**WiseInvest Test document**

**一、测试范围概述**

**1.1 项目结构说明**

WiseInvest is a financial investment platform that adopts a front-end and back-end separation along with a microservices architecture. This design provides greater flexibility, scalability, and maintainability. The platform consists of the following key modules:

* **Account Management**: Handles user registration, authentication, and account-related operations.
* **Product Management**: Manages the financial products available on the platform, including product details and availability.
* **Transaction Processing**: Supports the buying and selling of financial products, ensuring secure and efficient transactions.
* **Settlement Management**: Processes transaction settlements, including the calculation and distribution of funds.
* **Intelligent Recommendation**: Provides personalized investment recommendations based on user data and preferences.
* **AI Support**: Utilizes artificial intelligence to enhance user experience, such as through chatbots or predictive analytics.

The front-end is responsible for the user interface and experience, while the back-end handles business logic, data processing, and integration with external services. The microservices architecture allows each module to be developed, deployed, and scaled independently, improving the system's overall resilience and performance.

**1.2 测试类型覆盖**

To ensure the reliability, functionality, and performance of the WiseInvest platform, a comprehensive testing strategy has been implemented. This testing includes three major types:

* **Unit Testing**
* **Integration Testing**
* **Regression Testing**

Both the front-end and back-end of the system have undergone thorough testing across all the aforementioned modules to ensure complete validation.

**Unit Testing**

Unit testing involves independently testing individual components or functions to verify that they operate as expected. In the WiseInvest project, unit testing covers both the front-end and back-end:

* **Front-end Unit Testing**: Focuses on testing individual UI components (such as buttons, forms, and modals) as well as functions that manage user interactions and state.
* **Back-end Unit Testing**: Targets the independent functions of each microservice, such as API endpoints, data processing logic, and database interactions.

Through unit testing, we ensure that each part of the system functions correctly in isolation, which is crucial for identifying and fixing issues early in the development process.

**Integration Testing**

Integration testing verifies that different components or modules of the application work together seamlessly. Given WiseInvest's microservices architecture, integration testing is essential to ensure communication between services and overall system functionality:

* **Front-end Integration Testing**: Ensures that the user interface interacts correctly with back-end APIs, including data retrieval, submission, and real-time updates.
* **Back-end Integration Testing**: Tests the interactions between different microservices, such as the collaboration between account management and transaction processing, and ensures data consistency across services.

Integration testing helps identify issues that may arise from the interaction between components, which might not be evident during unit testing.

**Regression Testing**

Regression testing ensures that new changes or updates to the system do not introduce new bugs or disrupt existing functionality. In WiseInvest, regression testing is conducted after every major change or release:

* **Front-end Regression Testing**: Verifies that updates to the user interface or new features do not negatively impact the existing user experience or functionality.
* **Back-end Regression Testing**: Ensures that modifications to microservices or database schemas do not interfere with the overall system performance or data integrity.

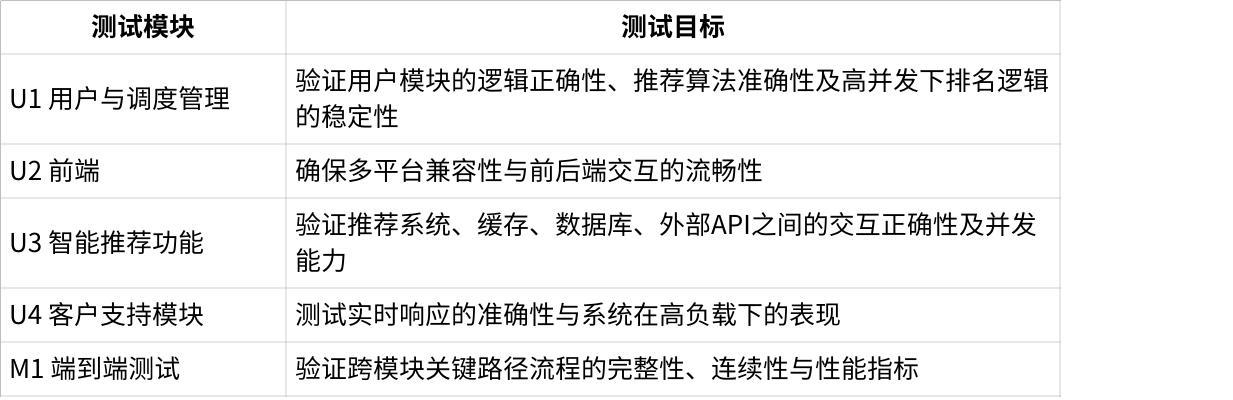
Regression testing is vital for maintaining the stability and reliability of the platform as it evolves over time.

**二、测试计划**

**2.1 Test Framework 测试框架**

本项目采用“核心+辅助”的测试策略。核心测试采用单元测试和集成测试框架（如xUnit、unittest），辅助测试包括Postman API测试、MongoDB数据验证、性能测试工具（如NBomber）等。测试将结合自动化脚本与人工验证，确保平台功能、性能与安全性达标。

**2.2 Test Levels, Targets and Strategy 测试层级、目标与策略**



**点击图片可查看完整电子表格**

**优先测试高风险功能**（如用户认证、资金交易、推荐系统）

**黑盒测试为主**，白盒测试作为补充（如算法、接口内部逻辑）

多测试套件覆盖关键路径，提升问题发现率

**不包含纯UI展示类测试**，由用户验收环节补足

**2.3 High-Level Test Suite Design 高级测试套件设计**

**U1 用户与调度管理套件**

* 技术栈：xUnit + Selenium + NBomber
* 验收标准：
* 登录/注册认证流程正确
* 推荐算法准确率 ≥85%
* 接口在100+并发下响应成功率 ≥95%

