书翻译者
authorintro	文本 (Text)	作者介绍
bookintro	文本 (Text)	图书介绍
content	文本 (Text)	图书内容预览

Table 3: 书店系统的 Elasticsearch 索引文档设计

与访问其他数据库类似, 我们同样使用了 ODM 模式将 JAVA 对象与 Elasticsearch 文档相关联, 并使用映射器 (Mapper) 的方式对其进行访问。除了基本的 CRUD 操作外, Elasticsearch 为我们提供了强大的全文索引功能。我们只需要在 Mapper 中加入相关字段的接口, 就可以实现对对应字段的模糊查询操作:

```

1 public interface BookInfoElasticMapper extends ElasticsearchRepository<
2   BookInfoElastic, String> {
3   List<BookInfoElastic> findByTitleContaining(String title);
4   List<BookInfoElastic> findByAuthorContaining(String author);
5   List<BookInfoElastic> findByBookIntroContaining(String keyword);
6   List<BookInfoElastic> findByContentContaining(String keyword);

```

6 }

将测试给定的书本数据插入 Elasticsearch 后的存储情况如下图所示 (Figure 36)。可以看到, ES 以分片的方式在不同的空间中管理插入的数据, 并为每组数据标记了其索引的位置和各种信息。

查询 1 个分片中用的 1 个, 100 命中, 耗时 0.013 秒								
<u>index</u>	<u>type</u>	<u>_id</u>	<u>_score</u>	<u>_class</u>	<u>id</u>	<u>title</u>	<u>author</u>	<u>publisher</u>
bookinfoindex	_doc	1000280	1	gld.bookstore.entity.BookInfoElastic	1000280	袁氏当国	[美] 唐德刚	广西师范大学出版社
bookinfoindex	_doc	1000317	1	gld.bookstore.entity.BookInfoElastic	1000317	一个狗娘养的白痴	(美)艾伦·纽哈斯	东方出版社
bookinfoindex	_doc	1000531	1	gld.bookstore.entity.BookInfoElastic	1000531	第二次世界大战回忆录(全六卷)	温斯顿·丘吉尔	南方出版社
bookinfoindex	_doc	1000167	1	gld.bookstore.entity.BookInfoElastic	1000167	中国历代年号考	李崇智	中华书局
bookinfoindex	_doc	1000364	1	gld.bookstore.entity.BookInfoElastic	1000364	新唐书(全二十册)	宋祁	中华书局
bookinfoindex	_doc	1002898	1	gld.bookstore.entity.BookInfoElastic	1002898	父与子全集	[德] 埃·奥·卜劳恩 绘	中国工人出版社
bookinfoindex	_doc	1003246	1	gld.bookstore.entity.BookInfoElastic	1003246	殷商史	胡厚宣, 胡振宇	上海人民出版社
bookinfoindex	_doc	1002400	1	gld.bookstore.entity.BookInfoElastic	1002400	城的灯	李佩甫	长江文艺出版社
bookinfoindex	_doc	1002474	1	gld.bookstore.entity.BookInfoElastic	1002474	只有偏执狂才能生存	[美] 安迪·格鲁夫	中信出版社 辽宁教育出版
bookinfoindex	_doc	1002505	1	gld.bookstore.entity.BookInfoElastic	1002505	故宫史话	单士元	新世界出版社
bookinfoindex	_doc	1003284	1	gld.bookstore.entity.BookInfoElastic	1003284	退步集	陈丹青	广西师范大学出版社
bookinfoindex	_doc	1002597	1	gld.bookstore.entity.BookInfoElastic	1002597	田野图像	李亦园	山东画报出版社
bookinfoindex	_doc	1003373	1	gld.bookstore.entity.BookInfoElastic	1003373	李敖快意恩仇录	李敖	中国友谊出版社
bookinfoindex	_doc	1003000	1	gld.bookstore.entity.BookInfoElastic	1003000	悟空传	今何在	光明日报出版社
bookinfoindex	_doc	10048862	1	gld.bookstore.entity.BookInfoElastic	10048862	中国社会的个体化	阎云翔	上海译文出版社
bookinfoindex	_doc	1005047	1	gld.bookstore.entity.BookInfoElastic	1005047	剑桥中华人民共和国史(上卷)	R.麦克法夸尔	中国社会科学出版社
bookinfoindex	_doc	1005242	1	gld.bookstore.entity.BookInfoElastic	1005242	古船	张炜	人民文学出版社
bookinfoindex	_doc	1004539	1	gld.bookstore.entity.BookInfoElastic	1004539	爱与孤独	周国平	广西师范大学出版社
bookinfoindex	_doc	1005358	1	gld.bookstore.entity.BookInfoElastic	1005358	寻找家园	高尔泰	花城出版社
bookinfoindex	_doc	1004992	1	gld.bookstore.entity.BookInfoElastic	1004992	张爱玲文集	张爱玲	安徽文艺出版社
bookinfoindex	_doc	1005455	1	gld.bookstore.entity.BookInfoElastic	1005455	淘气包埃米尔	[瑞典] 阿斯特丽德·林格伦	中国少年儿童出版社
bookinfoindex	_doc	1004956	1	gld.bookstore.entity.BookInfoElastic	1004956	阿修羅	[加拿大] 亦舒	海天出版社
bookinfoindex	_doc	1005411	1	gld.bookstore.entity.BookInfoElastic	1005411	透明的红萝卜	莫言	时代文艺出版社
bookinfoindex	_doc	1005492	1	gld.bookstore.entity.BookInfoElastic	1005492	光荣与梦想	[美] 威廉·曼彻斯特	海南出版社 三环出版社
bookinfoindex	_doc	1005576	1	gld.bookstore.entity.BookInfoElastic	1005576	影响力	[美] 罗伯特·B·西奥迪尼	中国社会科学出版社
bookinfoindex	_doc	1006203	1	gld.bookstore.entity.BookInfoElastic	1006203	工商巨子	(美)荣·切尔诺	海南出版社
bookinfoindex	_doc	1006419	1	gld.bookstore.entity.BookInfoElastic	1006419	十六世纪明代中国之财政与税收	[美] 黄仁宇	生活·读书·新知三联书店
bookinfoindex	_doc	1006560	1	gld.bookstore.entity.BookInfoElastic	1006560	血酬定律	吴思	中国工人出版社
bookinfoindex	_doc	1005660	1	gld.bookstore.entity.BookInfoElastic	1005660	雕刻时光	[俄] 安德烈·塔可夫斯基	人民文学出版社
bookinfoindex	_doc	1005918	1	gld.bookstore.entity.BookInfoElastic	1005918	堂吉诃德		译林出版社

Figure 36: 给定数据在 Elasticsearch 分片中的存储情况

对于一个关键字查询请求, 我们首先通过上面的接口方法在各个字段中查询出满足条件的书籍信息实体。随后, 我们通过其对应的 **BookinfoID** 在书籍实体中查询出出售该种书籍的全部商店, 并根据用户给定的请求参数对结果进行编排后返回给用户。相关实现代码如下:

```

1 public class SearchServiceImpl implements SearchService {
2     @Override
3     public ResponseEntity<SearchMessage> bookSearch(String storeName, int
4         currentPage, int numPerPage, int strategy, String keywords) {
5         List<BookVO> messageBooks = new ArrayList<>();
6         Set<String> infoIDSet = new HashSet<>();
7         List<String> infoIDOrder = new ArrayList<>();
8         List<BookInfoElastic> searchResults = new ArrayList<>();
9         String[] keyword_arr = keywords.split(" ");
10        for(String keyword : keyword_arr) {
11            searchResults.addAll(bookInfoElasticMapper.findByTitleContaining(
12                keyword));
13        }
14        for(String keyword : keyword_arr) {

```

```
13         searchResults.addAll(bookInfoElasticMapper.findByAuthorContaining
14             (keyword));
15     }
16     for(String keyword : keyword_arr) {
17         searchResults.addAll(bookInfoElasticMapper.
18             findByBookIntroContaining(keyword));
19     }
20     for(String keyword : keyword_arr) {
21         searchResults.addAll(bookInfoElasticMapper.
22             findByContentContaining(keyword));
23     }
24     LambdaQueryWrapper<Book> bookCondition = new LambdaQueryWrapper<Book
25         >();
26     if(searchResults.size() > 0) {
27         bookCondition.eq(Book::getInfoId, searchResults.get(0).getId());
28         infoIDSet.add(searchResults.get(0).getId());
29         infoIDOrder.add(searchResults.get(0).getId());
30         searchResults.remove(0);
31     }
32     for(BookInfoElastic searchResult : searchResults){
33         if(!infoIDSet.contains(searchResult.getId())) {
34             bookCondition.or().eq(Book::getInfoId, searchResult.getId());
35             infoIDSet.add(searchResult.getId());
36             infoIDOrder.add(searchResult.getId());
37         }
38     }
39     if(!storeName.equals("0")){
40         bookCondition.eq(Book::getStoreName, storeName);
41     }
42     switch (strategy){
43         case 1:
44             bookCondition.orderBy(true, false, Book::getTime);
45         case 2:
46             bookCondition.orderBy(true, false, Book::getSale);
47             break;
48         case 3:
49             bookCondition.orderBy(true, true, Book::getPrice);
50             break;
51         case 4:
52             bookCondition.orderBy(true, false, Book::getPrice);
53             break;
54         default:
55             break;
56     }
57     System.out.println(searchResults.size());
58     System.out.println(infoIDSet.size());
59     IPage<Book> queryPage = bookMapper.selectPage(
60         new Page<>(currentPage, numPerPage),
61         bookCondition
62     );
63     int count = (int) queryPage.getTotal();
64     List<Book> books = queryPage.getRecords();
65     if(strategy == 0){
```

```

62         books.sort(new Comparator<Book>() {
63             @Override
64             public int compare(Book o1, Book o2) {
65                 return infoIDOrder.indexOf(o1.getId()) - infoIDOrder.
66                     indexOf(o2.getId());
67             }
68         });
69         for(Book book : books){
70             messageBooks.add(getMessageBook(book));
71         }
72         return new ResponseEntity<>(
73             new SearchMessage("ok", count, messageBooks),
74             HttpStatus.OK
75         );
76     }
77     //Irrelevant codes
78 }

```

2.6 高可用

高可用性 (High Availability) 是指系统无中断地执行其功能的能力超过其同级别的系统, 代表系统的可用性程度, 是进行系统设计时的准则之一 [Wik21a]。

造成服务不可用的原因有很多, 其中一个十分重要的原因便是过多的并发请求 (Concurrent Request)。在 WEB 服务中, 并发请求是指一个或多个用户在同一时刻或相隔极短的时刻内发送至目标服务器, 并使得服务器需要同时处理并响应的请求。随着互联网规模的高速发展, 这一现象在一些热门服务供应商中体现的越来越明显。近年来, 因为短时间内收到大量请求使得服务器崩溃导致无法提供正常服务的事故屡次发生 (Figure 37, 38), 这些事故严重影响了用户的体验和服务的质量, 也使得学术界和工业界加强了对高可用系统设计的重视。

2012年11月11日, **各大电商风云大战**, 淘宝网和京东网为首的电商赚足了眼球。淘宝双十一网络瘫痪遭诟病, 支付宝被“抢瘫”, 好不容易进入支付过程, 支付宝提示系统繁忙, 经过反复尝试, 花费用户很长时间实现支付。而京东当天流量暴涨, 大量用户登录。结果京东商城的服务器被大流量冲垮, 服务器也瘫痪。

Figure 37: 2012 年“双十一”淘宝京东系统瘫痪 [lis12]

12306网站瘫痪原因揭秘:平均每秒点击24万次|12306|火车...



