# 课设设计源代码合集

## 基础1.cpp

|  |
| --- |
| #include <iostream>  class matrix  {  private:      int \*\*matri;      int row;      int col;  public:      matrix(int row, int col) // 创建一个row行col列的矩阵      {          this->row = row;          this->col = col;          matri = new int \*[row];          for (int i = 0; i < row; i++)          {              matri[i] = new int[col];          }      }      ~matrix()      {          if (matri != nullptr)          {              for (int i = 0; i < row; i++)              {                  if (matri[i] != nullptr)                  {                      delete[] matri[i];                      matri[i] = nullptr;                  }              }              delete[] matri;              matri = nullptr;          }      }      void matrixinput()      {          for (int i = 0; i < this->row; i++)          {              for (int j = 0; j < this->col; j++)              {                  std::cin >> matri[i][j];              }          }      }      void matrixoutput()      {          for (int i = 0; i < this->row; i++)          {              for (int j = 0; j < this->col; j++)              {                  std::cout << matri[i][j] << " ";              }              std::cout << std::endl;          }      }      // 使用常量引用而杜绝拷贝构造函数的调用      void matrixplus(const matrix &m1, const matrix &m2)      {          if (m1.row != m2.row || m1.col != m2.col)          {              std::cout << "矩阵大小不同，无法相加" << std::endl;              return;          }          for (int i = 0; i < m1.row; i++)          {              for (int j = 0; j < m1.col; j++)              {                  matri[i][j] = m1.matri[i][j] + m2.matri[i][j];              }          }      }      void matrixminus(const matrix &m1, const matrix &m2)      {          if (m1.row != m2.row || m1.col != m2.col)          {              std::cout << "矩阵大小不同，无法相减" << std::endl;              return;          }          for (int i = 0; i < m1.row; i++)          {              for (int j = 0; j < m1.col; j++)              {                  matri[i][j] = m1.matri[i][j] - m2.matri[i][j];              }          }      }  };  int main()  {      matrix m(4, 5);      std::cout << "矩阵创建成功，请输入4\*5矩阵" << std::endl;      m.matrixinput();      std::cout << "矩阵为：" << std::endl;      m.matrixoutput();      std::cout << "请输入两个矩阵，用于相加和相减" << std::endl;      matrix m1(4, 5);      m1.matrixinput();      matrix m2(4, 5);      m2.matrixinput();      m2.matrixoutput();      matrix m3(4, 5);      m3.matrixminus(m, m1);      m3.matrixoutput();      m3.matrixplus(m, m1);      m3.matrixoutput();      // m1.~matrix();      // m2.~matrix();      // m3.~matrix();      int row, col;      std::cout << "请输入行数和列数：" << std::endl;      std::cin >> row >> col;      matrix \*A1 = new matrix(row, col);      matrix \*A2 = new matrix(row, col);      matrix \*A3 = new matrix(row, col);      std::cout << "请输入两个矩阵，用于相加和相减" << std::endl;      A1->matrixinput();      A2->matrixinput();      A3->matrixminus(\*A1, \*A2);      A3->matrixoutput();      A3->matrixplus(\*A1, \*A2);      A3->matrixoutput();      delete A1;      delete A2;      delete A3;      // A1.~matrix();      // A2.~matrix();      // A3.~matrix();      return 0;  } |

## 基础2.1.cpp

|  |
| --- |
| #include <iostream>  #include <cmath>  #include <iomanip>  using namespace std;  class Point  {  private:      int x;      int y;  public:      Point(int x = 0, int y = 0)      {          this->x = x;          this->y = y;          cout << "构造点函数" << endl;      }      ~Point()      {          cout << "析构点函数，位置：（" << x << "，" << y << "）" << endl;      }      // 输入方法      void input()      {          std::cout << "请输入x坐标: ";          std::cin >> x;          std::cout << "请输入y坐标: ";          std::cin >> y;      }      // 静态工厂方法      static Point createFromInput()      {          Point p;          p.input();          return p;      }      // 获取坐标      int getX() const      {          return x;      }      int getY() const      {          return y;      }      double distance(const Point &p)      {          return sqrt((x - p.x) \* (x - p.x) + (y - p.y) \* (y - p.y));      }  };  class Circle  {  private:      Point center;      int radius;  public:      Circle(const Point &center)      {          this->center = center;          std::cout << "请输入半径：";          std::cin >> radius;      }      // 默认构造，通过输入创建      Circle()      {          std::cout << "是否要输入圆心坐标？(y/n): ";          char choice;          std::cin >> choice;          if (choice == 'y' || choice == 'Y')          {              int x, y;              std::cout << "请输入x坐标: ";              std::cin >> x;              std::cout << "请输入y坐标: ";              std::cin >> y;              center = Point(x, y);          }          else          {              center = Point();              std::cout << "使用默认圆心坐标(0,0)" << std::endl;          }          std::cout << "请输入半径：";          std::cin >> radius;      }      ~Circle()      {          std::cout << "析构圆函数，圆心：（" << center.getX() << "，" << center.getY()                    << "），半径：" << radius << std::endl;      }      bool isIntersect(const Circle &c)      {          return center.distance(c.center) < radius + c.radius; // 不包括相切      }  };  int main()  {      std::cout << "创建第一个点：" << std::endl;      // 使用静态工厂方法创建点      Point p1 = Point::createFromInput();      std::cout << "创建的点坐标为: (" << p1.getX() << ", " << p1.getY() << ")" << std::endl;      std::cout << "创建第二个点：" << std::endl;      Point p2;      p2.input();      std::cout << "创建的点坐标为: (" << p2.getX() << ", " << p2.getY() << ")" << std::endl;      std::cout << "使用第一个点创建圆：" << std::endl;      Circle c1(p1);      std::cout << "直接创建圆：" << std::endl;      Circle c2;      if (c1.isIntersect(c2))      {          std::cout << "两个圆相交" << std::endl;      }      else      {          std::cout << "两个圆不相交" << std::endl;      }      return 0;  } |