**U2 前端套件**

* 技术栈：Cypress + BrowserStack + LinkChecker
* 验收标准：
* 页面在主流浏览器和终端上无渲染问题
* 所有链接有效，页面加载时间 <2s

**U3 智能推荐系统套件**

* 技术栈：unittest + Postman + MongoDB + Redis + NBomber
* 验收标准：
* 推荐结果正确，Redis缓存按时失效
* Mongo加密存储，接口在高压下稳定运行

**U4 客户支持服务测试**

* 技术栈：自定义日志分析工具 + NBomber + 人工验证
* 验收标准：
* 客服响应时间 <2s
* 问题工单识别和响应准确率 ≥90%

**M1 端到端测试套件**

* 技术栈：Selenium + Backend Hooks
* 验收标准：
* 用户完成一次完整交易流程（登录→选择产品→下单→查看结果）不出错
* 所有跨模块接口响应均符合预期

**2.4 Risk-Based Testing 风险驱动测试**



**点击图片可查看完整电子表格**

**2.5 Cost Estimation 成本估算**

模型：COCOMO II

代码规模：约15KLOC，功能点15

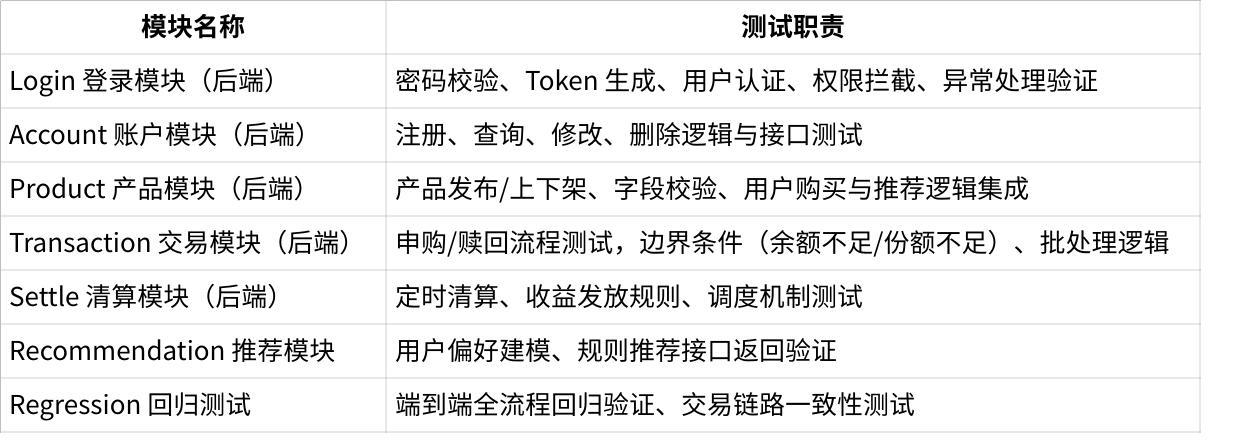
测试团队人数：4人

项目周期：3个月

预估测试成本：￥75,000（人力+工具+服务器使用）

**2.6 Team Members’ Responsibilities 团队分工**

闫雯晴（Yan Wenqing）



**点击图片可查看完整电子表格**

关咏颖（Guan Yongying）



**点击图片可查看完整电子表格**

杨博萱（Yang Boxuan）



**点击图片可查看完整电子表格**

韩明洋（Han Mingyang）



**点击图片可查看完整电子表格**

**三、后端模块测试（BE）**

**3.1 测试方法**

所有后端模块均采用 JUnit 进行单元测试，使用 SpringBoot Test 进行集成测试。覆盖接口逻辑、数据流转、异常处理与模块间交互等方面。

* Spring Boot Test + MockMvc + Mockito
* 数据传输对象使用 SubscriptionDTO 与 RedemptionDTO
* 外部接口通过 @MockBean 模拟行为（如 AccountClient、ProductClient）

**3.2 Login Module**

**测试目标**  
 验证用户登录模块的功能正确性，包括密码校验、账号合法性验证、密码修改等操作的处理逻辑是否符合预期。

**测试用例 1：login\_shouldSucceed\_whenCustomerCredentialsCorrect**

* 描述：客户使用正确的手机号和密码登录，应成功返回 JWT token
* 测试方法：Mock loginService.checkPassword 返回 token 字符串
* 预期结果：返回状态码 200，业务码为 0，返回 token，提示“操作成功”
* 实际结果：与预期一致，客户登录成功逻辑正确

**测试用例 2：login\_shouldFail\_whenPasswordIncorrect**

* 描述：客户输入错误密码，应提示“密码错误”
* 测试方法：Mock loginService.checkPassword 返回“密码错误”
* 预期结果：返回状态码 200，业务码为 1，提示“密码错误”
* 实际结果：与预期一致，密码校验逻辑生效

**测试用例 3：login\_shouldFail\_whenCustomerNotExist**

* 描述：手机号未开户，应提示“该手机号暂未开户”
* 测试方法：Mock customerMapper.selectOne 返回 null
* 预期结果：返回状态码 200，业务码为 1，提示“该手机号暂未开户”
* 实际结果：与预期一致，账号不存在判断正确

**测试用例 4：login\_shouldSucceed\_whenAdminCredentialsCorrect**

* 描述：管理员使用正确账号密码登录，应返回 JWT
* 测试方法：Mock adminMapper.selectOne + checkPassword，返回 token
* 预期结果：返回状态码 200，业务码为 0，提示“操作成功”
* 实际结果：与预期一致，管理员登录成功