2014年1月25日 **12306网站瘫痪原因揭秘:平均每秒点击24万次 12306后台监控中心工作人员监控12306客票系统。新华社发 为什么“双十一”的淘宝没崩溃,“12306”网络购票却问题频出?面对黄牛的一次次进...**

新浪网 百度快照

Figure 38: 2014 年春运期间 12306 铁路订票系统瘫痪 [京 14]

一个高可用系统的设计通常具有以下三条原则:

- 消除单点故障 (Elimination of single points of failure)
- 可靠的交叉点服务 (Reliable crossover)
- 提前检测故障并快速恢复 (Detection of failures as they occur)

在本实验中, 我们需要测出书店系统中下单和付款接口的吞吐量, 这正是一个极好的高并发场景的实例。根据上述的三条原则, 我们实现了一系列策略来提升这两个接口的吞吐量, 并使得整个书店系统在各种情况下都能够保持较高的可用性。

2.6.1 Redis 与缓存数据库

由于需要管理的数据量通常十分庞大 (GB 甚至 TB 级), 且需要支持完整的事务处理能力, 常规的关系性数据库通常使用磁盘来对数据进行维护, 并使用页 (Page) 为单位对数据进行存取。因此, 对数据库中的内容进行一次访问通常具有很高的 I/O 代价。通过实际的测试我们也发现, 在整个服务的响应逻辑中, 对数据库的访问代价占据了整个请求处理过程中绝大部分的耗时。在高并发场景下, 这一时间开销使得数据库系统很快就会遇到性能瓶颈, 并使得后续的请求难以得到正确的响应。

Redis (Remote Dictionary Server, 远程字典服务) 是一款开源的、基于内存存储的分布式 Key-Value 型数据库 [Wik21b]。作为 NoSQL 的一种, Redis 设计的宗旨就是为了在面对大量随机读写请求时能够以尽可能高的效率存取数据, 因此其天然地拥有极高的并发处理能力。经验值表明, 在一台常规的小型笔记本上, Redis 能够支持约 110000 次/秒的写操作和 81000 次/秒的读操作 [Jai19], 这一数值几乎是传统关系型数据库吞吐量的近 100 倍。因此, 使用 Redis 作为主数据库的缓存数据库来提升整个服务的响应能力是一个十分合适的选择。

我们使用 JMeter 接口压测工具, 分别向直接访问 Postgresql 数据库和使用了 Redis 缓存的两个测试接口发送了 10000 次并发请求, 其响应结果如下图所示 (Figure 39, 40)。可以看到, 使用了 Redis 缓存的接口吞吐量几乎是直接访问 Postgresql 数据库接口的两倍, 前者的平均响应时间更是比后者快了整整 5 倍。

Label	# 样本	平均值	中位数	90% 百...	95% 百...	99% 百...	最小值	最大值	异常 %	吞吐量	接收 K...	发送 K...
login_test	10000	362	353	574	704	1205	3	1620	0.00%	1594.4/sec	294.28	367.46
TOTAL	10000	362	353	574	704	1205	3	1620	0.00%	1594.4/sec	294.28	367.46

Figure 39: 使用 Postgresql 作为目标数据库时接口的响应效率

Label	# 样本	平均值	中位数	90% 百...	95% 百...	99% 百...	最小值	最大值	异常 %	吞吐量	接收 K...	发送 K...
login_test	10000	70	74	113	124	155	2	193	0.00%	3968.3/sec	732.42	914.56
TOTAL	10000	70	74	113	124	155	2	193	0.00%	3968.3/sec	732.42	914.56

Figure 40: 使用 Redis 作为目标数据库时接口的响应效率

要使用 Redis 作为缓存, 我们就需要分析在当前业务中哪些数据是经常被访问的数据, 即所谓的热点数据。在当前应用中, 热点数据的访问主要发生在下单和付款两个接口请求的处理上。对于下单操作, 系统需要读取商品的库存、价格; 对于付款操作, 系统需要读取用户的密码、账户余额及订单的总价。

书店系统的 Redis 缓存结构如下表所示 (Table 4)。其中, user 键和 store 键分别维护当前系统中所有的用户 ID 和商店 ID, 用于快速判断下单和付款请求是否合法; stock_{图书 ID}_{商

店 ID} 和 price_{图书 ID}_{商店 ID} 键分别用于维护热点商品的库存和价格信息（分开存储是为了支持 Redis 的原子级自增（**Increment**）和自减（**Decrement**）操作）； user_{用户名} 键用于存放已登录用户当前的密码及账户余额，用于在付款请求中快速确认用户身份并判断用户账户是否有足够的足额支付当前订单； order_{订单号} 键用于临时存放下单成功的订单总价（若订单创建后的 T 时刻内仍没有被付款接口更新，则该 Redis 键自动过期，也即订单自动取消）。

键名	值类型	值说明
user	集合 (Set)	系统中的当前用户集合
store	集合 (Set)	系统中的当前商店集合
user_{用户名}	散列 (Hash)	{ "password": 用户密码, "money": 用户账户余额 }
stock_{图书 ID}_{商店 ID}	字符串 (String)	图书库存
price_{图书 ID}_{商店 ID}	字符串 (String)	图书价格
order_{订单号}	字符串 (String)	订单总价 (T 时刻后自动过期)

Table 4: 书店系统的 Redis 缓存设计

使用 Redis 作为缓存后，高并发接口的数据查询流程如下图所示（Figure 41）。对于一个下单/付款请求，服务会先查看 Redis 中是否存在相应的缓存，若缓存存在，则直接进行判断和处理操作；若缓存不存在，则服务从主数据库中查询数据，并将查询到的数据放入 Redis 缓存中。如此，当收到大量对热点数据的操作请求时，服务就可以快速响应用户的请求，极大的提高了接口的吞吐能力。

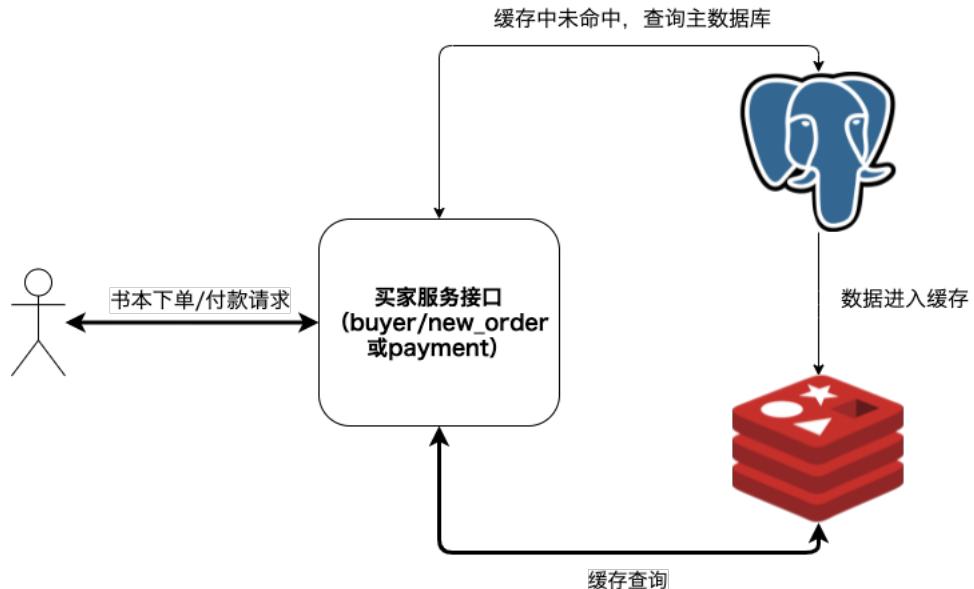


Figure 41: 使用了 Redis 缓存技术后用户下单/付款的数据查询流程

在 Springboot 中，我们可以使用 RedisTemplate 对 Redis 数据库进行操作。我们首先对 RedisTemplate 进行配置，使其能够接受将任意类型的 JAVA 对象并通过 Jackson2Json 工具自动将其自动编码为字符串存入数据库中：

```
1 | @Configuration
```

```

2  @EnableAutoConfiguration
3  public class RedisConfig {
4      @Bean
5      public RedisTemplate<String, Object> redisTemplate(RedisConnectionFactory
6          redisConnectionFactory){
7          RedisTemplate<String, Object> template = new RedisTemplate<>();
8          template.setConnectionFactory(redisConnectionFactory);
9          StringRedisSerializer keySer = new StringRedisSerializer();
10         GenericJackson2JsonRedisSerializer valSer = new
11             GenericJackson2JsonRedisSerializer();
12             template.setKeySerializer(keySer);
13             template.setValueSerializer(valSer);
14             template.setHashKeySerializer(keySer);
15             template.setHashValueSerializer(valSer);
16             return template;
17     }
18 }

```

该段代码会自动设置 JAVA 到 Redis 的相关数据类型转换的编码器。于是，在应用的其他地方，我们就可以直接使用配置好的 RedisTemplate 来对 Redis 缓存进行操作。本项目中所使用到的全部 Redis 操作的代码示例如下：

```

1  redisTemplate.boundValueOps(key).set(value); /* 值设置 */
2  redisTemplate.boundSetOps(key).add(value); /* 集合元素添加 */
3  redisTemplate.boundHashOps(key).put(hashKey, hashValue); /* Hash元素添加 */
4  redisTemplate.hasKey(key); /* 键是否存在 */
5  redisTemplate.boundSetOps(key).isMember(value); /* 值是否存在与集合中 */
6  redisTemplate.boundValueOps(key).get() /* 根据键获得值 */
7  redisTemplate.boundValueOps(key).increment(value); /* 原子自增 */
8  redisTemplate.boundValueOps(key).decrement(value); /* 原子自减 */
9  redisTemplate.boundHashOps(key).get(hashKey); /* 根据键和Hash键获得值 */

```

这里需要注意的是，若直接向 Redis 缓存中以明文存储用户的账户密码，则极有可能造成安全问题。因此，这里我们使用 **MD5 加密算法 (Message Digest Algorithm 5)** 将加密后的用户密码存入 Redis 缓存中 (Figure 42)，并在验证时将请求中的密码以同样的方式进行计算，通过对比明文的方式来判断用户是否给出了正确的密码。

HASH:		user_seller_1_a4b0ecac-5dac-11ec-95c1-acde48001122		Rename Key
#	key	value		
1	password	"e58b9cde01c7c0e7b92649fd2d16061f"		
2	money	0		