## 基础2.2.cpp

|  |
| --- |
| #include <iostream>  class matrix  {  private:      int \*\*matri;      int rows;      int cols;  public:      matrix(int rows, int cols) // 创建一个rows行lines列的矩阵      {          this->rows = rows;          this->cols = cols;          matri = new int \*[rows];          for (int i = 0; i < rows; i++)          {              matri[i] = new int[cols];          }          for (int i = 0; i < rows; i++) // 默认初始化          {              for (int j = 0; j < cols; j++)              {                  matri[i][j] = 0;              }          }      }      matrix()      {          std::cout << "请输入行数和列数：" << std::endl;          std::cin >> rows >> cols;          matri = new int \*[rows];          for (int i = 0; i < rows; i++)          {              matri[i] = new int[cols];          }          std::cout << "请输入矩阵元素：" << std::endl;          matrixinput();      }      matrix(const matrix &m) // 拷贝构造      {          this->rows = m.rows;          this->cols = m.cols;          matri = new int \*[rows];          for (int i = 0; i < rows; i++)          {              matri[i] = new int[cols];          }          for (int i = 0; i < rows; i++)          {              for (int j = 0; j < cols; j++)              {                  matri[i][j] = m.matri[i][j];              }          }      }      ~matrix() // 析构函数，释放内存      {          if (matri != nullptr)          {              for (int i = 0; i < rows; i++)              {                  if (matri[i] != nullptr)                  {                      delete[] matri[i];                      matri[i] = nullptr;                  }              }              delete[] matri;              matri = nullptr;          }      }      void matrixinput()      {          for (int i = 0; i < this->rows; i++)          {              for (int j = 0; j < this->cols; j++)              {                  std::cin >> matri[i][j];              }          }      }      void matrixoutput()      {          for (int i = 0; i < this->rows; i++)          {              for (int j = 0; j < this->cols; j++)              {                  std::cout << matri[i][j] << " ";              }              std::cout << std::endl;          }      }      // 友元函数方法可以需要两个参数（左右操作数）      // 也可以放置在类外部，作为全局函数      friend matrix operator+(const matrix &m1, const matrix &m2)      {          if (m1.rows != m2.rows || m1.cols != m2.cols)          {              std::cout << "矩阵大小不同，无法相加" << std::endl;              return matrix(m1);          }          matrix m3(m1.rows, m1.cols);          for (int i = 0; i < m1.rows; i++)          {              for (int j = 0; j < m1.cols; j++)              {                  m3.matri[i][j] = m1.matri[i][j] + m2.matri[i][j];              }          }          return m3;      }      matrix operator-(const matrix &m1)      {          if (rows != m1.rows || cols != m1.cols)          {              std::cout << "矩阵大小不同，无法相减" << std::endl;              return matrix(\*this);          }          matrix m3(rows, cols);          for (int i = 0; i < rows; i++)          {              for (int j = 0; j < cols; j++)              {                  m3.matri[i][j] = matri[i][j] - m1.matri[i][j];              }          }          return m3;      }      void matrixplus(const matrix &m1, const matrix &m2)      {          if (m1.rows != m2.rows || m1.cols != m2.cols)          {              std::cout << "矩阵大小不同，无法相加" << std::endl;              return;          }          for (int i = 0; i < m1.rows; i++)          {              for (int j = 0; j < m1.cols; j++)              {                  matri[i][j] = m1.matri[i][j] + m2.matri[i][j];              }          }      }        void matrixminus(const matrix &m1, const matrix &m2)      {          if (m1.rows != m2.rows || m1.cols != m2.cols)          {              std::cout << "矩阵大小不同，无法相减" << std::endl;              return;          }          for (int i = 0; i < m1.rows; i++)          {              for (int j = 0; j < m1.cols; j++)              {                  matri[i][j] = m1.matri[i][j] - m2.matri[i][j];              }          }      }      matrix &operator=(const matrix &m) // 复制并交换？      {          if (this == &m) // 自赋值检查          {              return \*this;          }          if (matri != nullptr) // 旧内存释放          {              for (int i = 0; i < rows; i++)              {                  if (matri[i] != nullptr)                  {                      delete[] matri[i];                      matri[i] = nullptr;                  }              }              delete[] matri;              matri = nullptr;          }          rows = m.rows;          cols = m.cols;          matri = new int \*[rows];          for (int i = 0; i < rows; i++)          {              matri[i] = new int[cols];              for (int j = 0; j < cols; j++)              {                  matri[i][j] = m.matri[i][j];              }          }          return \*this;      }  };  int main()  {      matrix A1 = matrix();      matrix A2 = matrix();      matrix A3 = A1 + A2;      std::cout << "相加后的矩阵：" << std::endl;      A3.matrixoutput();      matrix A4 = matrix(A3);      std::cout << "拷贝构造相加的矩阵：" << std::endl;      A4.matrixoutput();      int rows, cols;      std::cout << "请输入行数和列数：" << std::endl;      std::cin >> rows >> cols;      matrix B1(rows, cols);      matrix B2(rows, cols);      std::cout << "请输入第一个矩阵：" << std::endl;      B1.matrixinput();      std::cout << "请输入第二个矩阵：" << std::endl;      B2.matrixinput();      matrix B3 = B1 + B2;      std::cout << "相加后的矩阵：" << std::endl;      B3.matrixoutput();      matrix B4 = B1 - B2;      std::cout << "相减后的矩阵：" << std::endl;      B4.matrixoutput();      matrix \*C1 = new matrix(2, 2);      matrix \*C2 = new matrix(2, 2);      std::cout << "请输入第一个矩阵：" << std::endl;      C1->matrixinput();      std::cout << "请输入第二个矩阵：" << std::endl;      C2->matrixinput();      matrix C3 = \*C1 + \*C2;      std::cout << "相加后的矩阵：" << std::endl;      C3.matrixoutput();      matrix C4 = \*C1 - \*C2;      std::cout << "相减后的矩阵：" << std::endl;      C4.matrixoutput();      delete C1;      delete C2;      return 0;  } |

## 基础3.cpp

|  |
| --- |
| #include <iostream>  #include <cmath>  class shape  {  private:  public:      shape()      {          std::cout << "shape 构造" << std::endl;      }      virtual ~shape()      {          std::cout << "shape 析构" << std::endl;      }      virtual double area() = 0;      // 重载1：打印信息      void print()      {          std::cout << "这是一个形状" << std::endl;      }      // 重载2：打印信息(带信息)      void print(const std::string &message)      {          std::cout << "形状信息: " << message << std::endl;      }  };  class circle : public shape  {  private:      double radius;  public:      circle(double r) : radius(r)      {          std::cout << "circle 构造函数被调用，半径=" << radius << std::endl;      }      // 析构函数      ~circle() override      {          std::cout << "circle 析构函数被调用，半径=" << radius << std::endl;      }      double area() override      {          return M\_PI \* radius \* radius;      }      double area(int r) const      {          return M\_PI \* r \* r;      }      // 重写：基类的print方法      void print()      {          std::cout << "这是一个圆形，半径=" << radius << std::endl;      }  };  class rectangle : public shape  {  private:      double a;      double b;  public:      // 构造函数      rectangle(double w, double h) : a(w), b(h)      {          std::cout << "rectangle 构造函数被调用，宽=" << w << "，高=" << h << std::endl;      }      // 析构函数      virtual ~rectangle() override      {          std::cout << "rectangle 析构函数被调用" << std::endl;      }      virtual double area() override      {          return a \* b;      }      double area(int x, int y)      {          return x \* y;      }  };  class square : public rectangle  {  private:      double a;  public:      square(double a) : rectangle(a, a), a(a) // 首先初始化基类，然后初始化自己的成员      {          std::cout << "square 构造函数被调用，边长=" << a << std::endl;      }      ~square() override      {          std::cout << "square 析构函数被调用" << std::endl;      }      double area() override      {          return a \* a;      }      double area(int r)      {          return r \* r;      }  };  int main()  {      std::cout << "\n===== 测试1: 单独创建各种形状对象 =====\n"                << std::endl;      // 创建圆形对象      std::cout << "创建圆形对象:" << std::endl;      circle c(5.0);      std::cout << "圆形面积: " << c.area() << std::endl;      c.print();      std::cout << std::endl;      // 创建矩形对象      std::cout << "创建矩形对象:" << std::endl;      rectangle r(4.0, 6.0);      std::cout << "矩形面积: " << r.area() << std::endl;      r.print();      std::cout << std::endl;      // 创建正方形对象      std::cout << "创建正方形对象:" << std::endl;      square s(3.0);      std::cout << "正方形面积: " << s.area() << std::endl;      s.print();      std::cout << std::endl;      std::cout << "\n===== 测试2: 观察对象生命周期 =====\n"                << std::endl;      std::cout << "创建局部作用域..." << std::endl;      {          std::cout << "进入局部作用域" << std::endl;          circle localCircle(2.0);          rectangle localRect(2.0, 3.0);          square localSquare(4.0);          std::cout << "即将离开局部作用域" << std::endl;      } // 局部对象在这里销毁      std::cout << "已离开局部作用域，局部对象已被销毁" << std::endl;      std::cout << "\n===== 测试3: 使用基类指针实现多态 =====\n"                << std::endl;      // 使用基类指针数组存储不同类型的对象      shape \*shapes[3];      std::cout << "创建形状数组:" << std::endl;      shapes[0] = new circle(3.0);      shapes[1] = new rectangle(5.0, 4.0);      shapes[2] = new square(2.0);      // 通过基类指针调用虚函数      std::cout << "\n计算不同形状的面积:" << std::endl;      for (int i = 0; i < 3; i++)      {          std::cout << "形状 " << (i + 1) << " 的面积: " << shapes[i]->area() << std::endl;          shapes[i]->print(); // 多态调用      }      // 释放动态分配的对象      std::cout << "\n释放动态创建的对象:" << std::endl;      for (int i = 0; i < 3; i++)      {          delete shapes[i];      }      std::cout << "\n===== 测试4: 重载方法测试 =====\n"                << std::endl;      // 测试重载的area方法      std::cout << "调用重载的area方法:" << std::endl;      std::cout << "圆形: 指定半径为4的圆面积 = " << c.area(4) << std::endl;      std::cout << "矩形: 指定宽3高7的矩形面积 = " << r.area(3, 7) << std::endl;      std::cout << "正方形: 指定边长5的正方形面积 = " << s.area(5) << std::endl;      // 测试重载的print方法      std::cout << "\n调用重载的print方法:" << std::endl;      c.print();      r.print("这是一个矩形对象");      s.print("这是一个正方形对象");      std::cout << "\n===== 程序结束, 剩余对象将被销毁 =====\n"                << std::endl;      // 使用system("pause")暂停程序(仅适用于Windows)      system("pause");      return 0;  } |