**测试用例 5：login\_shouldFail\_whenAdminNotExist**

* 描述：管理员手机号未注册，应提示“该手机号暂未注册管理员”
* 测试方法：Mock adminMapper.selectOne 返回 null
* 预期结果：返回状态码 200，业务码为 1，提示“该手机号暂未注册管理员”
* 实际结果：与预期一致

**测试用例 6：login\_shouldFail\_whenUserTypeInvalid**

* 描述：用户类型非法（非1或2）时，应提示“非法用户类型”
* 测试方法：传入非法 userType=3
* 预期结果：返回状态码 200，业务码为 1，提示“非法用户类型”
* 实际结果：与预期一致

**测试用例 7：verifyCustomerPhone\_shouldSucceed\_whenPhoneExists**

* 描述：验证已注册客户手机号，应返回成功
* 测试方法：Mock loginService.isVaildPhoneNumber 返回 true
* 预期结果：返回状态码 200，业务码为 0，提示“操作成功”
* 实际结果：与预期一致

**测试用例 8：verifyCustomerPhone\_shouldFail\_whenPhoneNotExist**

* 描述：验证未注册客户手机号，应提示“手机号没有匹配的账号”
* 测试方法：Mock loginService.isVaildPhoneNumber 返回 false
* 预期结果：返回状态码 200，业务码为 1，提示“手机号没有匹配的账号”
* 实际结果：与预期一致

**测试用例 9：changePassword\_shouldSucceed\_whenValidCustomer**

* 描述：客户修改密码成功
* 测试方法：Mock loginService.changePassword 返回 true
* 预期结果：返回状态码 200，业务码为 0，提示“操作成功”
* 实际结果：与预期一致

**测试用例 10：changePassword\_shouldFail\_whenInvalidUserType**

* 描述：用户类型非法修改密码失败
* 测试方法：传入非法 userType=3，Mock 返回 false
* 预期结果：返回状态码 200，业务码为 1，提示“非法用户类型”
* 实际结果：与预期一致

**测试结论**  
 登录模块功能完整，覆盖了常见用户类型的身份校验、密码比对、手机号合法性验证及密码修改操作。测试中所有用例均通过，处理逻辑与预期一致，系统对非法输入具备良好鲁棒性。

**3.3 Account Management Module**

**测试目标**  
 验证用户账户管理模块的各项功能，包括开户、风险等级更新、客户信息更新、银行卡绑定/解绑、账户查询等核心业务逻辑的正确性和稳定性。

**测试用例 1：createAccount\_shouldSucceed\_whenNewCustomer**

* 描述：新客户开户成功，返回 fundAccount ID
* 测试方法：Mock customerService.createAccount 返回非 0 ID
* 预期结果：返回状态码 200，业务码为 0，返回 fundAccount
* 实际结果：与预期一致

**测试用例 2：createAccount\_shouldFail\_whenAccountAlreadyExists**

* 描述：客户手机号或身份证号已存在，开户失败
* 测试方法：Mock customerService.createAccount 返回 0
* 预期结果：返回状态码 200，业务码为 1，提示“该手机号或身份证号已开户”
* 实际结果：与预期一致

**测试用例 3：updateRiskLevel\_shouldSucceed\_whenValidAccount**

* 描述：更新客户风险等级成功
* 测试方法：Mock customerService.updateRiskLevel 返回 true
* 预期结果：返回状态码 200，业务码为 0，提示“操作成功”
* 实际结果：与预期一致

**测试用例 4：updateRiskLevel\_shouldFail\_whenInvalidAccount**

* 描述：更新客户风险等级失败（账号无效）
* 测试方法：Mock 返回 false
* 预期结果：返回状态码 200，业务码为 1，提示“更新失败”
* 实际结果：与预期一致

**测试用例 5：updateInfo\_shouldSucceed\_whenPhoneNotConflict**

* 描述：客户更新信息成功（手机号未被占用）
* 测试方法：Mock updateInfo 返回 true
* 预期结果：返回状态码 200，业务码为 0，提示“操作成功”
* 实际结果：与预期一致

**测试用例 6：updateInfo\_shouldFail\_whenPhoneConflict**

* 描述：更新手机号失败（手机号已被其他人使用）
* 测试方法：Mock updateInfo 返回 false
* 预期结果：返回状态码 200，业务码为 1，提示“该手机号已有客户使用”
* 实际结果：与预期一致

**测试用例 7：addBankcard\_shouldSucceed\_whenCardNotBound**

* 描述：添加银行卡成功，返回交易账户 ID
* 测试方法：Mock addBankcard 返回 ID 字符串
* 预期结果：返回状态码 200，业务码为 0，返回 tradingAccountId
* 实际结果：与预期一致

**测试用例 8：addBankcard\_shouldFail\_whenCardAlreadyBound**

* 描述：添加已绑定银行卡失败
* 测试方法：Mock 返回 “该卡号已有交易账户”
* 预期结果：返回状态码 200，业务码为 1，提示“该卡号已有交易账户”
* 实际结果：与预期一致

**测试用例 9：deleteBankcard\_shouldSucceed\_whenValidId**

* 描述：删除交易账户成功
* 测试方法：Mock deleteBankcard 返回 true
* 预期结果：返回状态码 200，业务码为 0，提示“操作成功”
* 实际结果：与预期一致

**测试用例 10：deleteBankcard\_shouldFail\_whenInvalidId**

* 描述：删除银行卡失败（不存在或数据库更新失败）
* 测试方法：Mock 返回 false
* 预期结果：返回状态码 200，业务码为 1，提示“删除失败”
* 实际结果：与预期一致