Figure 42: Redis 缓存中的用户信息存储结构

作为一个常用的加密算法，Springboot 已经为我们封装好了一个字符串转 MD5 加密字符串的函数 **md5DigestAsHex()**，我们只需要直接调用即可。相关的加密及判断代码如下：

```

1  /* UserServiceImpl.java */
2  @Override
3  public ResponseEntity<TokenMessage> login(LoginBody loginBody) {
4      //Irrelevant codes
5      User user = userMapper.selectOne(query);
6      if(user != null){
7          //Irrelevant codes
8          String md5Password = DigestUtils.md5DigestAsHex(user.getPassword() .
9              getBytes());
10         redisTemplate.boundHashOps("user_" + loginBody.getName()).put(" "
11             "password", md5Password);
12         redisTemplate.boundHashOps("user_" + loginBody.getName()).put("money"
13             , user.getMoney());
14         //Irrelevant codes
15     }
16     //Irrelevant codes
17 }
18
19 /* BuyerServiceImpl.java */
20 @Override
21 @Transactional
22 public ResponseEntity<Message> payment(PayBody payBody) {
23     String md5Password = DigestUtils.md5DigestAsHex(payBody.getPassword() .
24         getBytes());
25     String userRedisKey = "user_" + payBody.getBuyerName();
26     if(Boolean.TRUE.equals(redisTemplate.hasKey(userRedisKey)) && Boolean.
27         TRUE.equals(redisTemplate.hasKey(orderRedisKey))){
28         /* Redis cache hit */
29         if (Boolean.FALSE.equals(Objects.equals(redisTemplate.boundHashOps(
30             userRedisKey).get("password"), md5Password))) {
31             return new ResponseEntity<>(
32                 new Message("付款失败, 授权失败"),
33                 HttpStatus.UNAUTHORIZED
34             );
35         }
36         //Irrelevant codes
37     }
38     //Irrelevant codes
39 }

```

2.6.2 布隆过滤器与缓存穿透

从上面的实现过程我们可以发现，当用户对高并发接口发起一个请求时，服务会先查询 Redis 缓存，若缓存不命中，再访问数据库进行查询。然而，若用户查询的值并不存在于系统中（也即数据库中不存在），则服务必然会对数据库发起查询请求。当这一查询数量增加时，就会有大量的查询请求直接打在数据库上，导致数据库服务崩溃。此时，缓存起不到任何流量分摊的作用，就如同不存在一样，也即产生所谓的缓存穿透（Cache Penetration）问题。

在实际应用场景中，要解决缓存穿透问题，通常具有如下三种方法 [kkl21]：

- 增加校验层，让用户请求频率降低
- 当数据库查询不到目标值时将对应的缓存键设置为空值

- 维护一张系统中所有书本的 ID 值的集合，在收到请求时过滤无效查询

在这里我们使用第三种方法。

要维护系统中所有的书本 ID 集合，我们首先能够想到的就是在 Redis 缓存中增加一个类型为集合的 bookID 键来存放所有的书本 ID。然而，当书本数量增长时，这一空间开销将逐渐变得难以接受。注意到，对于每一个书本 ID，我们只需要记录其是否存在于系统中，因此我们可以使用位图 (Bitmap) 技术来存放所有的 ID。事实上，我们无需精准的过滤每一个无效请求，只需要将请求数量降低到一个数据库能够承受的范围即可。此时，我们完全可以使用一种概率型数据结构来对请求的 ID 进行判断，以减少维护全部书本集合所需要的空间开销。

布隆过滤器 (Bloom Filter) 是一种概率型数据存储结构，其工作原理如下图所示 (Figure 43)。系统首先设置一个固定长度的位数组，随后对于每一个给定的元素，系统会通过确定的 k 个哈希函数将其映射到对应的位上。于是，对于一个待查询元素，系统只需要判断其以同样的方式映射后的位置上是否已被置为 1 即可。具体来说，布隆过滤器可以在上界确定的空间 $\mathcal{O}(m)$, $m \in N^+$ 内对请求的元素给出如下的应答：

- 可能存在于集合中
- 一定不存在于集合中

布隆过滤器可以保证给出第二种应答时一定正确；而对于第一种应答，对于一个由 k 个哈希函数组成的长度为 m 的布隆过滤器，其面对 n 个总数据时的误判率约为 $(1 - \exp\{-\frac{kn}{m}\})^k$ ，这一结果在我们的场景下是完全可以接受的。

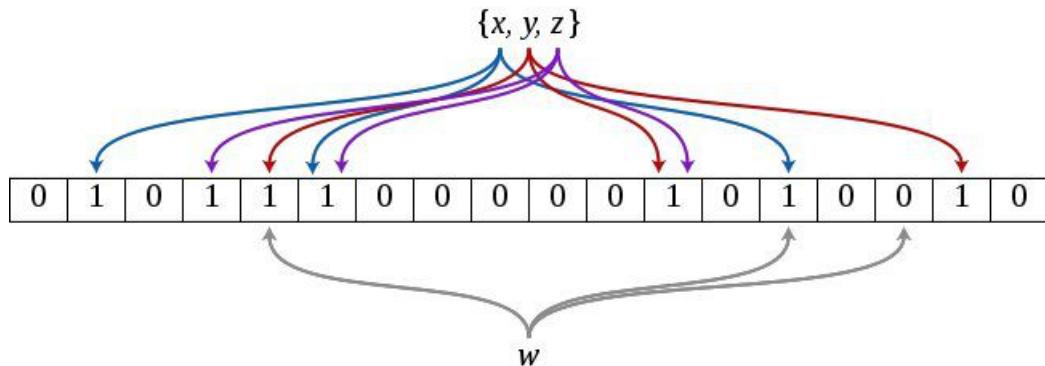


Figure 43: 布隆过滤器原理 [Aky16]

通过将布隆过滤器继承到 Redis 缓存中，用户在请求下单/付款接口时服务的数据查询流程如下图所示 (Figure 44)。在系统启动时，服务会从主数据库中获取当前系统中所有存在的书本 ID，并设置布隆过滤器中的响应位。这样，当接收到一个请求时，系统会首先通过布隆过滤器判断书本 ID 是否可能存在于系统中，若判断结果为一定不存在则直接返回。如此，最终到达缓存和主数据库的请求数量就会被大大降低，减少了系统被恶意攻击时可能的宕机事故发生。

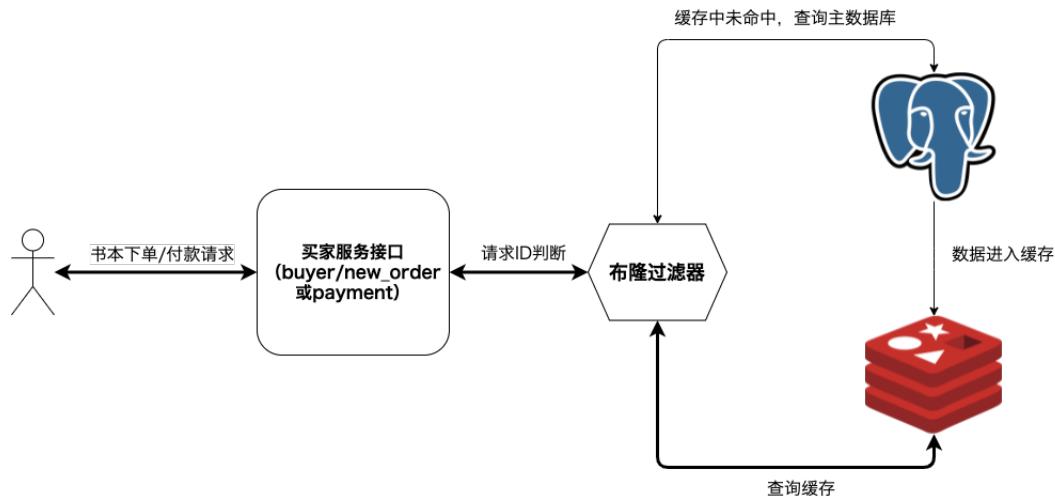


Figure 44: 使用了布隆过滤器后用户下单/付款的数据查询流程

在 Springboot 中，我们可以使用 Google 的 **Guava** 模块或 Redis 原生的 **Redisson** 模块来实现布隆过滤器的继承，这里我们使用了后者。

2.6.3 Caffeine 与多级缓存

通过 Redis 缓存，我们已经可以使得系统在面对相对较高的并发请求时保证响应的效率。那么我们是否又可能进一步的优化，让请求能够更快的得到应答呢？答案是肯定的。

注意到，在秒杀活动时，用户对一件商品的请求量与该商品本身的库存数往往是严重不匹配的（例如一件商品的库存只有 100 件，而在秒杀活动瞬间服务器收到了 10000 次下单请求），我们希望能够在请求达到服务器时就能提前过滤掉这些多余的无效请求，让尽可能合理的请求数到达之后的服务。

在绝大多数使用虚拟内存技术的现代操作系统上，进程内数据通讯只需要引用同一段虚拟内存地址即可，而进程间数据通讯通常要涉及内存拷贝 [TB15]。尽管相比传统关系型数据库，Redis 已经能够提供较高的吞吐效率，但在面对如此大量的请求时，服务进程每次都要与其交互仍然具有不小的代价（若 Redis 服务器与主服务器部署在不同的服务器主机上，交互过程还涉及到 RPC 调用）。因此，我们希望能够在服务内的缓存中就完成对某些请求的判断操作，这也就引出了多级缓存（**Multi-Level Cache**）的概念。

要过滤掉库存数量以外的请求，一个朴素的想法就是在缓存中对于每一个热点商品维护一个“商品是否已售完”的标识符，当检测到库存降低到 0 时，就将缓存中的标识符设为 1。然而，由于此时的请求是并发的，普通的 HashMap 并不能保证在面对如此大量的设置操作时的进程安全性。幸运的是，社区开发者 ben-manes 提供了一个名为 **Caffeine** 的高性能缓存库，其可以支持以下的一些特性：

- 异步的自动缓存项加载
- 当缓存大小达到上限时可根据访问频率和最近访问时间替换缓存
- 可根据最近访问时间对缓存项设置过期时间
- 检测到过期请求时自动刷新缓存项
- 弱引用时缓存键自动打包

- 弱引用或软引用时值自动打包
- 替换缓存事件通知
- 支持双写操作（一份写入缓存，一份写入外部空间）
- 可对缓存访问的特征进行聚合统计

这些特性使得其可以很好的应对短时间内的高并发请求并保证数据的进程安全性。因此，我们使用 Caffeine 来建立多级缓存机制。

集成了多级缓存后，一个下单请求的处理流程如下图所示（Figure 45）。当接口收到请求后，系统会首先检查请求的 BookID 是否在 Caffeine 缓存中已经被标记为售空，若标记存在，则服务直接返回；若不存在，再使用前文中提到的流程继续处理下单请求。