## 基础4.cpp

|  |
| --- |
| #include <iostream>  #include <cstdlib>  #include <ctime>  #include <limits> // 用于 std::numeric\_limits  using namespace std;  int main()  {      int num = rand() % 1000 + 1;      int guess = 0;      while (guess != num)      {          cout << "请输入一个1-1000的整数：" << endl;          if (!(cin >> guess))          {              cout << "输入错误，请输入整数" << endl;              cin.clear();                                         // 清除错误标志              cin.ignore(numeric\_limits<streamsize>::max(), '\n'); // 清除输入缓冲区              continue;                                            // 继续循环，重新输入          }          if (guess < 1 || guess > 1000)          {              cout << "请输入1-1000范围内的整数！" << endl;              continue;          }          if (guess > num)          {              cout << "猜大了" << endl;          }          else if (guess < num)          {              cout << "猜小了" << endl;          }          else          {              cout << "猜对了" << endl;          }      }  } |

## 基础5.cpp

|  |
| --- |
| #include <iostream>  #include <cmath>  class Point  {  private:      int x;      int y;  public:      Point(int x = 0, int y = 0)      {          this->x = x;          this->y = y;          std::cout << "构造点函数" << std::endl;      }      ~Point()      {          std::cout << "析构点函数，位置：（" << x << "，" << y << "）" << std::endl;      }      // 输入方法      void input()      {          std::cout << "请输入x坐标: ";          std::cin >> x;          std::cout << "请输入y坐标: ";          std::cin >> y;      }      // 静态工厂方法      static Point createFromInput()      {          Point p;          p.input();          return p;      }      // 获取坐标      int getX() const      {          return x;      }      int getY() const      {          return y;      }      double distance(const Point &p)      {          return sqrt((x - p.x) \* (x - p.x) + (y - p.y) \* (y - p.y));      }      // 重载运算符,修改x和y      Point operator+(const Point &p) const      {          return Point(x + p.x, y + p.y);      }      Point operator-(const Point &p) const      {          return Point(x - p.x, y - p.y);      }      Point operator++() // 不使用const表示修改自身      {                  // 前置改变自身          ++x;          ++y;          return \*this;      }      Point operator++(int) // 自身修改但返回之前的值      {                     // 后置改变自身          Point temp = \*this;          ++x;          ++y;          return temp;      }      Point operator--()      {          --x;          --y;          return \*this;      }      Point operator--(int)      {          Point temp = \*this;          --x;          --y;          return temp;      }      Point &operator=(const Point &p)      {          if (this != &p) // 防止自赋值          {              x = p.x;              y = p.y;          }          // 无内存分配，故无需判断是否需要释放旧内存          return \*this;      }  };  class shape  {  private:  public:      shape()      {          std::cout << "shape 构造" << std::endl;      }      virtual ~shape()      {          std::cout << "shape 析构" << std::endl;      }      virtual double area() = 0;      // 重载1：打印信息      void print()      {          std::cout << "这是一个形状" << std::endl;      }      // 重载2：打印信息(带信息)      void print(const std::string &message)      {          std::cout << "形状信息: " << message << std::endl;      }  };  class circle : public shape  {  private:      double radius;      Point center; // 圆心坐标  public:      circle(double r, double x = 0, double y = 0) : radius(r), center(x, y)      {          std::cout << "circle 构造函数被调用，半径=" << radius                    << "，圆心=(" << center.getX() << "," << center.getY() << ")" << std::endl;      }      circle()      {          std::cout << "是否要输入圆心坐标？(y/n): ";          char choice;          std::cin >> choice;          std::cin.ignore(std::numeric\_limits<std::streamsize>::max(), '\n');          if (choice == 'y' || choice == 'Y')          {              int x, y;              std::cout << "请输入x坐标: ";              std::cin >> x;              std::cout << "请输入y坐标: ";              std::cin >> y;              center = Point(x, y);          }          else          {              center = Point();              std::cout << "使用默认圆心坐标(0,0)" << std::endl;          }          std::cout << "请输入半径：";          std::cin >> radius;          std::cout << "半径为：" << radius << std::endl;      }      // 析构函数      ~circle() override      {          std::cout << "circle 析构函数被调用，半径=" << radius << std::endl;      }      double area() override      {          return M\_PI \* radius \* radius;      }      double area(int r) const      {          return M\_PI \* r \* r;      }      // 重写：基类的print方法      void print()      {          std::cout << "这是一个圆形，半径=" << radius << std::endl;      }      void printXY()      {          std::cout << "圆心坐标： (" << center.getX() << ", " << center.getY() << ")" << std::endl;      }      void centerpluspre() // 前置++重载      {          center++;      }      void centerpluspost() // 后置++重载      {          center++;      }      void centerminuspre() // 前置--重载      {          center--;      }      void centerminuspost() // 后置--重载      {          center--;      }  };  class rectangle : public shape  {  private:      double a;      double b;  public:      // 构造函数      rectangle(double w, double h) : a(w), b(h)      {          std::cout << "rectangle 构造函数被调用，宽=" << w << "，高=" << h << std::endl;      }      // 析构函数      virtual ~rectangle() override      {          std::cout << "rectangle 析构函数被调用" << std::endl;      }      virtual double area() override      {          return a \* b;      }      double area(int x, int y)      {          return x \* y;      }  };  class square : public rectangle  {  private:      double a;  public:      square(double a) : rectangle(a, a), a(a) // 首先初始化基类，然后初始化自己的成员      {          std::cout << "square 构造函数被调用，边长=" << a << std::endl;      }      ~square() override      {          std::cout << "square 析构函数被调用" << std::endl;      }      double area() override      {          return a \* a;      }      double area(int r)      {          return r \* r;      }  };  int main()  {      std::cout << "\n===== 测试1: 创建各种形状对象并计算面积 =====\n"                << std::endl;      // 创建圆形对象      std::cout << "创建圆形对象:" << std::endl;      circle c1(5.0, 2.0, 3.0); // 半径为5，圆心为(2, 3)      std::cout << "圆形面积: " << c1.area() << std::endl;      c1.print();      std::cout << std::endl;      // 创建矩形对象      std::cout << "创建矩形对象:" << std::endl;      rectangle r1(4.0, 6.0); // 宽为4，高为6      std::cout << "矩形面积: " << r1.area() << std::endl;      r1.print();      std::cout << std::endl;      // 创建正方形对象      std::cout << "创建正方形对象:" << std::endl;      square s1(3.0); // 边长为3      std::cout << "正方形面积: " << s1.area() << std::endl;      s1.print();      std::cout << std::endl;      std::cout << "\n===== 测试2: 观察对象生命周期 =====\n"                << std::endl;      std::cout << "创建局部作用域..." << std::endl;      {          std::cout << "进入局部作用域" << std::endl;          circle localCircle(2.0, 1.0, 1.0); // 半径为2，圆心为(1, 1)          rectangle localRect(2.0, 3.0);     // 宽为2，高为3          square localSquare(4.0);           // 边长为4          std::cout << "即将离开局部作用域" << std::endl;      } // 局部对象在这里销毁      std::cout << "已离开局部作用域，局部对象已被销毁" << std::endl;      std::cout << "\n===== 测试3: 使用基类指针实现多态 =====\n"                << std::endl;      // 使用基类指针数组存储不同类型的对象      shape \*shapes[3];      std::cout << "创建形状数组:" << std::endl;      shapes[0] = new circle(3.0, 0.0, 0.0); // 半径为3，圆心为(0, 0)      shapes[1] = new rectangle(5.0, 4.0);   // 宽为5，高为4      shapes[2] = new square(2.0);           // 边长为2      // 通过基类指针调用虚函数      std::cout << "\n计算不同形状的面积:" << std::endl;      for (int i = 0; i < 3; i++)      {          std::cout << "形状 " << (i + 1) << " 的面积: " << shapes[i]->area() << std::endl;          shapes[i]->print(); // 多态调用      }      // 释放动态分配的对象      std::cout << "\n释放动态创建的对象:" << std::endl;      for (int i = 0; i < 3; i++)      {          delete shapes[i];      }      std::cout << "\n===== 测试4: Point 类的运算符重载 =====\n"                << std::endl;      circle c2; // 默认构造函数      std::cout << "创建圆形对象 c2:" << std::endl;      c2.printXY();         // 打印圆心坐标      c2.centerminuspost(); // 后置--重载      c2.printXY();         // 打印圆心坐标      c2.centerminuspre();  // 前置--重载      c2.printXY();         // 打印圆心坐标      c2.centerpluspost();  // 后置++重载      c2.printXY();         // 打印圆心坐标      c2.centerpluspre();   // 前置++重载      c2.printXY();         // 打印圆心坐标      Point p1(1, 2), p2(3, 4);      std::cout << "初始点 p1: (" << p1.getX() << ", " << p1.getY() << ")" << std::endl;      std::cout << "初始点 p2: (" << p2.getX() << ", " << p2.getY() << ")" << std::endl;      // 测试加法运算符      Point p3 = p1 + p2;      std::cout << "p1 + p2 = (" << p3.getX() << ", " << p3.getY() << ")" << std::endl;      // 测试减法运算符      Point p4 = p1 - p2;      std::cout << "p1 - p2 = (" << p4.getX() << ", " << p4.getY() << ")" << std::endl;      // 测试前置 ++      ++p1;      std::cout << "前置 ++p1: (" << p1.getX() << ", " << p1.getY() << ")" << std::endl;      // 测试后置 ++      p1++;      std::cout << "后置 p1++: (" << p1.getX() << ", " << p1.getY() << ")" << std::endl;      // 测试前置 --      --p2;      std::cout << "前置 --p2: (" << p2.getX() << ", " << p2.getY() << ")" << std::endl;      // 测试后置 --      p2--;      std::cout << "后置 p2--: (" << p2.getX() << ", " << p2.getY() << ")" << std::endl;      std::cout << "\n===== 测试5: 重载方法测试 =====\n"                << std::endl;      // 测试重载的 area 方法      std::cout << "调用重载的 area 方法:" << std::endl;      std::cout << "圆形: 指定半径为4的圆面积 = " << c1.area(4) << std::endl;      std::cout << "矩形: 指定宽3高7的矩形面积 = " << r1.area(3, 7) << std::endl;      std::cout << "正方形: 指定边长5的正方形面积 = " << s1.area(5) << std::endl;      // 测试重载的 print 方法      std::cout << "\n调用重载的 print 方法:" << std::endl;      c1.print();      r1.print("这是一个矩形对象");      s1.print("这是一个正方形对象");      std::cout << "\n===== 程序结束, 剩余对象将被销毁 =====\n"                << std::endl;      return 0;  } |