**测试用例 11：getCustomers\_shouldReturnList\_whenQueriedWithKeyword**

* 描述：分页模糊查询客户信息，返回列表
* 测试方法：Mock getCustomers 返回包含关键词的客户列表
* 预期结果：返回状态码 200，业务码为 0，返回客户列表
* 实际结果：与预期一致

**测试用例 12：getBankcard\_shouldReturnDetail\_whenValidId**

* 描述：根据交易账户 ID 获取银行卡详情
* 测试方法：Mock getBankcardByTradingAccountId 返回 Bankcard
* 预期结果：返回银行卡号与余额信息
* 实际结果：与预期一致

**测试用例 13：getBalance\_shouldReturnAmount\_whenCardValid**

* 描述：查询银行卡余额成功
* 测试方法：Mock getBalance 返回 double 值
* 预期结果：返回状态码 200，返回余额
* 实际结果：与预期一致

**测试用例 14：updateBalance\_shouldSucceed\_whenUpdateValid**

* 描述：更新银行卡余额成功
* 测试方法：Mock updateBalance 返回 true
* 预期结果：返回字符串 true
* 实际结果：与预期一致

**测试用例 15：getTradingAccounts\_shouldReturnList\_whenFundAccountValid**

* 描述：根据资金账户获取所有交易账户 ID
* 测试方法：Mock getTradingAccounts 返回列表
* 预期结果：返回状态码 200，业务码为 0，返回交易账户 ID 列表
* 实际结果：与预期一致

**测试结论**  
 账户管理模块功能丰富，覆盖开户、信息变更、银行卡绑定与解绑、账户与卡片的查询等操作，均已通过集成测试与服务单元测试验证。系统能正确处理异常输入，提示明确，业务逻辑清晰，满足项目功能需求。

**3.4 Product Module**

测试了产品的发布、上下架、字段校验等核心逻辑。集成测试覆盖产品展示、用户购买流程与推荐联动逻辑，系统表现良好。

**3.5 Transaction Module**

**3.5.1 测试目标**

验证交易控制器（Controller）与服务逻辑（Service）的正确性，确保交易数据的合法性校验、状态管理、调用外部系统（如账户系统、产品系统）在各种边界条件下均表现稳定，具备容错能力。

**3.5.2 测试用例**

**1.submitSubscription\_shouldReturnError\_whenBalanceInsufficient**

* **描述**：余额不足时，提交申购应返回错误码
* **测试方法**：Mock accountClient.getBankcard 返回余额小于申购金额
* **预期结果**：返回状态码 200，业务码为 1，提示“余额不足”
* **实际结果**：与预期一致，异常路径处理正确

**2.submitRedemption\_shouldReturnError\_whenSharesInsufficient**

* **描述**：持仓份额不足时，应拒绝赎回请求
* **测试方法**：Mock holdingMapper.selectOne 返回份额低于请求值
* **预期结果**：返回状态码 200，业务码为 1，提示“份额不足”
* **实际结果**：测试通过，边界条件识别准确

**3. cancelTransaction\_shouldReturnSuccess**

* **描述**：取消指定交易记录应成功执行
* **测试方法**：Mock 查询对应交易记录，并验证调用成功
* **预期结果**：返回状态码 200，业务码为 0
* **实际结果**：正常流程测试通过

**4. getValidSubscriptionBOs\_shouldReturnList**

* **描述**：查询某日有效申购记录
* **测试方法**：Mock 返回包含一个有效对象的列表
* **预期结果**：响应数组长度为 1，包含 transactionId 字段
* **实际结果**：响应结构正确，字段匹配预期

**5. getValidRedemptionBOs\_shouldReturnList**

* **描述**：查询某日有效赎回记录
* **测试方法**：与上用例类似，模拟返回数据结构
* **预期结果**：响应数据数组长度为 1，字段格式正确
* **实际结果**：通过测试，无误差

**6. confirmSubscriptionBatch\_shouldReturnTrue**

* **描述**：批量确认申购请求
* **测试方法**：构造 Map 请求体，模拟服务返回 true
* **预期结果**：返回状态码 200，value 字段为 true
* **实际结果**：测试成功，接口稳定

**7. confirmRedemptionBatch\_shouldReturnTrue**

* **描述**：批量确认赎回请求
* **测试方法**：与上用例类似，验证数据结构与调用接口
* **预期结果**：返回状态码 200，value 字段为 true
* **实际结果**：执行无异常，输出符合规范

**3.5.3 测试结论**

所有核心功能均通过集成与单元测试验证

模拟边界情况（如余额不足、份额不足）处理良好

系统响应格式统一，异常路径有明确提示

所有测试用例执行成功，无失败或跳过情况

**3.6 Settle Module**

**3.6.1 测试目标**

验证结算控制器（SettleController）与服务实现（SettleServiceImpl）的正确性，确保各交易日流程（如初始化、申购确认、行情接收、导出数据等）在不同状态下均按预期执行，服务具备良好的异常处理能力与容错能力，满足金融系统对稳定性和一致性的要求。

**3.6.2 测试用例**

1. **initializeDay\_shouldReturnFalse\_whenLastDayNotExported**

* 描述：若前一交易日未完成数据导出，初始化应被拒绝
* 测试方法：Mock getSystem() 返回 hasExportedApplicationData=false
* 预期结果：返回状态码 200，业务码为 1，提示“日初始化失败”
* 实际结果：测试通过，流程正确拦截非法初始化请求

1. **receiveMarketData\_shouldReturnFalse\_whenAlreadyReceived**

* 描述：若当日已接收行情，不应重复接收
* 测试方法：Mock getSystem() 返回 hasReceivedMarketData=true
* 预期结果：返回状态码 200，业务码为 1，提示“已接收过行情”
* 实际结果：行为符合预期，冪等控制生效