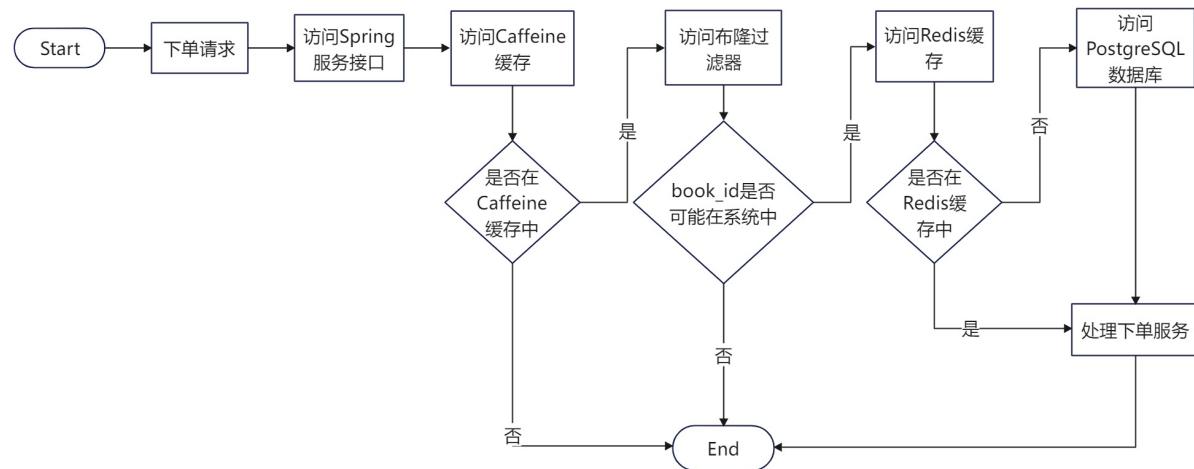


Figure 45: 使用了多级缓存后用户下单的服务响应流程图

对于当前系统，Caffeine 的缓存设置代码如下：

```

1 @Configuration
2 public class CacheConfig {
3     @Bean
4     public Cache<String, Integer> caffeineCache(){
5         return Caffeine.newBuilder()
6             .expireAfterWrite(30, TimeUnit.SECONDS)
7             .initialCapacity(100)
8             .maximumSize(1000)
9             .build();
10    }
11 }
  
```

这里我们设置内部缓存的初始大小为 100 个键值对，最大空间限制为 1000 个键值对。对于每个缓存项，我们设置默认 30 秒的缓存过期时间，以尽快腾出足够的空间方便新的热点数据设置。

下单请求中利用多级缓存过滤无效请求的相关代码如下：

```

1 Cache<String, Integer> caffeineCache;
2
  
```

```

3  public ResponseEntity<OrderMessage> newOrder(NewOrderBody newOrderBody) {
4      //Irrelevant codes
5      /* Redundant request blocking */
6      if(caffeineCache.getIfPresent(cacheKey) != null){
7          return new ResponseEntity<>(
8              new OrderMessage("下单失败, 商品库存不足", null), null,
9              IHttpStatus.LOW_STOCK_LEVEL);
10     }
11     if(redisTemplate.hasKey(stockRedisKey) && redisTemplate.hasKey(
12         priceRedisKey)){
13         /* Redis cache hit procedure */
14         Long stockLevel = redisTemplate.boundValueOps(stockRedisKey).
15             decrement(orderCount);
16         if(stockLevel < 0) {
17             stockLevel = redisTemplate.boundValueOps(stockRedisKey).increment
18                 (orderCount);
19             if(stockLevel <= 0){
20                 caffeineCache.put(cacheKey, 0);
21             }
22         }
23     }
24 }

```

2.6.4 Kafka 与消息队列

在微服务构架中，我们通常会将服务分拆到多个服务器上，以方便运维和负载分离。此时，不同服务器间的通讯和协同工作就成为了一个重要的问题。由于不同服务的处理速率差异，对于一个特定的请求，不同服务能够接受并开始处理的时刻也各不相同。若直接让上一个服务访问下一个服务，则会造成大量的等待时间。更进一步的，若请求响应链中的其中一个服务宕机，则会造成整个响应过程卡死，造成请求响应失败。这时候，我们便可以使用消息队列（Message Queue）机制来解决这一问题。消息队列提供了一个公共的状态空间，使得与其关联的每个服务都能够向其中发送/从其中获取消息。当响应链中的每个服务完成处理后，便向消息队列发送一条消息，并标识需要接受这条消息的服务。当下游服务完成了其他请求后，便可以从消息队列中拉取相应的消息并继续完成请求处理。这样，整个响应就可以以异步的方式完成，减少服务处理的等待时间。

Kafka 是由 Apache 软件基金会开发的一个开源流处理平台，其基于 Zookeeper 实现的分片式中介服务可以很好的作为一个消息队列来使用。Kafka 的工作原理如下图所示（Figure 46）。在一个带有 Kafka 消息队列的系统中，服务被分为生产者和消费者。生产者通过 **Push** 操作向 Kafka 的特定话题（Topic）发送消息，消费者通过 **Pull** 操作从对应的话题中拉取消息。其中，每个 Kafka 协调器都被维护在一个 Kafka 容器中，并被统一注册到一个 Zookeeper 集群中。这样分布式的管理方式使得 Kafka 拥有很强的伸缩性和极高的消息吞吐能力。

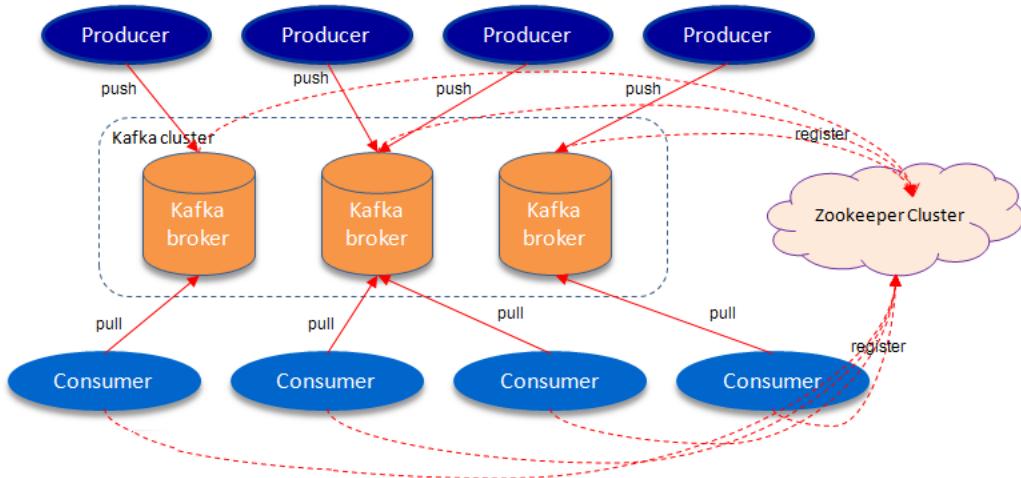


Figure 46: Kafka 消息队列的一般工作原理 [Kum17]

集成了消息队列后，一个下单请求的完整处理流程如下图所示 (Figure 47)，这也是本项目最终所采用的下单请求处理逻辑 (付款请求使用了同样的逻辑)。请求到达后，系统会通过多级缓存对请求进行一系列判断，并将无效请求快速过滤。随后，对于缓存命中的合法请求，系统会利用 Redis 缓存中的信息快速创建订单，并将订单的信息放置到 Kafka 消息队列中。至此，下单请求的处理已全部完成，服务会直接将订单号快速返回给用户。与此同时，一个异步的后台进程会从消息队列中抽取下单成功的消息，并继续接管这一处理流程，将订单信息写入主数据库中。这一分离式的服务处理流程构成了本系统能够应对高并发请求的重要基础。

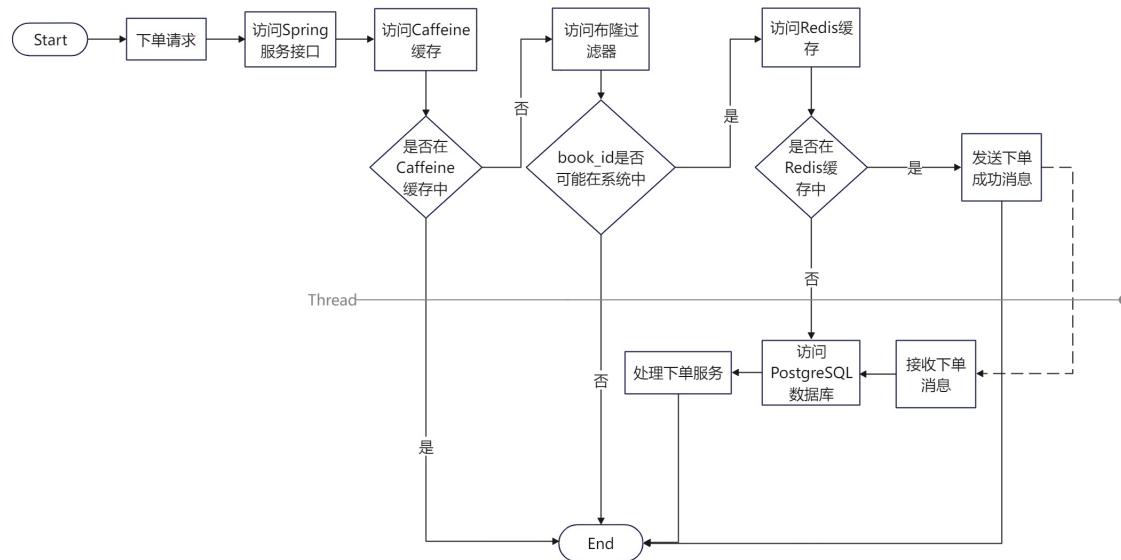


Figure 47: 使用了消息队列后用户下单的服务响应流程图

在本项目中，Kafka 的相关配置如下：(application.yml)

```

1 kafka:
2     bootstrap-servers: 127.0.0.1:9092
3     producer:
4         key-serializer: org.apache.kafka.common.serialization.StringSerializer
5         value-serializer: org.apache.kafka.common.serialization.
6             StringSerializer
7     acks: 1
8     consumer:
9         key-deserializer: org.apache.kafka.common.serialization.
10            StringDeserializer
11         value-deserializer: org.apache.kafka.common.serialization.
12             StringDeserializer
13         enable-auto-commit: false
14         auto-offset-reset: earliest
15     listener:
16         concurrency: 5
17         ack-mode: manual_immediate
18         missing-topics-fatal: false

```

生产者发送订单消息的实现如下：

```

1 @Component
2 public class KafkaProducer {
3     //Irrelevant codes
4     public static final String topicOrder = "topic.order";
5     /* 发送下单消息 */
6     public boolean sendOrder(Order order, List<OrderBook> orderBooks){
7         Map<String, Object> orderMap = new HashMap<>();
8         orderMap.put("order", order);
9         orderMap.put("books", orderBooks);
10        ObjectMapper jsonMapper = new ObjectMapper();
11        String orderStr;
12        try{
13            orderStr = jsonMapper.writeValueAsString(orderMap);
14        }
15        catch (JsonProcessingException e){
16            System.out.println("JSON字符串创建失败！");
17            e.printStackTrace();
18            return false;
19        }
20        ListenableFuture<SendResult<String, Object>> future = kafkaTemplate.
21            send(topicOrder, orderStr);
22        future.addCallback(new ListenableFutureCallback<SendResult<String,
23            Object>>() {
24            @Override
25            public void onFailure(Throwable exception) {
26                System.out.println("消息队列发送失败：" + exception.
27                    getMessage());
28            }
29
27            @Override
28            public void onSuccess(SendResult<String, Object> result) {
29                System.out.println("消息队列发送成功：" + result.toString());