## 综合1.cpp

### Terminal

#### Main.cpp

|  |
| --- |
| #include <iostream>  #include <vector>  #include <string>  #include <limits>  #include "user/user.h"  #include "page/page.h"   // 包含 Page 基类和派生类  #include "store/store.h" // 包含 Store 类  #include <cmath>  using namespace std;  // 文件路径常量  extern const string USER\_FILE = "./user/users.txt";  extern const string STORE\_FILE = "./store";  int main()  {      vector<User \*> users;      User \*currentUser = nullptr;      // 创建 Store 对象      Store store(STORE\_FILE);      // 创建具体的 Page 派生类对象      Customer\_page customerPage;      Seller\_page sellerPage;      // Page\* currentPageManager = &customerPage; // 如果需要基类指针进行多态调用      // 加载用户数据      users = User::loadUsersFromFile(USER\_FILE);      cout << "已加载 " << users.size() << " 个用户数据。" << endl;      int menuResult = 1; // 1=继续, 0=退出程序, 2=退出登录, 3=进入商城      while (menuResult != 0)      {          // 根据 menuResult 处理状态          if (menuResult == 2)          { // 处理退出登录              currentUser = nullptr;              menuResult = 1;          }          if (menuResult == 3)          { // 处理进入商城              if (currentUser == nullptr || dynamic\_cast<Customer \*>(currentUser))              {                  menuResult = customerPage.pagestore(currentUser, users, store);              }              else if (dynamic\_cast<Seller \*>(currentUser))              {                  menuResult = sellerPage.pagestore(currentUser, users, store);              }              else              {                  cout << "当前用户类型无法进入商城。" << endl;                  menuResult = 1;              }              if (menuResult != 0 && menuResult != 2)                  menuResult = 1; // 确保返回后继续菜单循环              continue;          }          // --- 显示用户账户菜单 ---          if (currentUser == nullptr) // 未登录          {              // 使用 customerPage 对象显示主菜单              menuResult = customerPage.pagemain(users, currentUser);          }          else // 已登录          {              if (dynamic\_cast<Customer \*>(currentUser))              {                  menuResult = customerPage.pagemainCustomer(currentUser, users);              }              else if (dynamic\_cast<Seller \*>(currentUser))              {                  menuResult = sellerPage.pagemainSeller(currentUser, users);              }              // else if (dynamic\_cast<Admin\*>(currentUser)) { ... }              else // 使用字符串匹配作为后备              {                  string type = currentUser->getUserType();                  if (type == "消费者")                  {                      menuResult = customerPage.pagemainCustomer(currentUser, users);                  }                  else if (type == "商家")                  {                      menuResult = sellerPage.pagemainSeller(currentUser, users);                  }                  // else if (type == "管理员") { ... }                  else                  {                      cout << "未知的用户类型 (" << type << ")，将退出登录。" << endl;                      menuResult = 2;                  }              }          }      } // End while loop      // ... (程序结束部分保持不变) ...      User::saveUsersToFile(users, USER\_FILE);      cout << "用户数据已保存。" << endl;      for (auto \*user : users)      {          delete user;      }      users.clear();      return 0;  } |

#### User.h

|  |
| --- |
| #ifndef USER\_H  #define USER\_H  #include <iostream>  #include <string>  #include <vector>  #include <fstream>  #include <limits>  class User  {  protected: // protected成员可被派生类访问      std::string username;      std::string password;      std::string userType; // 用户类型      double balance;      static void clearInputBuffer() // 清空输入缓冲区      {          std::cin.clear();                                                   // 清除错误标志          std::cin.ignore(std::numeric\_limits<std::streamsize>::max(), '\n'); // 忽略剩余输入      }  public:      // 构造函数      User(std::string username, std::string password, double balance = 0.0,           bool customer = false, bool seller = false, bool admin = false);      User();      virtual ~User() = default; // 虚析构函数      // 纯虚函数，使User成为抽象类      virtual std::string getUserType() const = 0;      // 账户操作函数      void displayInfo(int index = -1) const;      bool changePassword(const std::string &oldPassword, const std::string &newPassword);      bool deposit(double amount);      bool withdraw(double amount);      double checkBalance() const;      //      static User \*registerUser(std::vector<User \*> &users);    // 注册新用户      static User \*userLogin(const std::vector<User \*> &users); // 用户登录 (获取输入并验证)      // 文件操作函数      static bool saveUsersToFile(const std::vector<User \*> &users, const std::string &filename);      static std::vector<User \*> loadUsersFromFile(const std::string &filename);      static User \*findUser(const std::vector<User \*> &users, const std::string &username);      static void displayAllUsers(const std::vector<User \*> &users);      static bool isUsernameExists(const std::vector<User \*> &users, const std::string &username);      // Getter 函数      std::string getUsername() const;      std::string getPassword() const;      // Setter 函数      void setUsername(const std::string &newUsername);      void setPassword(const std::string &newPassword);      void setBalance(double newBalance);      // 用户验证      static User \*login(const std::vector<User \*> &users, const std::string &username, const std::string &password);  };  // 消费者类  class Customer : public User  {  public:      Customer(std::string username, std::string password, double balance = 0.0);      Customer();      std::string getUserType() const override { return "customer"; }  };  // 商家类  class Seller : public User  {  public:      Seller(std::string username, std::string password, double balance = 0.0);      Seller();      std::string getUserType() const override { return "seller"; }  };  // 管理员类  class Admin : public User  {  public:      Admin(std::string username, std::string password, double balance = 0.0);      Admin();      std::string getUserType() const override { return "admin"; }  };  #endif // USER\_H |