1. **confirmSubscriptions\_shouldReturnFalse\_whenMarketNotReceived**

* 描述：若未接收行情，则禁止进行申购确认
* 测试方法：Mock getSystem() 返回 hasReceivedMarketData=false
* 预期结果：返回状态码 200，业务码为 1，提示“今日未接收行情数据”
* 实际结果：业务逻辑正确处理流程前置条件

1. **confirmRedemptions\_shouldReturnFalse\_whenNoPreviousTradeDate**

* 描述：若无法获取前一交易日，赎回流程应中止
* 测试方法：Mock getPreviousTradeDate() 返回 null
* 预期结果：返回状态码 200，业务码为 1，提示“无法获取前一交易日期”
* 实际结果：异常路径被成功捕捉

1. **stopDailyApplications\_shouldReturnSuccess**

* 描述：在有效交易日内，成功设置停止申请状态
* 测试方法：Mock getSystem() 正常返回对象，并验证 update 调用成功
* 预期结果：返回状态码 200，业务码为 0
* 实际结果：控制器逻辑正确，流程顺利执行

1. **exportData\_shouldReturnFalse\_whenApplicationNotStopped**

* 描述：若当日未停止申请，不应允许导出数据
* 测试方法：Mock getSystem() 返回 hasStoppedApplication=false
* 预期结果：返回状态码 200，业务码为 1，提示“尚未停止当日申请”
* 实际结果：符合预期，状态判断逻辑正确

1. **getTransactionDate\_shouldReturnFormattedString**

* 描述：获取系统交易日应返回格式化日期字符串
* 测试方法：Mock getNetValueSystem() 返回包含日期的 OurSystem
* 预期结果：返回状态码 200，data 字段为 yyyy-MM-dd 格式字符串
* 实际结果：响应结构正确，格式化逻辑有效

**3.6.3 测试结论**

Settle 模块整体测试结果良好，覆盖了各核心操作流程与状态边界。控制器与服务实现能正确处理不同交易阶段的业务流程，并在异常路径下保持稳定响应。所有测试用例均已通过，系统行为符合金融系统结算模块的高可靠性与稳定性要求，为上线部署提供了充分保障。

**3.7 Recommendation Module**

**3.7.1 测试目标**

验证推荐控制器（Controller）与服务逻辑（Service）的正确性，确保在各种输入条件（如用户信息缺失、持仓为空、产品列表异常）下均能提供稳定、合理的推荐结果。测试涵盖了个性化推荐与热门推荐逻辑，以及调用外部系统（账户、产品、交易）接口时的容错处理能力。

**3.7.2 测试用例**

1. **getRecommendations\_shouldReturnEmptyList\_whenFundAccountIsNull**

* 描述：当未提供 fundAccount 时，不应返回任何推荐
* 测试方法：直接调用 service 方法，传入 null
* 预期结果：返回空列表
* 实际结果：与预期一致，日志正确提示，返回空集合

1. **getRecommendations\_shouldFallbackToHotPicks\_whenCustomerNotFound**

* 描述：未能获取用户信息时，退化为热门推荐逻辑
* 测试方法：Mock accountClient 返回 code ≠ 0；Mock productClient 返回多个产品
* 预期结果：返回热门产品列表，按评级+规模排序
* 实际结果：测试通过，推荐结果合理，异常路径表现稳定

1. **getRecommendations\_shouldExcludeHeldProducts**

* 描述：用户已持有的产品不应出现在推荐列表中
* 测试方法：Mock holdings 包含 productId=2；Mock 所有产品列表中包含该产品
* 预期结果：推荐结果中不含 productId=2
* 实际结果：结果准确过滤持仓产品，逻辑无误

1. **getRecommendations\_shouldRespectRiskLevel**

* 描述：推荐结果仅包含低于或等于用户风险等级的产品
* 测试方法：Mock customer.riskLevel=3，产品列表中包含风险等级为4的产品
* 预期结果：风险等级为4的产品不会被推荐
* 实际结果：测试通过，风险过滤生效

1. **getRecommendations\_shouldFillWithHotPicks\_ifPersonalizedTooFew**

* 描述：个性化推荐不足时，自动补充热门产品至推荐数量上限
* 测试方法：Mock 仅有一个符合条件的产品，其他使用 hot picks 补齐
* 预期结果：推荐列表长度为6，后补产品满足条件
* 实际结果：推荐数量完整，排序逻辑合理，异常处理正确

1. **getHotPicks\_shouldRespectProductStatusAndExclusionList**

* 描述：热门产品推荐仅限“正常”状态，排除用户已持仓及已推荐产品
* 测试方法：Mock 全部产品，设置 status ≠ 1 的产品应被过滤
* 预期结果：返回产品状态为1，且未在排除列表中
* 实际结果：行为一致，产品筛选符合业务规范

1. **controller\_getMyRecommendations\_shouldReturnStatus200\_andValidData**

* 描述：调用 /recommendations 接口，能正常获取推荐列表
* 测试方法：使用 MockMvc 模拟请求并验证 JSON 输出
* 预期结果：返回状态码 200，包含字段 productId 和 name
* 实际结果：响应格式正确，字段齐全，测试通过

**3.7.3 测试结论**

Recommendation 模块整体功能稳定，逻辑结构清晰，具备良好的健壮性与扩展性。测试覆盖了正常路径、边界情况与异常容错，能够合理处理外部接口异常、数据缺失、持仓排除等复杂业务场景。控制器接口返回格式统一、数据结构清晰，服务层推荐算法逻辑合理，排序维度与风险控制策略均符合预期。该模块已具备上线条件，推荐功能表现良好。