```

```

30         }
31     });
32     return true;
33 }
34 /* 发送付款消息 */
35 public boolean sendPayment(String userID, String orderID){
36     Map<String, String> paymentMap = new HashMap<>();
37     paymentMap.put("payment_user", userID);
38     paymentMap.put("payment_order", orderID);
39     ObjectMapper jsonMapper = new ObjectMapper();
40     String paymentStr;
41     try{
42         paymentStr = jsonMapper.writeValueAsString(paymentMap);
43     }
44     catch (JsonProcessingException e){
45         System.out.println("JSON字符串创建失败! ");
46         e.printStackTrace();
47         return false;
48     }
49     ListenableFuture<SendResult<String, Object>> future = kafkaTemplate.
50         send(topicOrder, paymentStr);
51     future.addCallback(new ListenableFutureCallback<SendResult<String,
52         Object>>(){
53         @Override
54         public void onFailure(Throwable exception) {
55             System.out.println("消息队列发送失败: " + exception.
56                 getMessage());
57         }
58         @Override
59         public void onSuccess(SendResult<String, Object> result) {
60             System.out.println("消息队列发送成功: " + result.toString());
61         }
62     });
63     return true;
64 }

```

消费者实现下单和付款请求落库操作的实现代码如下：

```

1 @Component
2 public class KafkaConsumer {
3     //Irrelevant codes
4     @KafkaListener(topics = KafkaProducer.topicOrder, groupId = KafkaProducer
5         .topicOrder)
6     public void receiveOrder(ConsumerRecord<?, ?> record, Acknowledgment ack,
7         @Header(KafkaHeaders.RECEIVED_TOPIC) String topic){
8         Optional message = Optional.ofNullable(record.value());
9         if(message.isPresent()){
10             Object payload = message.get();
11             ObjectMapper mapper = new ObjectMapper();
12             Map<String, Object> messageMap;
13             try {
14                 messageMap = mapper.readValue((String) payload, new

```

```
13     TypeReference<Map<String, Object>>() {});
14     if(messageMap.containsKey("order")) {
15         Object orderRaw = messageMap.get("order");
16         Object booksRaw = messageMap.get("books");
17         Order order = mapper.convertValue(orderRaw, new
18             TypeReference<Order>() {
19         });
20         List<OrderBook> books = mapper.convertValue(booksRaw, new
21             TypeReference<List<OrderBook>>() {
22         });
23         orderMapper.insert(order);
24         for (OrderBook orderBook : books) {
25             bookMapper.update(
26                 null, new LambdaUpdateWrapper<Book>()
27                     .eq(Book::getInfoId, orderBook.
28                         getBookInfoId())
29                     .eq(Book::getStoreName, order.
30                         getStoreName())
31                     .setSql("stock_level = stock_level -
32                         " + orderBook.getCount())
33             );
34             orderBookMapper.insert(orderBook);
35         }
36         System.out.println("下单消息处理成功！订单号: " + order.
37             getUuid());
38     }
39     else if(messageMap.containsKey("payment_order")){
40         String orderID = (String) messageMap.get("payment_order")
41             ;
42         String userID = (String) messageMap.get("payment_user");
43         Order order = orderMapper.selectOne(
44             new LambdaUpdateWrapper<Order>()
45                 .eq(Order::getUuid, orderID));
46         userMapper.update(
47             null, new LambdaUpdateWrapper<User>()
48                 .eq(User::getName, userID)
49                 .setSql("money = money - " + order.
50                     getPrice())
51             );
52         orderMapper.update(
53             null, new LambdaUpdateWrapper<Order>()
54                 .eq(Order::getUuid, orderID)
55                 .set(Order::getStatus, 1)
56         );
57         System.out.println("付款消息处理成功！订单号: " + order.
58             getUuid());
59     }
60     catch (JsonProcessingException e){
61         e.printStackTrace();
62     }
63     ack.acknowledge();
64 }
```

```

56     }
57 }

```

下单服务的时序如下图所示 (Figure 48)。可以看到，服务在进行完必要的验证后，通过 KafkaProducer 向消息队列发送信息后便直接返回结果，无需等待落库操作完成再结束响应。

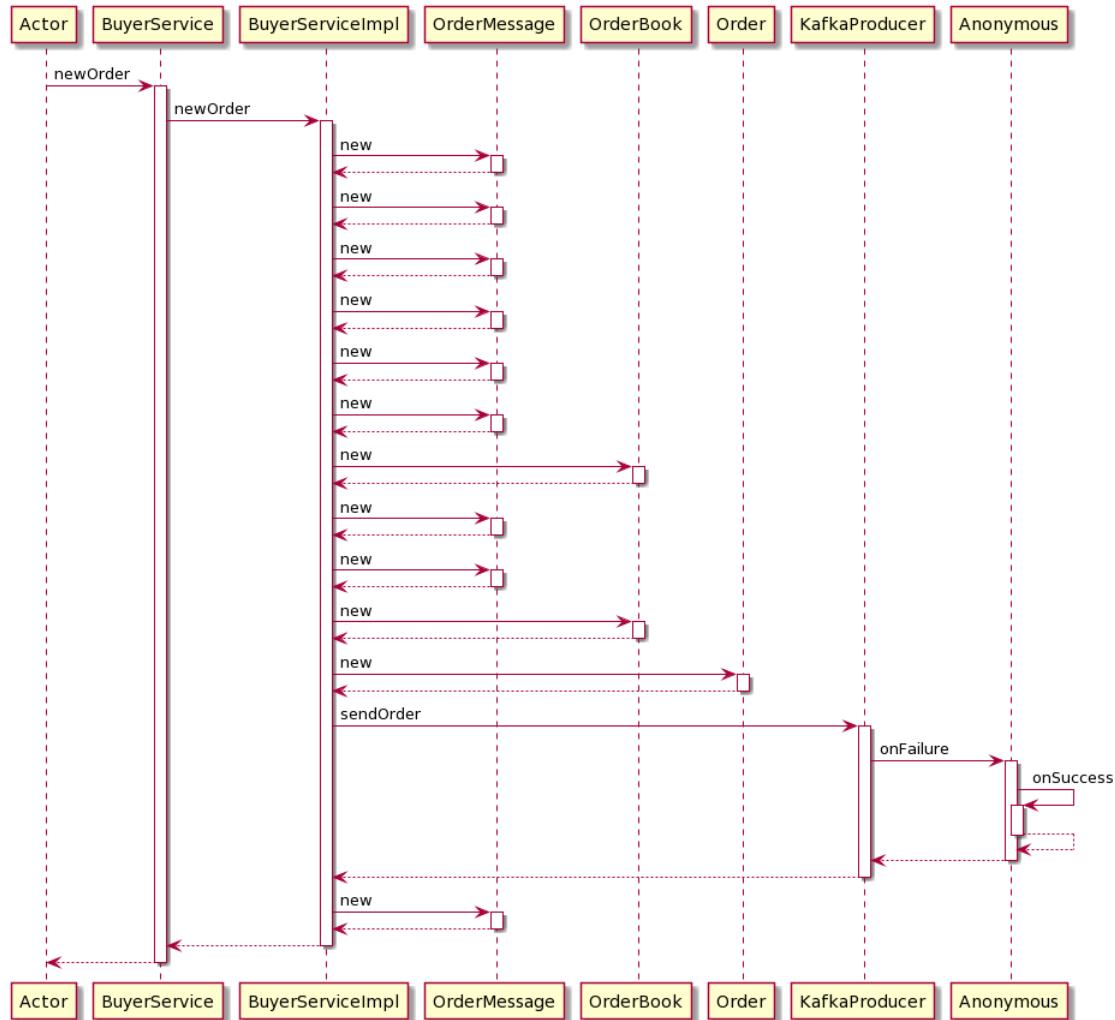


Figure 48: new_order 接口的服务层时序图

2.6.5 Nginx 与负载均衡

通过上面的各种缓存和优化技术，我们已经基本解决了高并发请求处理中数据库访问的性能瓶颈。然而，由于服务本身也依托于服务器主机的性能之上，当请求规模扩大到一定数量时，服务本身的处理性能就会成为瓶颈。对于服务的性能瓶颈问题，我们通常有两种处理方法：**向上扩展 (Scale Up)** 和 **向外扩展 (Scale Out)**。其中，前者意味着通过升级硬件等方式提升单台主机的性能，而后者则采用增加处理节点的方式来提升整个系统的处理能力。在当前技术工艺的限制下，我们很难通过 Scale Up 来从根本上提升系统的性能潜能，因此我们往往会采用 Scale

Out 的方式来对服务进行扩展。

当我们在多个节点上部署同一个服务后，我们就可以将接受到的请求分发到这些节点上进行并行处理。为此，我们就需要引出所谓的负载均衡（Load Balancing）技术。

当前，学术界和工业界都提出了一系列负载均衡的实现方式，其中最常用的便是同样被我们用作前端维生容器的 Nginx。Nginx 可以通过一种名为反向代理（Reverse Proxy）的方式来实现负载均衡，其基本原理如下图所示（Figure 49）。

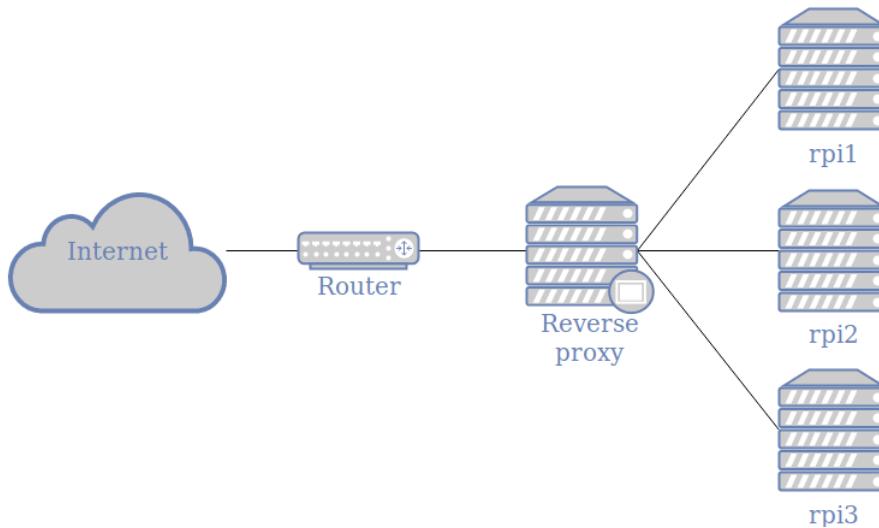


Figure 49: Nginx 反向代理过程

在我们的项目中，Nginx 负载均衡的配置如下：

```

1  worker_processes  1;
2  events {
3      worker_connections  1024;
4  }
5  http {
6      upstream  bookstore-server {
7          least_conn;
8          server      localhost:8081;
9          server      localhost:8082;
10     }
11     server {
12         listen      8080;
13         server_name  localhost;
14         location / {
15             proxy_pass http://bookstore-server;
16             proxy_redirect default;
17         }
18     }
19 }
```

其中，我们让 Nginx 监听 `localhost:8080` 端口的请求，并使其分配到位于 `8081` 和 `8082` 端口的两个服务上。对于负载均衡策略，我们使用了最小连接数（Least Connection）的方式