#### User.cpp

|  |
| --- |
| #include "user.h"  #include <iostream>  #include <limits>  #include <string>  #include <fstream>  #include <sstream>  using namespace std;  // User 构造函数实现  User::User(string username, string password, double balance, bool customer, bool seller, bool admin)      : username(username), password(password), balance(balance) {}  User::User()  {      username = "defaultUser";      password = "12345678";      balance = 0.0;      userType = "customer"; // 默认用户类型为消费者  }  // Customer 构造函数实现  Customer::Customer(string username, string password, double balance)      : User(username, password, balance, true, false, false) {}  Customer::Customer() : User()  {      userType = "customer"; // 默认用户类型为消费者  }  // Seller 构造函数实现  Seller::Seller(string username, string password, double balance)      : User(username, password, balance, false, true, false) {}  Seller::Seller() : User()  {      userType = "seller"; // 默认用户类型为商家  }  // Admin 构造函数实现  Admin::Admin(string username, string password, double balance)      : User(username, password, balance, false, false, true) {}  Admin::Admin() : User()  {      userType = "admin"; // 默认用户类型为管理员  }  // --- 静态方法：用户注册实现 ---  User \*User::registerUser(vector<User \*> &users)  {      string username, password;      int userTypeChoice;      cout << "\n--- 用户注册 ---" << endl;      // 获取并验证用户名      while (true)      {          cout << "请输入用户名: ";          getline(cin >> ws, username); // ws 忽略前导空格          if (username.empty())          {              cout << "用户名不能为空，请重新输入！" << endl;              continue;          }          // 调用 User 类的静态方法检查用户名是否存在          if (User::isUsernameExists(users, username))          {              cout << "该用户名已存在，请选择其他用户名！" << endl;              continue;          }          break;      }      // 获取密码      cout << "请输入密码: ";      getline(cin >> ws, password);      // 选择用户类型      cout << "请选择用户类型 (1: 消费者, 2: 商家): ";      while (!(cin >> userTypeChoice) || (userTypeChoice != 1 && userTypeChoice != 2))      {          cout << "无效选择，请输入 1 或 2: ";          User::clearInputBuffer(); // 调用 User 类的静态 clearInputBuffer      }      User::clearInputBuffer(); // 清除 cin 留下的换行符      User \*newUser = nullptr;      // 根据选择创建 Customer 或 Seller 对象 (User 的派生类)      if (userTypeChoice == 1)      {          newUser = new Customer(username, password);      }      else // userTypeChoice == 2      {          newUser = new Seller(username, password);      }      users.push\_back(newUser); // 将新用户指针添加到向量      // 注意：这里不再自动保存文件，让调用者决定何时保存      // User::saveUsersToFile(users, USER\_FILE); // 移除或注释掉      cout << "注册成功！" << endl;      return newUser; // 返回新注册的用户指针  }  // --- 静态方法：用户登录实现 ---  User \*User::userLogin(const vector<User \*> &users)  {      string username, password;      cout << "\n--- 用户登录 ---" << endl;      cout << "请输入用户名: ";      getline(cin >> ws, username);      cout << "请输入密码: ";      getline(cin >> ws, password);      User \*user = findUser(users, username);      if (user == nullptr)      {          cout << "用户不存在！" << endl;          return nullptr;      }      if (user->getPassword() != password)      {          cout << "密码错误！" << endl;          return nullptr;      }      cout << "登录成功！" << endl;      cout << "欢迎回来，" << user->getUsername() << "！" << endl;      return user; // 返回登录成功的用户指针  }  // 显示单个用户信息  void User::displayInfo(int index) const  {      if (index > 0)      {          cout << "用户 " << index << ":" << endl;      }      cout << "  用户名: " << username << endl;      cout << "  密码: " << password << endl;      cout << "  用户类型: " << getUserType() << endl;      cout << "  账户余额: " << balance << endl;      cout << "---------------------" << endl;      // 需改为按下enter也能返回      cout << "按任意键返回" << endl;      cin.get();                                           // 等待用户输入      cin.ignore(numeric\_limits<streamsize>::max(), '\n'); // 忽略剩余输入  }  // 静态方法：显示所有用户信息  void User::displayAllUsers(const vector<User \*> &users)  {      cout << "\n--- 所有用户信息 ---" << endl;      if (users.empty())      {          cout << "当前没有已注册的用户。" << endl;      }      else      {          for (size\_t i = 0; i < users.size(); ++i)          {              users[i]->displayInfo(i + 1);          }      }  }  // 修改密码  bool User::changePassword(const string &oldPassword, const string &newPassword)  {      if (password != oldPassword)      {          cout << "原密码错误！" << endl;          return false;      }      password = newPassword;      cout << "密码修改成功！" << endl;      return true;  }  // 充值余额  bool User::deposit(double amount)  {      if (amount <= 0)      {          cout << "充值金额必须大于0！" << endl;          return false;      }      balance += amount;      // cout << "充值成功！当前余额: " << balance << endl;      return true;  }  // 消费余额  bool User::withdraw(double amount)  {      if (amount <= 0)      {          cout << "消费金额必须大于0！" << endl;          return false;      }      if (balance < amount)      {          cout << "余额不足！当前余额: " << balance << endl;          return false;      }      balance -= amount;      cout << "支付成功！当前余额: " << balance << endl;      return true;  }  // 查询余额  double User::checkBalance() const  {      return balance;  }  // 将用户信息保存到文件  bool User::saveUsersToFile(const vector<User \*> &users, const string &filename)  {      ofstream file(filename);      if (!file.is\_open())      {          cerr << "无法打开文件: " << filename << endl;          return false;      }      for (const auto &user : users)      {          file << user->getUserType() << ","               << user->getUsername() << ","               << user->getPassword() << ","               << user->checkBalance() << endl;      }      file.close();      return true;  }  // 从文件加载用户信息  vector<User \*> User::loadUsersFromFile(const string &filename)  {      vector<User \*> users;      ifstream file(filename);      if (!file.is\_open())      {          cerr << "无法打开文件: " << filename << " (可能是首次运行)" << endl;          return users;      }      string line;      while (getline(file, line))      {          stringstream ss(line);          string userType, username, password;          double balance;          getline(ss, userType, ',');          getline(ss, username, ',');          getline(ss, password, ',');          ss >> balance;          User \*user = nullptr;          if (userType == "customer")          {              user = new Customer(username, password, balance);          }          else if (userType == "seller")          {              user = new Seller(username, password, balance);          }          else if (userType == "admin")          {              user = new Admin(username, password, balance);          }          if (user != nullptr)          {              users.push\_back(user);          }      }      file.close();      return users;  }  // 查找指定用户名的用户  User \*User::findUser(const vector<User \*> &users, const string &username)  {      for (auto user : users)      {          if (user->getUsername() == username)          {              return user;          }      }      return nullptr;  }  // 检查用户名是否已存在  bool User::isUsernameExists(const vector<User \*> &users, const string &username)  {      return findUser(users, username) != nullptr;  }  // 用户登录验证  User \*User::login(const vector<User \*> &users, const string &username, const string &password)  {      User \*user = findUser(users, username);      if (user && user->getPassword() == password)      {          return user;      }      return nullptr;  }  // Getter 函数实现  string User::getUsername() const { return username; }  string User::getPassword() const { return password; }  // Setter 函数实现  void User::setUsername(const string &newUsername) { username = newUsername; }  void User::setPassword(const string &newPassword) { password = newPassword; }  void User::setBalance(double newBalance) { balance = newBalance; } |