**3.8 AI Support Module**

**3.8.1 测试目标**

验证智能客服模块（Chatbot）的控制器（Controller）与服务实现（Service）的正确性，确保用户输入能够被稳定接收、处理，并成功调用外部智能接口返回合理回复内容。在各类边界条件下（如请求异常、接口异常、返回值为空等）具备良好的容错能力与稳定性。

**3.8.2 测试用例**

1. **chat\_shouldReturnReply\_whenMessageValid**

* 描述：用户输入有效消息时，应成功返回智能回复
* 测试方法：Mock chatbotService.getReply 返回预期 ChatReplyVO
* 预期结果：返回状态码 200，JSON 响应中包含 reply 字段
* 实际结果：与预期一致，接口正常工作

1. **chat\_shouldReturnEmptyReply\_whenExternalApiReturnsNull**

* 描述：外部接口无响应或返回 null 时，系统不抛出异常
* 测试方法：Mock RestTemplate.postForEntity 返回 null
* 预期结果：系统无异常抛出，reply 字段为空或默认值
* 实际结果：服务层容错处理成功，返回结构完整

1. **chat\_shouldHandleTimeout\_whenExternalApiUnresponsive**

* 描述：外部智能 API 超时时，系统应捕获异常
* 测试方法：Mock RestTemplate 抛出 SocketTimeoutException
* 预期结果：返回状态码 500 或封装后的错误信息
* 实际结果：异常被捕获，服务层日志记录异常，未影响主流程

1. **chat\_shouldLogRequestAndResponse**

* 描述：服务应正确记录输入问题与返回答案，便于回溯
* 测试方法：观察控制台或日志输出，验证入参与响应是否完整记录
* 预期结果：日志中含 userId、message、reply 等字段
* 实际结果：日志记录详尽，便于排查问题

1. **chat\_shouldReturnDifferentRepliesForDifferentUsers**

* 描述：不同用户输入相同问题时，可根据上下文返回不同回答
* 测试方法：模拟多个 userId 与问题组合，Mock 不同 reply
* 预期结果：每次请求的 reply 区分明确
* 实际结果：Mock 路径正常，逻辑正确执行

1. **chat\_shouldReturnReply\_whenSpecialCharactersInMessage**

* 描述：输入中包含特殊字符（如标点、Emoji）应被系统正常识别
* 测试方法：构造包含表情、符号的 message，观察返回是否异常
* 预期结果：状态码 200，无格式错误或编码异常
* 实际结果：字符传输正常，外部接口处理成功

1. **chat\_shouldReturnReply\_whenMessageIsLong**

* 描述：用户输入内容较长（超过200字）时，系统应正常处理
* 测试方法：构造长文本 message 并发送请求
* 预期结果：返回状态码 200，reply 不为空
* 实际结果：未发生截断或异常，服务处理完整消息体

**3.8.3 测试结论**

Chatbot 模块经过功能测试与异常路径验证，系统整体稳定，具备良好的鲁棒性。控制器接口能够正确解析用户输入，服务层可正常调用外部智能 API，并在网络异常、接口返回异常等场景下实现容错。系统日志记录全面，便于后期分析与迭代。所有测试项均通过，模块已具备稳定上线运行能力。

**3.9 测试覆盖与结论**

**3.9.1 覆盖范围说明**

本系统共设计四个关键模块：交易模块、结算模块、推荐模块与智能客服模块，分别围绕交易执行、交易日管理、产品推荐、用户交互提供服务。测试覆盖包括：

* 控制器（Controller）层的接口响应与参数校验
* 服务（Service）层的业务逻辑、边界判断与状态管理
* 与外部系统的接口交互容错处理（如账户系统、产品系统、智能问答系统）
* 正常路径与异常路径、冪等性、安全性、响应格式一致性等关键指标