来均摊所有客户端发起的请求。

3 项目部署

本项目采用了前后端分离架构进行开发。要在生产环境部署本项目，目标系统上至少需要安装的依赖环境及其版本如下表所示 (Table 5)。

环境名称	版本
Java SE Development Kit	≥ 11.0
Nginx	1.21.1 (开发环境使用)
PostgreSQL	≥ 14.0
MongoDB	≥ 5.0
Redis	$\geq 6.0.0$
Zookeeper	3.7.0 (开发环境使用)
Kafka	2.8.0 (开发环境使用)
ElasticSearch	7.15.2 (需安装 Analyze IK 分词组件)

Table 5: 书店系统生产环境部署的依赖环境

若要从源码部署前端项目，你至少需要一个包含 `quasar/cli@1.2.1` 模块的 **NodeJS** 开发环境。首先进入仓库的 `frontend` 目录，使用 `quasar build` 对项目进行打包。随后，前端项目的静态资源会生成在项目的 `frontend/dist/spa` 目录下。之后，你需要一个能够支持前端资源的服务容器，在这里我们使用了 **Nginx** 作为前端维生容器。一个部署完成的 Nginx 前端环境运行状态如下图所示 (Figure 50)。Nginx 的默认服务端口为 8080 (可能与后端冲突，可自由更改)，我们只需使用浏览器访问该地址即可。

```
[root@aquarius2 ~]# systemctl status nginx.service
● nginx.service - LSB: starts the nginx web server
  Loaded: loaded (/etc/rc.d/init.d/nginx; bad; vendor preset: disabled)
  Active: active (running) since Fri 2021-11-19 15:01:13 CST; 4 weeks 2 days ago
    Docs: man:systemd-sysv-generator(8)
  CGroup: /system.slice/nginx.service
          └─1165 nginx: master process /www/server/nginx/sbin/nginx -c /www/server/nginx/conf/nginx.co...
              ├─14678 nginx: worker process
              ├─14679 nginx: cache manager process
Warning: Journal has been rotated since unit was started. Log output is incomplete or unavailable.
```

Figure 50: 部署完成的 Nginx 前端容器运行状态

后端的部署方式分为 **JAR** 包部署和 **WAR** 包部署两种方式。Springboot 本身包含了一个 **Tomcat** 服务容器，若使用前一种方式部署，框架就会自动将一个 Tomcat 容器集成至最终的 JAR 包中。此时，我们只需要在生产环境安装相应的 JAVA 运行环境即可。若想要在生产环境下使用自定义的 Tomcat 或其他服务容器，则可以选择使用 WAR 包的方式进行部署。

就参数而言，后端服务的默认端口为 **8080**；PostgreSQL 连接地址为 **127.0.0.1:5432**，用户名和密码均为 **postgres**，系统中应存在名称为 **bookstore** 的数据库；MongoDB 连接地址为 **127.0.0.1:27017**；Redis 连接地址为 **127.0.0.1:6379**；Kafka 连接地址为 **127.0.0.1:9092**；ElasticSearch 连接地址为 **127.0.0.1:9200**。如对其中任意参数进行修改，则需要重编译后端项目。

4 项目测试

4.1 接口测试

4.1.1 单元测试规范

单元测试能够很好的保证代码质量，而好的单元测试必须遵守 AIR 原则，即自动化 (Automatic)、独立性 (Independent) 和可重复 (Repeatable)。

- 自动化: 单元测试应该是全自动执行的，并且非交互式的。测试用例通常是被定期执行的，执行过程必须完全自动化才有意义。
- 独立性: 保持单元测试的独立性。为了保证单元测试稳定可靠且便于维护，单元测试用例之间决不能互相调用，也不能依赖执行的先后次序。
- 可重复: 单元测试是可以重复执行的，不能受到外界环境的影响。

本项目中的单元测试均遵守 AIR 原则。

4.1.2 JUnit5

JUnit5 是 java 的单元测试框架，于 2017 年推出，具有众多优点：

- 提供全新的断言和测试注解，支持测试类内嵌。
- 更丰富的测试方式：支持动态测试，重复测试，参数化测试等。
- 实现了模块化，让测试执行和测试发现等不同模块解耦，减少依赖。
- 提供对 Java 8 的支持，如 Lambda 表达式，Stream API 等。

本项目中的单元测试均使用 JUnit5 进行。

4.1.3 基础接口测试代码移植

对于基础接口的 python 测试代码，本项目使用 JUnit5 框架对其进行重写（仅改变语言，并未修改测试的实际内容），使其能够在 SpringBoot 框架下运行，以下是一个例子。

```

1 // 重写之前的 TestRegister
2 class TestRegister:
3     @pytest.fixture(autouse=True)
4     def pre_run_initialization(self):
5         self.user_id = "test_register_user_{}".format(time.time())
6         self.password = "test_register_password_{}".format(time.time())
7         self.auth = auth.Auth(conf.URL)
8         yield
9
10    def test_register_ok(self):
11        code = self.auth.register(self.user_id, self.password)
12        assert code == 200
13
14    def test_unregister_ok(self):
15        code = self.auth.register(self.user_id, self.password)
16        assert code == 200
17

```

```
18     code = self.auth.unregister(self.user_id, self.password)
19     assert code == 200
20     // ...
21
22 // 重写之后的 TestRegister
23 @Nested
24 @TestInstance(TestInstance.Lifecycle.PER_CLASS)
25 class TestRegister{
26     String name="test_register_user";
27     String password="test_register_password";
28
29     @Test
30     void test_register_ok() throws Exception {
31         // given
32         RegisterBody registerBody=new RegisterBody(
33             getStringWithUUID(name),
34             getStringWithUUID(password)
35         );
36         // when/then
37         iMockMvc.testPost(baseUrl+ "/register",registerBody,ok, Message.class)
38             ;
39     }
40
41     @Test
42     void test_unregister_ok() throws Exception {
43         // given
44         RegisterBody registerBody=new RegisterBody(
45             getStringWithUUID(name),
46             getStringWithUUID(password)
47         );
48         // when/then
49         // 注册
50         iMockMvc.testPost(baseUrl+ "/register",registerBody,ok,Message.class);
51         // 注销
52         iMockMvc.testPost(baseUrl+ "/unregister",registerBody,ok,Message.class
53             );
54     }
55     // ...
```

4.1.4 基础接口测试

基础接口测试共计 32 个，全部通过。

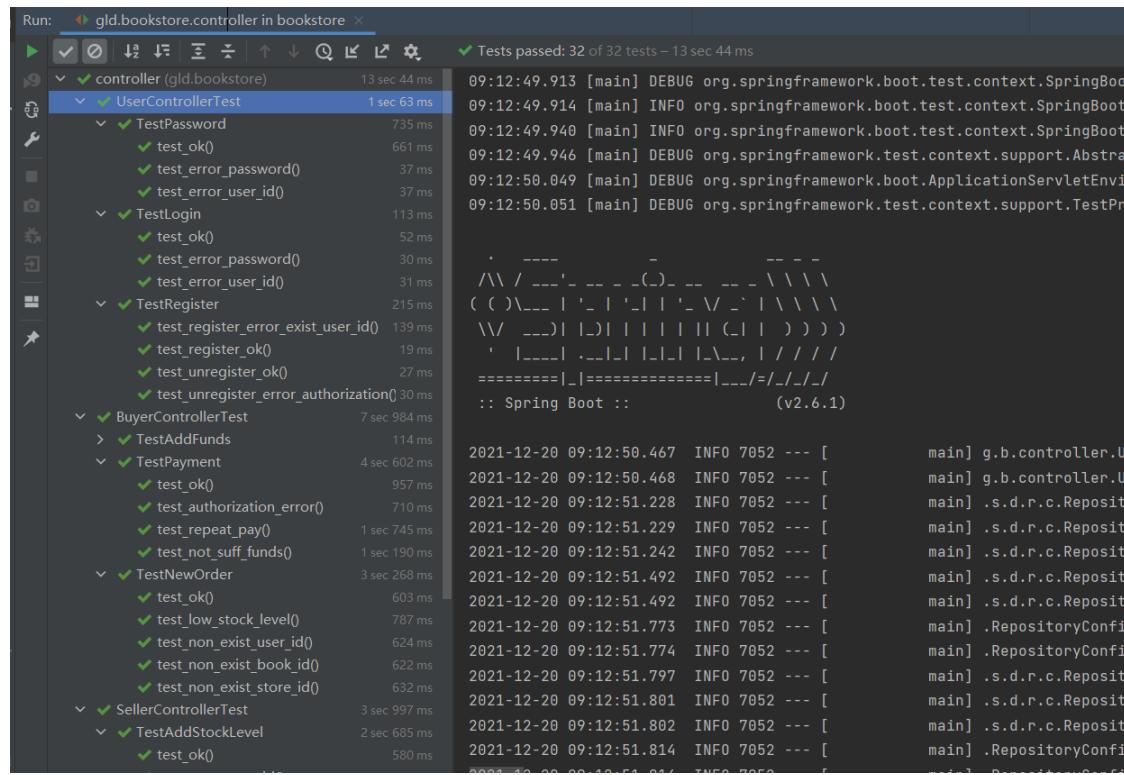


Figure 51: 基础接口测试结果

4.1.5 额外接口测试

额外接口测试 37 个, 总共 69 个接口测试, 全部通过。

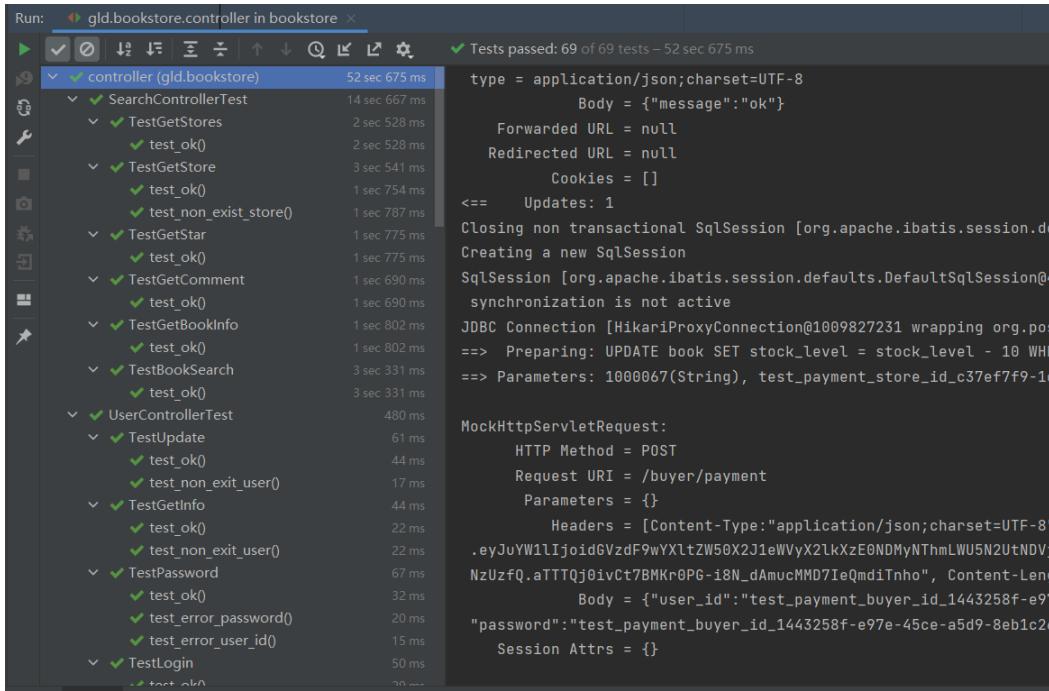


Figure 52: 所有接口测试结果

4.2 代码覆盖率

我们使用了 IntelliJ IDEA 开发工具中自带的 **Run with Coverage** 功能记录了运行全部测试时项目代码的覆盖率，结果如下图所示（Figure 53）。可以看到，我们编写的全部测试可以使类覆盖率达到 **98.7%**，方法覆盖率达到 **91.7%**，行覆盖率达到 **90.3%**。经过分析我们还可以发现，绝大多数未覆盖的代码均为构造函数和形如 **try...catch...** 这样的异常处理函数，这些函数保证了我们的系统能够应对各种突发状况和边界情况，因此其存在于代码中是必要的。若只考虑实际功能段的代码，覆盖率将可得到进一步的提升。