#### Store.h

|  |
| --- |
| #ifndef STORE\_H  #define STORE\_H  #include <string>  #include <vector>  #include <iostream>  #include <fstream>  #include <sstream>  #include <iomanip>  #include <limits>  #include <map>  #include <filesystem>  // 前向声明 User  class User;  // --- Product 基类和派生类 ---  class Product  {  protected:      std::string name;      std::string description;      double originalPrice;      int quantity;      double discountRate;      std::string sellerUsername; // 添加商品所属商家  public:      Product(std::string name, std::string desc, double price, int qty, std::string seller = "")          : name(name), description(desc), originalPrice(price), quantity(qty),            discountRate(0.0), sellerUsername(seller) {}      virtual ~Product() = default;      virtual double getPrice() const { return originalPrice \* (1.0 - discountRate); }      virtual void display() const;      virtual std::string getType() const = 0;      virtual void save(std::ofstream &ofs) const;      // Getters and Setters      std::string getName() const { return name; }      std::string getDescription() const { return description; }      double getOriginalPrice() const { return originalPrice; }      int getQuantity() const { return quantity; }      double getDiscountRate() const { return discountRate; }      std::string getSellerUsername() const { return sellerUsername; } // 获取商品所属商家      void setName(const std::string &newName) { name = newName; }      void setDescription(const std::string &newDesc) { description = newDesc; }      void setOriginalPrice(double newPrice)      {          if (newPrice >= 0)              originalPrice = newPrice;      }      void setQuantity(int newQuantity)      {          if (newQuantity >= 0)              quantity = newQuantity;      }      void setDiscountRate(double newRate)      {          if (newRate >= 0.0 && newRate <= 1.0)              discountRate = newRate;      }      void setSellerUsername(const std::string &seller) { sellerUsername = seller; } // 设置商品所属商家  };  class Book : public Product  {  private:      std::string author;      std::string isbn;  public:      Book(std::string name, std::string desc, double price, int qty,           std::string auth, std::string isbn\_num, std::string seller = "")          : Product(name, desc, price, qty, seller), author(auth), isbn(isbn\_num) {}      std::string getType() const override { return "Book"; }      void display() const override;      void save(std::ofstream &ofs) const override;      // 新增的 getters      std::string getAuthor() const { return author; }      std::string getIsbn() const { return isbn; }  };  class Clothing : public Product  {  private:      std::string size;      std::string color;  public:      Clothing(std::string name, std::string desc, double price, int qty,               std::string sz, std::string clr, std::string seller = "")          : Product(name, desc, price, qty, seller), size(sz), color(clr) {}      std::string getType() const override { return "Clothing"; }      void display() const override;      void save(std::ofstream &ofs) const override;      // 新增的 getters      std::string getSize() const { return size; }      std::string getColor() const { return color; }  };  class Food : public Product  {  private:      std::string expirationDate;  public:      Food(std::string name, std::string desc, double price, int qty,           std::string expDate, std::string seller = "")          : Product(name, desc, price, qty, seller), expirationDate(expDate) {}      std::string getType() const override { return "Food"; }      void display() const override;      void save(std::ofstream &ofs) const override;      // 新增的 getters      std::string getExpirationDate() const { return expirationDate; }  };  // --- Store Class ---  class Store  {  private:      std::vector<Product \*> allProducts;                           // 存储所有商家的商品      std::map<std::string, std::vector<Product \*>> sellerProducts; // 每个商家的商品映射      std::string storeDirectory;                                   // 商品文件所在目录      // 辅助方法      std::string getSellerFilename(const std::string &username) const;      bool saveProductsForSeller(const std::string &sellerUsername);      bool ensureDirectoryExists(const std::string &path) const;  public:      // 构造函数接收目录名      Store(const std::string &directory);      ~Store();      Product \*findProductByName(const std::string &name, const std::string &sellerUsername = "");      // 文件操作      bool loadAllProducts();      bool loadSellerProducts(const std::string &sellerUsername);      bool saveAllProducts();      // 显示功能      void displayAllProducts() const;                                     // 显示所有商品      void displaySellerProducts(const std::string &sellerUsername) const; // 显示指定商家的商品      // 搜索功能      std::vector<Product \*> searchProductsByName(const std::string &searchTerm, const std::string &sellerUsername = "") const;      // 商家商品管理功能      bool createBook(User \*currentUser, const std::string &name, const std::string &desc,                      double price, int qty, const std::string &author, const std::string &isbn);      bool createClothing(User \*currentUser, const std::string &name, const std::string &desc,                          double price, int qty, const std::string &size, const std::string &color);      bool createFood(User \*currentUser, const std::string &name, const std::string &desc,                      double price, int qty, const std::string &expDate);      bool manageProductPrice(User \*currentUser, const std::string &productName, double newPrice);      bool manageProductQuantity(User \*currentUser, const std::string &productName, int newQuantity);      bool manageProductDiscount(User \*currentUser, const std::string &productName, double newDiscount);      bool applyCategoryDiscount(User \*currentUser, const std::string &category, double discount);      // 获取商品      const std::vector<Product \*> &getProducts() const { return allProducts; }      std::vector<Product \*> getSellerProducts(const std::string &sellerUsername) const;  };  #endif // STORE\_H |