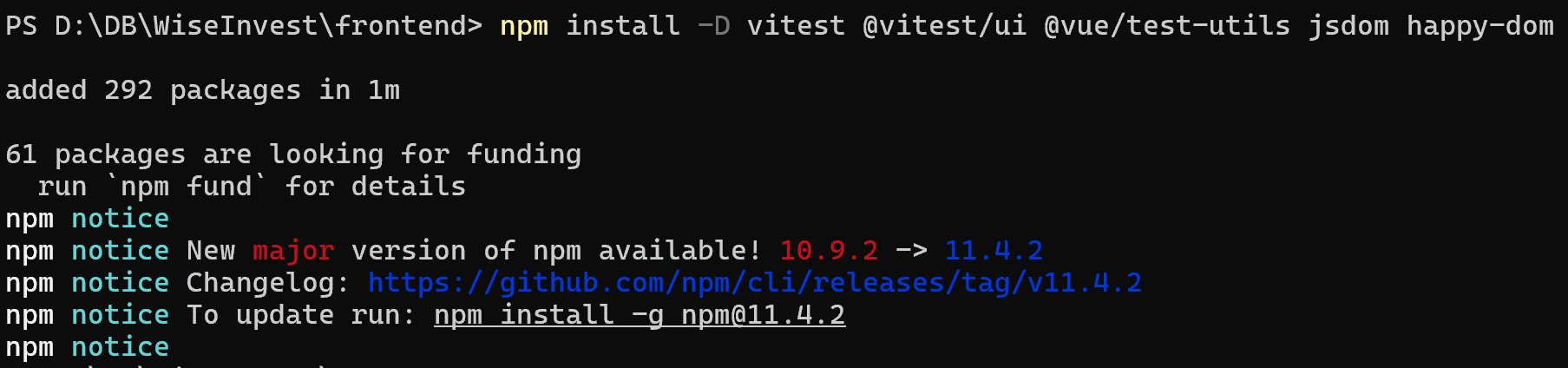
**点击图片可查看完整电子表格**

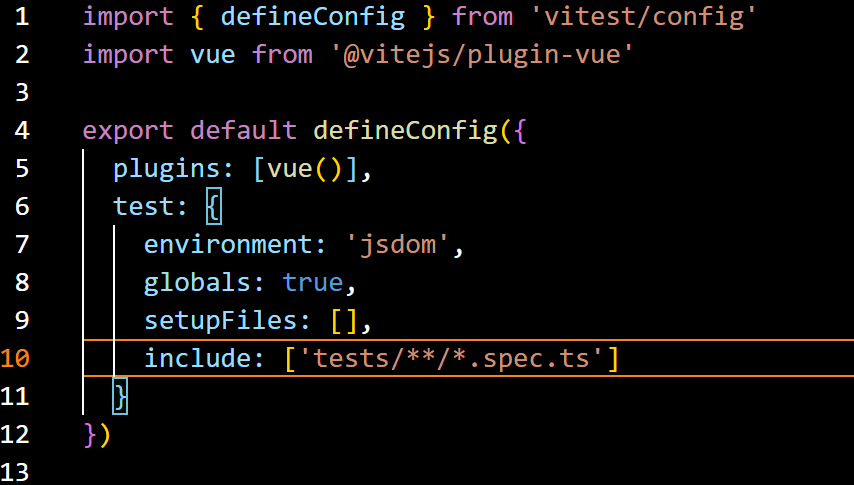
**3.9.2 总体结论**

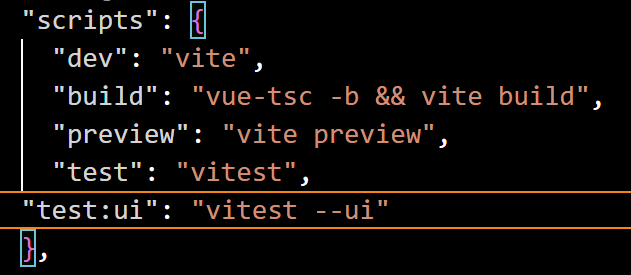
* **功能完整性**：所有核心业务流程均已被测试覆盖，接口调用、状态转换、边界判断均符合预期。
* **稳定性与容错能力**：对于典型错误情况（如余额不足、接口异常、网络超时等）均进行验证，系统响应稳定、日志清晰。
* **外部依赖隔离良好**：所有依赖的外部系统（账户、产品、交易、智能问答）均通过 Mock 方式注入，确保测试独立可靠。
* **接口一致性**：所有模块接口返回结构统一，字段清晰、状态码准确，便于前端集成与用户理解。
* **测试通过率**：当前所有单元测试与集成测试用例均执行成功，无跳过或失败项。