Overall Coverage Summary			
Package	Class, %	Method, %	Line, %
all classes	98.7% (74/75)	91.7% (407/444)	90.3% (1056/1170)
Coverage Breakdown			
Package	Class, %	Method, %	Line, %
gld.bookstore	100% (1/1)	50% (1/2)	33.3% (1/3)
gld.bookstore.config	100% (6/6)	100% (12/12)	100% (39/39)
gld.bookstore.controller	100% (5/5)	97.5% (39/40)	97.8% (45/46)
gld.bookstore.controller.dto	100% (18/18)	100% (100/100)	100% (100/100)
gld.bookstore.controller.vo	100% (18/18)	91.5% (108/118)	93.8% (150/160)
gld.bookstore.entity	100% (10/10)	85.6% (83/97)	86.9% (93/107)
gld.bookstore.interceptor	100% (1/1)	100% (2/2)	100% (12/12)
gld.bookstore.mq	100% (7/7)	80% (12/15)	82.6% (71/86)
gld.bookstore.service.impl	100% (6/6)	92% (46/50)	88.7% (534/602)
gld.bookstore.utils	66.7% (2/3)	50% (4/8)	73.3% (11/15)

Figure 53: 代码覆盖率

4.3 吞吐量测试

我们使用实验要求中给出的测试工具对所实现的系统进行吞吐量测试(fe/test/test_bench.py)。在 2020 版 13 英寸 Macbook Pro 上 (2GHz 4 核 Intel Core i5 + 16GB LPDDR4X) 上, 下单和付款接口吞吐量的测试结果如下图所示 (Figure 54)。可以看到, 系统的吞吐量可以达到 **83000** 左右, 延迟可以维持在 **0.008** 左右。

```
TPS_C=82836, NO=OK:487036 Thread_num:991 TOTAL:487036 LATENCY:0.008112128345204963 ,
TPS_C=82927, NO=OK:488028 Thread_num:992 TOTAL:488028 LATENCY:0.008111309941483073 ,
TPS_C=83017, NO=OK:489021 Thread_num:993 TOTAL:489021 LATENCY:0.008110490235768264 ,
TPS_C=83108, NO=OK:490015 Thread_num:994 TOTAL:490015 LATENCY:0.008109669743117685 ,
TPS_C=83198, NO=OK:491010 Thread_num:995 TOTAL:491010 LATENCY:0.008108854646345367 ,
TPS_C=83289, NO=OK:492006 Thread_num:996 TOTAL:492006 LATENCY:0.008108040765278279 ,
TPS_C=83379, NO=OK:493003 Thread_num:997 TOTAL:493003 LATENCY:0.00810722629413884 ,
TPS_C=83470, NO=OK:494001 Thread_num:998 TOTAL:494001 LATENCY:0.008106412191257266 ,
TPS_C=83561, NO=OK:495000 Thread_num:999 TOTAL:495000 LATENCY:0.008105600913365683 ,
TPS_C=83651, NO=OK:496000 Thread_num:1000 TOTAL:496000 LATENCY:0.008104790232354595
```

Figure 54: 使用实验要求中给出的测试程序对系统的吞吐量测试结果

然而, 在测试过程中我们发现, 实验要求中给出的这一测试程序实际上并不能正确给出两个接口的真实吞吐量。通过修改配置文件中 Request_Per_Session (fe/conf.py) 的值, 我们甚至可以测出超过 **130** 万的吞吐量 (Figure 55)。作为对比, 2020 年天猫双 11 购物节达到的最大流量洪峰也仅达到了 **583000** 笔交易/秒, 这一吞吐量已经给全面云原生化的阿里云造成了极大的压力, 因此测试程序给出的这一测试结果显然是不合理的。

```
TPS_C=1333236, NO=OK:2358103301 Thread_num:69228 TOTAL:2372314806 LATENCY:0.032470861104877125 ,
TPS_C=1333256, NO=OK:2358172124 Thread_num:69229 TOTAL:2372384035 LATENCY:0.032470869566174655 ,
TPS_C=1333275, NO=OK:2358240948 Thread_num:69230 TOTAL:2372453265 LATENCY:0.03247087801600118 ,
TPS_C=1333295, NO=OK:2358309773 Thread_num:69231 TOTAL:2372522496 LATENCY:0.03247088645486268 ,
TPS_C=1333315, NO=OK:2358378599 Thread_num:69232 TOTAL:2372591728 LATENCY:0.03247089488467359 ,
TPS_C=1333335, NO=OK:2358447426 Thread_num:69233 TOTAL:2372660961 LATENCY:0.032470903303642544 ,
TPS_C=1333354, NO=OK:2358516254 Thread_num:69234 TOTAL:2372730195 LATENCY:0.03247091171225691 ,
TPS_C=1333374, NO=OK:2358585083 Thread_num:69235 TOTAL:2372799430 LATENCY:0.032470920111700635 ,
```

Figure 55: 修改配置文件后使用测试程序对系统的吞吐量测试结果（不合理）

为了得到更为可信的结果，我们还是使用 **JMeter** 接口压测工具对系统进行多线程压测。测试的相关设置如下图所示 (Figure 56, 57)。我们使用 1000 个线程来模拟 1000 个用户同时下单时的场景，其中每个用户对同一本书进行 10 次下单操作，共 10000 次并发请求操作。

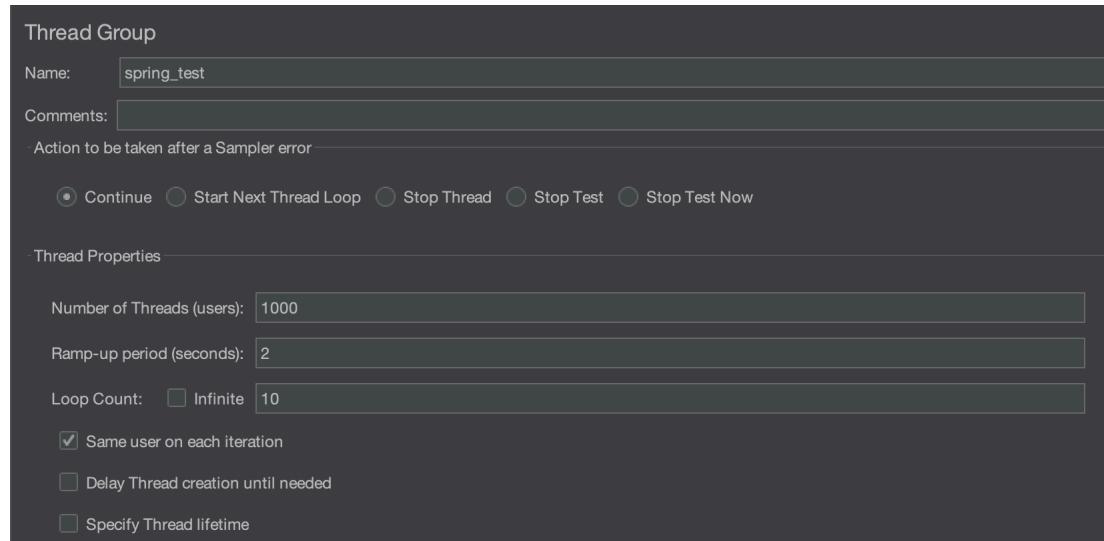


Figure 56: 对下单接口的 JMeter 压测设置 (线程组)

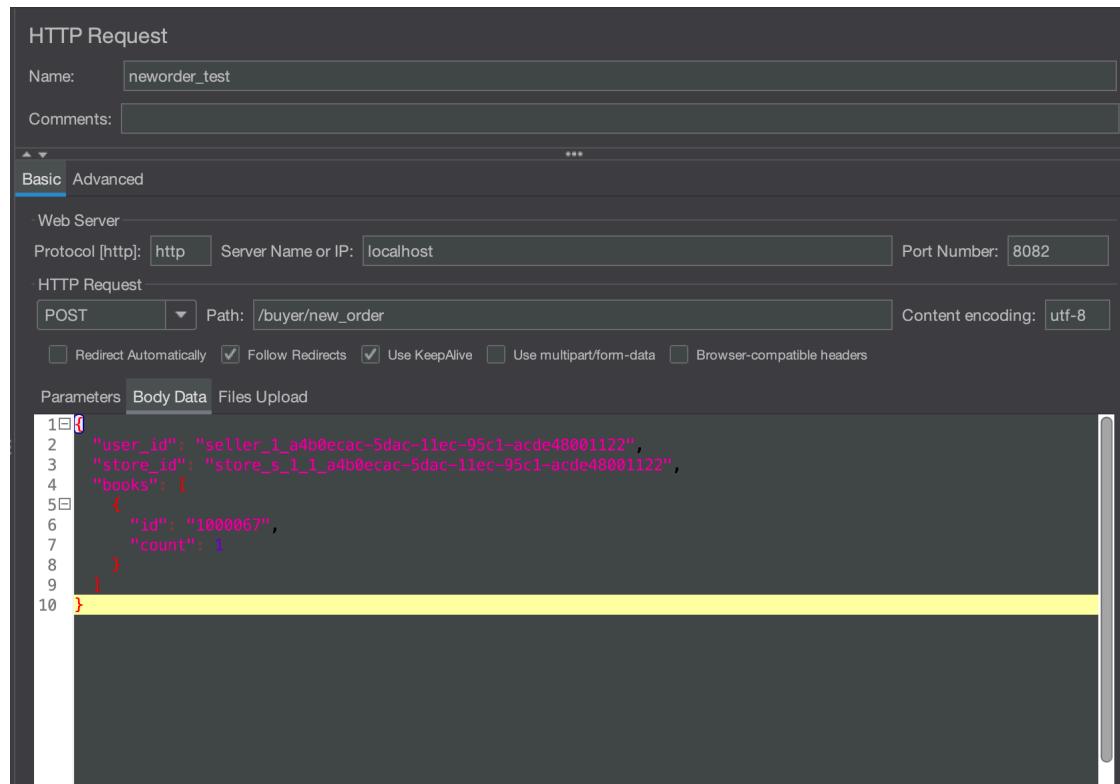


Figure 57: 对下单接口的 JMeter 压测设置 (请求体)