#### Store.cpp

|  |
| --- |
| #include "store.h"  #include "../user/user.h"  #include <iostream>  #include <vector>  #include <string>  #include <fstream>  #include <sstream>  #include <iomanip>  #include <algorithm>  #include <filesystem>  #include <set>  #include <direct.h> // Windows 系统特定的目录操作  namespace fs = std::filesystem;  using namespace std;  // --- Product 方法实现 ---  void Product::display() const  {      std::cout << "  商户名称：" << sellerUsername << std::endl;      std::cout << "  商品名称: " << name << std::endl;      std::cout << "  描述: " << description << std::endl;      if (discountRate > 0)      {          std::cout << "  原价: ¥" << std::fixed << std::setprecision(2) << originalPrice << std::endl;          std::cout << "  折扣: " << (discountRate \* 100) << "%" << std::endl;          std::cout << "  现价: ¥" << std::fixed << std::setprecision(2) << getPrice() << std::endl;      }      else      {          std::cout << "  价格: ¥" << std::fixed << std::setprecision(2) << getPrice() << std::endl;      }      std::cout << "  库存: " << quantity << " 件" << std::endl;      if (!sellerUsername.empty())      {          std::cout << "  商家: " << sellerUsername << std::endl;      }  }  void Product::save(std::ofstream &ofs) const  {      ofs << getType() << "," << name << "," << description << ","          << originalPrice << "," << quantity << "," << discountRate << "," << sellerUsername;  }  void Book::display() const // 复用父类  {      Product::display();      std::cout << "  作者: " << author << std::endl;      std::cout << "  ISBN: " << isbn << std::endl;  }  void Book::save(std::ofstream &ofs) const  {      Product::save(ofs);      ofs << "," << author << "," << isbn;  }  void Clothing::display() const  {      Product::display();      std::cout << "  尺寸: " << size << std::endl;      std::cout << "  颜色: " << color << std::endl;  }  void Clothing::save(std::ofstream &ofs) const  {      Product::save(ofs);      ofs << "," << size << "," << color;  }  void Food::display() const  {      Product::display();      std::cout << "  过期日期: " << expirationDate << std::endl;  }  void Food::save(std::ofstream &ofs) const  {      Product::save(ofs);      ofs << "," << expirationDate;  }  // --- Store 类实现 ---  // 构造函数  Store::Store(const string &directory) : storeDirectory(directory)  {      // 确保商家目录存在      string sellersDir = storeDirectory + "/sellers";      ensureDirectoryExists(sellersDir);      // 加载所有商品数据      if (!loadAllProducts())      {          cerr << "警告: 无法加载商店数据。商店可能为空或文件丢失/损坏。" << endl;      }      else      {          cout << "已从商店加载 " << allProducts.size() << " 件商品。" << endl;      }  }  // 析构函数  Store::~Store()  {      for (Product \*p : allProducts)      {          delete p;      }      allProducts.clear();      sellerProducts.clear(); // 清除映射  }  // 确保目录存在  bool Store::ensureDirectoryExists(const string &path) const  {      try      {          if (!fs::exists(path))          {              fs::create\_directories(path);              cout << "创建目录: " << path << endl;          }          return true;      }      catch (const fs::filesystem\_error &e)      {          cerr << "创建目录失败 " << path << ": " << e.what() << endl;          // 尝试使用传统方法创建目录          if (\_mkdir(path.c\_str()) == 0)          {              cout << "使用传统方法创建目录: " << path << endl;              return true;          }          return false;      }      catch (...)      {          cerr << "创建目录时发生未知错误: " << path << endl;          return false;      }  }  // 获取商家文件名  string Store::getSellerFilename(const string &username) const  {      return storeDirectory + "/sellers/" + username + ".txt";  }  // 加载所有商品  bool Store::loadAllProducts()  {      // 清空现有商品      for (Product \*p : allProducts)      {          delete p;      }      allProducts.clear();      sellerProducts.clear();      // 获取sellers目录下的所有文件      string sellersDir = storeDirectory + "/sellers";      try      {          // 如果目录不存在，创建目录并返回成功（表示商店为空）          if (!fs::exists(sellersDir))          {              ensureDirectoryExists(sellersDir);              return true;          }          // 遍历目录中的所有文件          for (const auto &entry : fs::directory\_iterator(sellersDir))          {              if (entry.is\_regular\_file())              {                  string filename = entry.path().string();                  string sellerUsername = entry.path().stem().string(); // 获取不带扩展名的文件名                  loadSellerProducts(sellerUsername);              }          }          return true;      }      catch (const fs::filesystem\_error &e)      {          cerr << "读取商家目录失败: " << e.what() << endl;          return false;      }      catch (...)      {          cerr << "读取商家数据时发生未知错误" << endl;          return false;      }  }  // 加载指定商家的商品  bool Store::loadSellerProducts(const string &sellerUsername)  {      string filename = getSellerFilename(sellerUsername);      ifstream file(filename);      if (!file.is\_open())      {          // 对于新商家，文件不存在是正常的          return true;      }      vector<Product \*> sellerProductsList;      string line;      while (getline(file, line))      {          stringstream ss(line);          string segment;          vector<string> seglist;          while (getline(ss, segment, ','))          {              seglist.push\_back(segment);          }          // 格式检查: 至少需要7个字段 (类型,名称,描述,价格,数量,折扣,商家)          if (seglist.size() < 7)          {              cerr << "无效的产品数据行: " << line << endl;              continue;          }          try          {              string type = seglist[0];              string name = seglist[1];              string desc = seglist[2];              double price = stod(seglist[3]);              int qty = stoi(seglist[4]);              double discount = stod(seglist[5]);              string seller = seglist[6]; // 第7个字段是商家名              // 确认商家名称匹配              if (seller != sellerUsername)              {                  cerr << "警告: 文件 " << filename << " 中的商品所属商家 \""                       << seller << "\" 与预期的 \"" << sellerUsername << "\" 不符" << endl;                  seller = sellerUsername; // 强制使用文件名指示的商家              }              Product \*newProduct = nullptr;              if (type == "Book" && seglist.size() >= 9)              {                  // Book需要额外的author和isbn字段                  newProduct = new Book(name, desc, price, qty, seglist[7], seglist[8], seller);              }              else if (type == "Clothing" && seglist.size() >= 9)              {                  // Clothing需要额外的size和color字段                  newProduct = new Clothing(name, desc, price, qty, seglist[7], seglist[8], seller);              }              else if (type == "Food" && seglist.size() >= 8)              {                  // Food需要额外的expirationDate字段                  newProduct = new Food(name, desc, price, qty, seglist[7], seller);              }              if (newProduct)              {                  newProduct->setDiscountRate(discount);                  allProducts.push\_back(newProduct);                  sellerProductsList.push\_back(newProduct);              }          }          catch (const exception &e)          {              cerr << "解析商品数据时出错: " << e.what() << " 行: " << line << endl;          }      }      file.close();      // 更新商家商品映射      sellerProducts[sellerUsername] = sellerProductsList;      cout << "已加载商家 \"" << sellerUsername << "\" 的 "           << sellerProductsList.size() << " 件商品" << endl;      return true;  }  // 保存所有商品  bool Store::saveAllProducts()  {      // 对于每个商家，保存其商品      set<string> processedSellers; // 用于跟踪已处理的商家      for (const auto &product : allProducts)      {          string seller = product->getSellerUsername();          if (!seller.empty() && processedSellers.find(seller) == processedSellers.end())          {              saveProductsForSeller(seller);              processedSellers.insert(seller);          }      }      return true;  }  // 保存指定商家的商品  bool Store::saveProductsForSeller(const string &sellerUsername)  {      if (sellerUsername.empty())      {          cerr << "错误: 尝试保存无效商家的商品" << endl;          return false;      }      vector<Product \*> products;      // 获取该商家的所有商品      for (Product \*p : allProducts)      {          if (p->getSellerUsername() == sellerUsername)          {              products.push\_back(p);          }      }      string filename = getSellerFilename(sellerUsername);      ofstream file(filename);      if (!file.is\_open())      {          cerr << "错误: 无法打开文件保存商品: " << filename << endl;          return false;      }      for (const Product \*p : products)      {          p->save(file);          file << endl;      }      file.close();      // cout << "商家 \"" << sellerUsername << "\" 的 " << products.size()      //      << " 件商品已保存至 " << filename << endl;      return true;  }  // 查找商品  Product \*Store::findProductByName(const string &name, const string &sellerUsername)  {      if (!sellerUsername.empty())      {          // 只在指定商家的商品中查找          auto it = sellerProducts.find(sellerUsername);          if (it != sellerProducts.end())          {              for (Product \*p : it->second)              {                  if (p->getName() == name)                  {                      return p;                  }              }          }          return nullptr;      }      else      {          // 在所有商品中查找          for (Product \*p : allProducts)          {              if (p->getName() == name)              {                  return p;              }          }          return nullptr;      }  }  // 显示所有商品  void Store::displayAllProducts() const  {      if (allProducts.empty())      {          cout << "\n--- 商品列表为空 ---" << endl;          return;      }      cout << "\n--- 所有商品列表 ---" << endl;      int count = 0;      for (const Product \*p : allProducts)      {          cout << "--------------------------" << endl;          p->display();          count++;      }      cout << "--------------------------" << endl;      cout << "共显示 " << count << " 件商品" << endl;  }  // 显示指定商家的商品  void Store::displaySellerProducts(const