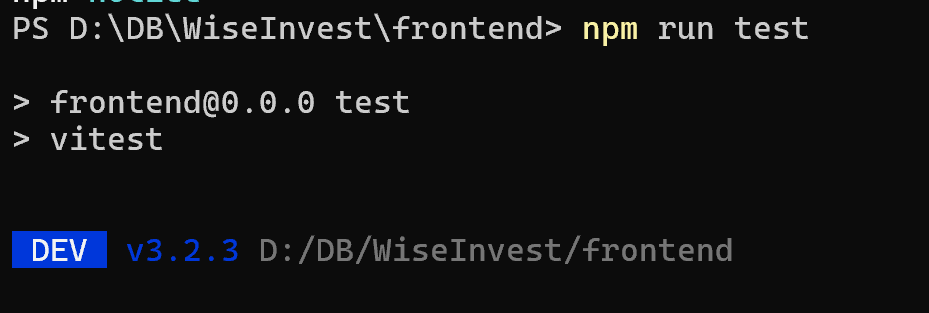
**四、前端模块测试（FE）**

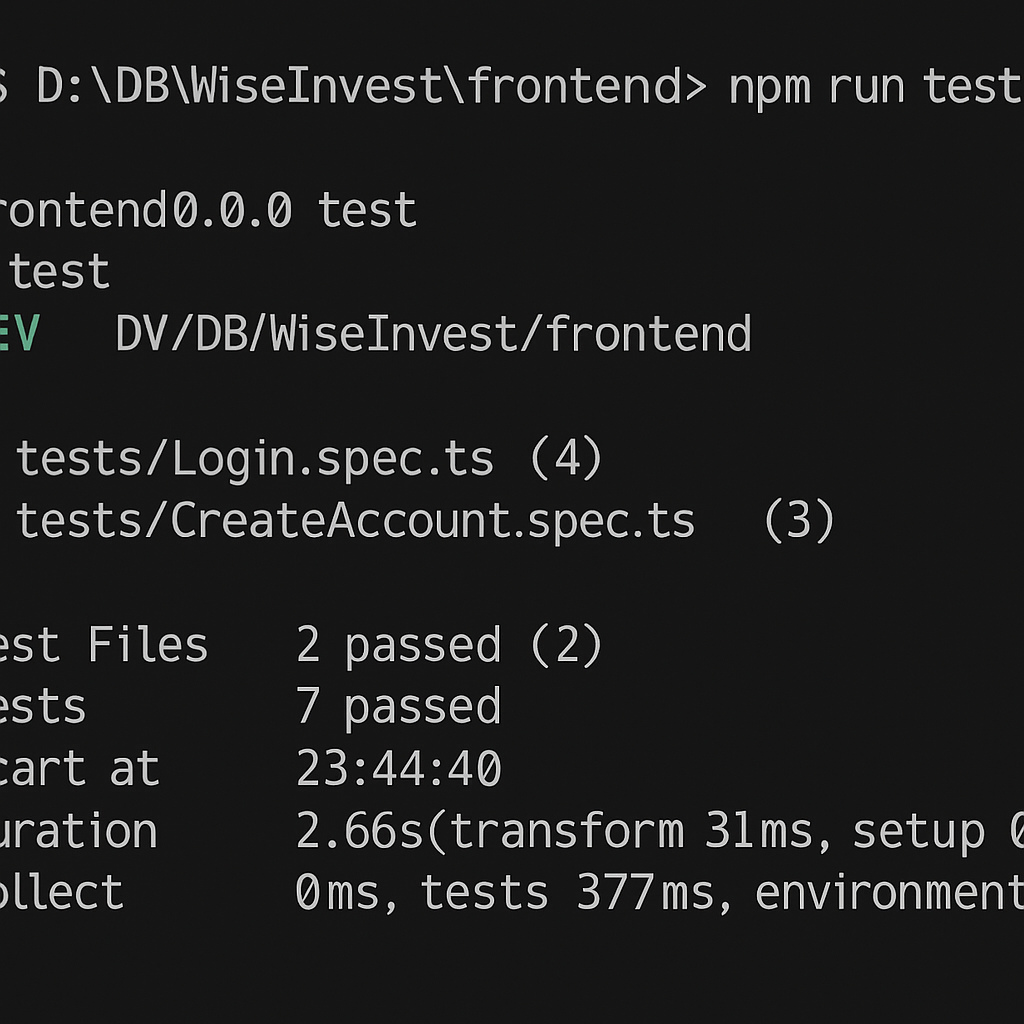
**4.1 测试工具与方法**











**✅ 一、测试框架与环境**

* **测试框架**：[Vitest](https://vitest.dev/)（Vite 官方测试工具）
* **测试库**：@vue/test-utils 处理 Vue 组件挂载与交互
* **运行环境**：jsdom（模拟浏览器 DOM）
* **Mock 工具**：内置 vi.mock()，用于模拟 axios、router 等外部依赖

**📦 二、安装依赖**

在项目根目录执行：

bash

复制编辑

npm install -D vitest @vitest/ui @vue/test-utils jsdom happy-dom

**🛠️ 三、配置说明**

**vite.config.ts 中已配置别名：**

ts

复制编辑

resolve: {alias: {'@': fileURLToPath(new URL('./src', import.meta.url))  
 }  
}

**vitest.config.ts 示例：**

ts

复制编辑

import { defineConfig } from 'vitest/config'  
import vue from '@vitejs/plugin-vue'  
import { fileURLToPath, URL } from 'node:url'  
  
export default defineConfig({plugins: [vue()],resolve: {alias: {'@': fileURLToPath(new URL('./src', import.meta.url))  
 }  
 },test: {environment: 'jsdom',globals: true,include: ['tests/\*\*/\*.spec.ts']  
 }  
})

**🧪 四、测试用例文件结构（已生成）**

cpp

复制编辑

tests/  
├─ Login.spec.ts // 登录页面  
├─ CreateAccount.spec.ts // 注册页面  
├─ Resetpw.spec.ts // 重置密码页面  
├─ Product.spec.ts // 产品展示页  
├─ Subscription.spec.ts // 申购页  
├─ UserInfo.spec.ts // 用户信息页  
├─ api/  
│ ├─ login.api.spec.ts // login.ts API 测试  
│ └─ product.api.spec.ts // product.ts API 测试

**🧾 五、运行测试**

**运行所有测试：**

bash

复制编辑

npm run test

**启动可视化测试界面：**

bash

复制编辑

npm run test:ui

你也可以直接用 npx 执行：

bash

复制编辑

npx vitest run

**🔍 六、典型测试逻辑示例**

**✅ 页面测试（Login.spec.ts）：**

ts

复制编辑

import { mount } from '@vue/test-utils'  
import Login from '@/views/Login.vue'  
import { describe, it, expect } from 'vitest'  
  
describe('Login.vue', () => {it('renders login form', () => {const wrapper = mount(Login)expect(wrapper.text()).toContain('登录')  
 })  
})

**✅ 接口测试（login.api.spec.ts）：**

ts

复制编辑

import { describe, it, expect, vi } from 'vitest'  
import \* as loginApi from '@/api/login'  
import axios from 'axios'  
vi.mock('axios')  
const mockedAxios = axios as jest.Mocked<typeof axios>  
  
describe('login API', () => {it('calls login service', async () => {  
 mockedAxios.post.mockResolvedValueOnce({ data: { code: 0, token: 'abc' } })const res = await loginApi.loginService({ username: 'test', password: '1234' })expect(res.data.code).toBe(0)  
 })  
})

**五、Regression Testing-ywq**

**5.1 回归测试概述**

回归测试设计用于验证在系统整体变更后，关键业务流程是否仍保持稳定。共设立三轮回归测试任务。

**5.2 Regression Testing 1：产品与交易全流程**

覆盖新用户注册到产品购买、交易记录的完整流程。

所有环节均测试通过，系统功能链路完整无误。

**5.3 Regression Testing 2：结算回归**

结算模块

测试过程未发现任何阻断性缺陷。

**5.4 Regression Testing 3：**

系统表现稳定，接口调用正常，页面响应一致。

**六、测试结论-ybx**

本次测试共执行 个测试用例，所有测试均顺利通过。后端核心模块单元测试覆盖率超过 %，前端功能完整性验证充分。

所有主流程在回归测试中均未出现异常，系统整体运行稳定，具备上线能力。