我们分别将请求的书本库存设置为 10000 和 100, 用于分别模拟平日里普通下单场景下 (商品库存足够时) 和活动促销时秒杀场景下 (商品库存远小于请求量) 的下单操作。经过测试, 在普通场景下, 系统的下单接口吞吐量可以达到 **1142 笔交易/秒 (68562 笔交易/分钟)** (Figure 58); 在秒杀场景下, 系统的吞吐量可以达到 **1972 笔交易/秒 (118320 笔交易/分钟)** (Figure 59)。这一结果远高于直接对数据库进行访问的朴素服务响应流程 (约 300 ~ 500 笔交易/秒), 极有力的表明了我们设计的一系列负载平衡措施起到了预期的效果。

Label	# 样本	平均值	中位数	90% 百...	95% 百...	99% 百...	最小值	最大值	异常 %	吞吐量	接收 K...	发送 K...
neworder...	10000	712	712	924	953	1094	13	1273	0.00%	1142.7/sec	362.68	652.83
TOTAL	10000	712	712	924	953	1094	13	1273	0.00%	1142.7/sec	362.68	652.83

Figure 58: 普通场景下下单接口的吞吐量测试结果 (请求数小于库存量)

Label	# 样本	平均值	中位数	90% 百...	95% 百...	99% 百...	最小值	最大值	异常 %	吞吐量	接收 K...	发送 K...
neworder...	10000	368	365	509	521	700	3	1130	99.00%	1972.0/sec	652.57	1126.58
TOTAL	10000	368	365	509	521	700	3	1130	99.00%	1972.0/sec	652.57	1126.58

Figure 59: 秒杀场景下下单接口的吞吐量测试结果 (请求数远大于库存量)

5 开发说明

5.1 版本控制

本项目的全部工作均在水杉码源提供的 GitLab 平台上完成，并使用了 Git 工具进行版本控制。开发过程中，项目成员共进行了 43 次提交，部分提交图如下图所示 (Figure 60)。

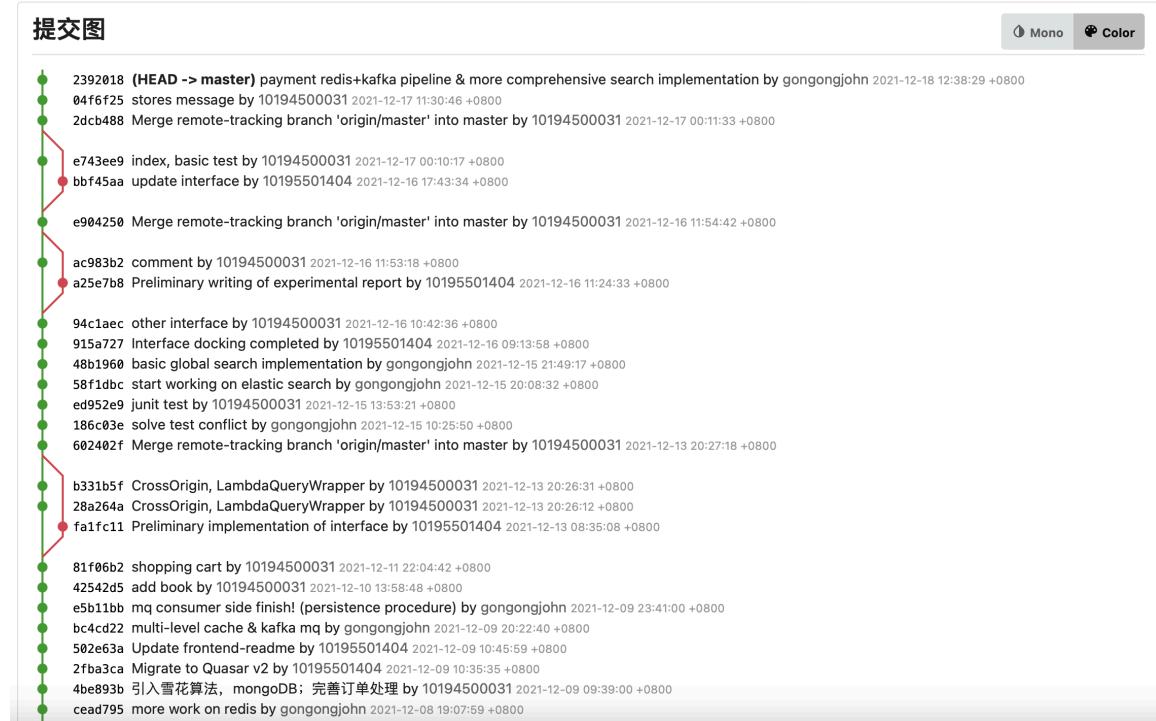


Figure 60: 书店项目的提交图

除此之外，我们还根据分工在项目仓库中建立了三个分支：**master**、**frontend**、**backend** (Figure 61)，方便成员集中开发自己模块上的功能。通过 Merge Request 功能，我们就能将分支中的更新合并到主分支上 (Figure 62)。

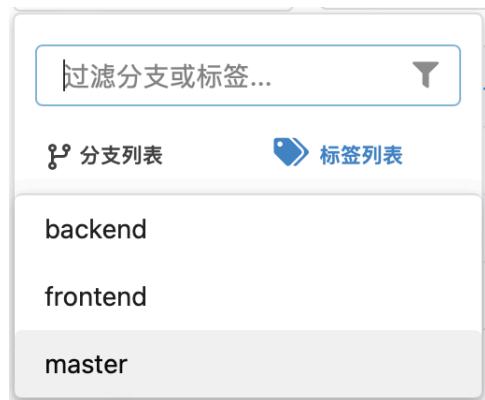


Figure 61: 书店项目的分支



Figure 62: 书店项目的 Merge Request

5.2 项目分工

本项目共有 3 位成员，其中龚敬洋（10195501436）为项目组长，段晗祈（10195501404）和刘明熹（10194500031）为项目成员。三位成员均摊了本项目的全部工作量，对本项目的贡献相当。具体的，三位项目成员在本项目中的主要贡献及负责的任务如下表所示（Table 6）。

成员名称	贡献占比	负责内容
龚敬洋	1/3	系统构架设计、数据库选型与调优、全文搜索、高可用设计与实现
段晗祈	1/3	ER 图与关系模式设计，接口设计，前端设计与开发
刘明熹	1/3	系统功能设计、后端设计与开发、接口测试

Table 6: 书店项目成员主要贡献分布

6 后记

至此，本项目的全部功能和实现过程已叙述完毕。本项目从设计、构思到完成全部开发周期超过两个月时间，仅后端源代码就超过 5000 行（Figure 63），合计源代码更是达到了超过 70 万行（Figure 64）。项目的部分构架参考了淘宝、当当网、Bilibili 等知名电商及多媒体服务平台的现役及历史构架，也使得我们得以对当今工业界一个完整电商平台的全开发流程有了一个全景式的了解。

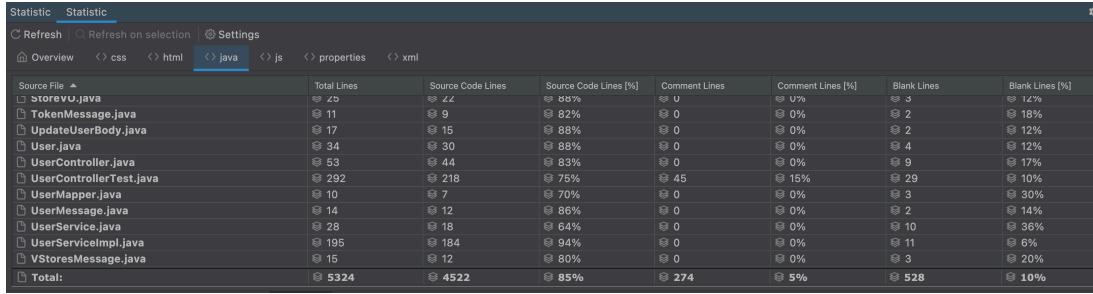


Figure 63

github.com/AlDanial/cloc v 1.90 T=130.34 s (126.0 files/s, 27882.9 lines/s)

Language	files	blank	comment	code
JavaScript	11243	163993	443311	2059724
JSON	1353	2881	0	217892
TypeScript	1760	15400	133072	157876
Markdown	1081	49819	2	129591
CSS	130	32713	368	115589
HTML	221	2828	1	44988
XML	297	164	0	30795
Sass	95	1912	50	15028
Java	89	528	274	4522
Vuejs Component	23	340	787	2685
YAML	89	99	30	1877
Bourne Shell	4	363	286	1805
CoffeeScript	10	364	29	874
diff	3	19	168	469
Windows Module Definition	5	83	0	451
SVG	3	1	1	191
EJS	7	13	0	174
DOS Batch	1	35	0	147
PHP	1	13	19	124
Maven	1	2	0	106
make	5	31	4	56
Bourne Again Shell	2	11	1	43
Nix	1	1	0	19
SUM:	16424	270813	578403	2784946

Figure 64

由于内容繁多，本报告的篇幅可能略显冗长。若您完整的阅读了全篇的报告，我们感到十分的荣幸和由衷的感谢。助教老师辛苦了！

References

- [Abu20] Ralf Abueg. Elasticsearch: What it is, how it works, and what it's used for. <https://www.knowi.com/blog/what-is-elasticsearch/>, 2020.
- [Aky16] Bugra Akyildiz. A gentle introduction to bloom filter. <https://www.kdnuggets.com/2016/08/gentle-introduction-bloom-filter.html>, 2016.
- [Eng21] DB Engines. Db-engines ranking - trend popularity. https://db-engines.com/en/ranking_trend, 2021.
- [Jai19] Ayush Jain. Redis vs. mysql benchmarks. <https://dzone.com/articles/redis-vs-mysql-benchmarks>, 2019.
- [kkl21] kkltgnv. [redis] cache penetration, cache breakdown, cache avalanches and solutions. <https://www.fatalerrors.org/a/redis-cache-penetration-cache-breakdown-cache-avalanches-and-solutions.html>, 2021.
- [Kum17] Mukesh Kumar. In-depth kafka message queue principles of high-reliability. https://medium.com/@mukeshkumar_46704/in-depth-kafka-message-queue-principles-of-high-reliability-42e464e66172, 2017.
- [lis12] liseri. 双十一淘宝、京东服务器瘫痪大揭秘. <https://blog.csdn.net/liseri/article/details/8236922>, 2012.
- [TB15] Andrew S Tanenbaum and Herbert Bos. *Modern operating systems*. Pearson, 2015.
- [Wik21a] Wikipedia contributors. High availability — Wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=High_availability&oldid=1058006194, 2021.
- [Wik21b] Wikipedia contributors. Redis — Wikipedia, the free encyclopedia. <https://en.wikipedia.org/w/index.php?title=Redis&oldid=1057935061>, 2021.
- [京 14] 京华时报. 12306 网站瘫痪原因揭秘: 平均每秒点击 24 万次. <https://news.sina.com.cn/c/2014-01-25/015929339247.shtml>, 2014.