string &sellerUsername) const  {      auto it = sellerProducts.find(sellerUsername);      if (it == sellerProducts.end() || it->second.empty())      {          cout << "\n--- 商家 \"" << sellerUsername << "\" 没有商品 ---" << endl;          return;      }      cout << "\n--- 商家 \"" << sellerUsername << "\" 的商品列表 ---" << endl;      int count = 0;      for (const Product \*p : it->second)      {          cout << "--------------------------" << endl;          p->display();          count++;      }      cout << "--------------------------" << endl;      cout << "共显示 " << count << " 件商品" << endl;  }  // 按名称搜索商品  vector<Product \*> Store::searchProductsByName(const string &searchTerm, const string &sellerUsername) const  {      vector<Product \*> results;      const vector<Product \*> &productsToSearch = sellerUsername.empty() ? allProducts : (sellerProducts.count(sellerUsername) ? sellerProducts.at(sellerUsername) : vector<Product \*>());      for (Product \*p : productsToSearch)      {          // 简单的包含搜索 (不区分大小写)          string productName = p->getName();          string searchTermLower = searchTerm;          // 将两个字符串转为小写进行比较          transform(productName.begin(), productName.end(), productName.begin(), ::tolower);          transform(searchTermLower.begin(), searchTermLower.end(), searchTermLower.begin(), ::tolower);          if (productName.find(searchTermLower) != string::npos)          {              results.push\_back(p);          }      }      return results;  }  // 获取商家的商品  vector<Product \*> Store::getSellerProducts(const string &sellerUsername) const  {      auto it = sellerProducts.find(sellerUsername);      return (it != sellerProducts.end()) ? it->second : vector<Product \*>();  }  // 创建图书  bool Store::createBook(User \*currentUser, const string &name, const string &desc,                         double price, int qty, const string &author, const string &isbn)  {      if (!currentUser)      {          cerr << "错误: 用户未登录" << endl;          return false;      }      if (currentUser->getUserType() != "seller")      {          cerr << "错误: 只有商家可以添加商品" << endl;          return false;      }      string sellerUsername = currentUser->getUsername();      // 检查该商家是否已有同名商品      if (findProductByName(name, sellerUsername) != nullptr)      {          cerr << "错误: 您已有名为 \"" << name << "\" 的商品" << endl;          return false;      }      if (price < 0 || qty < 0)      {          cerr << "错误: 价格和数量不能为负数" << endl;          return false;      }      try      {          Book \*newBook = new Book(name, desc, price, qty, author, isbn, sellerUsername);          allProducts.push\_back(newBook);          // 更新商家商品映射          sellerProducts[sellerUsername].push\_back(newBook);          // 保存商家的商品          if (saveProductsForSeller(sellerUsername))          {              cout << "商品 \"" << name << "\" 添加成功！" << endl;              return true;          }          else          {              cout << "商品 \"" << name << "\" 添加失败！" << endl;              return false;          }      }      catch (const bad\_alloc &e)      {          cerr << "错误: 无法分配内存: " << e.what() << endl;          return false;      }      catch (const exception &e)      {          cerr << "错误: " << e.what() << endl;          return false;      }      catch (...)      {          cerr << "错误: 添加商品时发生未知错误" << endl;          return false;      }  }  // 创建服装  bool Store::createClothing(User \*currentUser, const string &name, const string &desc,                             double price, int qty, const string &size, const string &color)  {      if (!currentUser)      {          cerr << "错误: 用户未登录" << endl;          return false;      }      if (currentUser->getUserType() != "seller")      {          cerr << "错误: 只有商家可以添加商品" << endl;          return false;      }      string sellerUsername = currentUser->getUsername();      // 检查该商家是否已有同名商品      if (findProductByName(name, sellerUsername) != nullptr)      {          cerr << "错误: 您已有名为 \"" << name << "\" 的商品" << endl;          return false;      }      if (price < 0 || qty < 0)      {          cerr << "错误: 价格和数量不能为负数" << endl;          return false;      }      try      {          Clothing \*newClothing = new Clothing(name, desc, price, qty, size, color, sellerUsername);          allProducts.push\_back(newClothing);          // 更新商家商品映射          sellerProducts[sellerUsername].push\_back(newClothing);          // 保存商家的商品          if (saveProductsForSeller(sellerUsername))          {              cout << "商品 \"" << name << "\" 添加成功！" << endl;              return true;          }          else          {              cout << "商品 \"" << name << "\" 添加失败！" << endl;              return false;          }      }      catch (const bad\_alloc &e)      {          cerr << "错误: 无法分配内存: " << e.what() << endl;          return false;      }      catch (const exception &e)      {          cerr << "错误: " << e.what() << endl;          return false;      }      catch (...)      {          cerr << "错误: 添加商品时发生未知错误" << endl;          return false;      }  }  // 创建食品  bool Store::createFood(User \*currentUser, const string &name, const string &desc,                         double price, int qty, const string &expDate)  {      if (!currentUser)      {          cerr << "错误: 用户未登录" << endl;          return false;      }      if (currentUser->getUserType() != "seller")      {          cerr << "错误: 只有商家可以添加商品" << endl;          return false;      }      string sellerUsername = currentUser->getUsername();      // 检查该商家是否已有同名商品      if (findProductByName(name, sellerUsername) != nullptr)      {          cerr << "错误: 您已有名为 \"" << name << "\" 的商品" << endl;          return false;      }      if (price < 0 || qty < 0)      {          cerr << "错误: 价格和数量不能为负数" << endl;          return false;      }      try      {          Food \*newFood = new Food(name, desc, price, qty, expDate, sellerUsername);          allProducts.push\_back(newFood);          // 更新商家商品映射          sellerProducts[sellerUsername].push\_back(newFood);          // 保存商家的商品          if (saveProductsForSeller(sellerUsername))          {              cout << "商品 \"" << name << "\" 添加成功！" << endl;              return true;          }          else          {              cout << "商品 \"" << name << "\" 添加失败！" << endl;              return false;          }      }      catch (const bad\_alloc &e)      {          cerr << "错误: 无法分配内存: " << e.what() << endl;          return false;      }      catch (const exception &e)      {          cerr << "错误: " << e.what() << endl;          return false;      }      catch (...)      {          cerr << "错误: 添加商品时发生未知错误" << endl;          return false;      }  }  // 修改商品价格  bool Store::manageProductPrice(User \*currentUser, const string &productName, double newPrice)  {      if (!currentUser)      {          cerr << "错误: 用户未登录" << endl;          return false;      }      if (currentUser->getUserType() != "seller")      {          cerr << "错误: 只有商家可以修改商品价格" << endl;          return false;      }      string sellerUsername = currentUser->getUsername();      Product \*product = findProductByName(productName, sellerUsername);      if (!product)      {          cerr << "错误: 未找到您的商品 \"" << productName << "\"" << endl;          return false;      }      if (newPrice < 0)      {          cerr << "错误: 价格不能为负数" << endl;          return false;      }      product->setOriginalPrice(newPrice);      // 保存商家的商品      if (saveProductsForSeller(sellerUsername))      {          cout << "价格保存成功！" << endl;          return true;      }      else      {          cout << "价格保存失败！" << endl;          return false;      }  }  // 修改商品库存  bool Store::manageProductQuantity(User \*currentUser, const string &productName, int newQuantity)  {      if (!currentUser)      {          cerr << "错误: 用户未登录" << endl;          return false;      }      if (currentUser->getUserType() != "seller")      {          cerr << "错误: 只有商家可以修改商品库存" << endl;          return false;      }      string sellerUsername = currentUser->getUsername();      Product \*product = findProductByName(productName, sellerUsername);      if (!product)      {          cerr << "错误: 未找到您的商品 \"" << productName << "\"" << endl;          return false;      }      if (newQuantity < 0)      {          cerr << "错误: 库存不能为负数" << endl;          return false;      }      product->setQuantity(newQuantity);      if (saveProductsForSeller(sellerUsername))      {          cout << "库存修改成功！" << endl;          return true;      }      else      {          cout << "库存修改失败！" << endl;          return false;      }  }  // 修改商品折扣  bool Store::manageProductDiscount(User \*currentUser, const string &productName, double newDiscount)  {      if (!currentUser)      {          cerr << "错误: 用户未登录" << endl;          return false;      }      if (currentUser->getUserType() != "seller")      {          cerr << "错误: 只有商家可以修改商品折扣" << endl;          return false;      }      string sellerUsername = currentUser->getUsername();      Product \*product = findProductByName(productName, sellerUsername);      if (!product)      {          cerr << "错误: 未找到您的商品 \"" << productName << "\"" << endl;          return false;      }      if (newDiscount < 0 || newDiscount > 100)      {          cerr << "错误: 折扣必须在0到100之间" << endl;          return false;      }      product->setDiscountRate(newDiscount);      return saveProductsForSeller(sellerUsername);  }  // 应用分类折扣  bool Store::applyCategoryDiscount(User \*currentUser, const string &category, double discount)  {      if (!currentUser)      {          cerr << "错误: 用户未登录" << endl;          return false;      }      if (currentUser->getUserType() != "seller")      {          cerr << "错误: 只有商家可以应用折扣" << endl;          return false;      }      if (discount < 0.0 || discount > 1.0)      {          cerr << "错误: 折扣必须在0到100之间" << endl;          return false;      }      string sellerUsername = currentUser->getUsername();      bool changed = false;      // 只应用折扣到该商家的商品      auto it = sellerProducts.find(sellerUsername);      if (it != sellerProducts.end())      {          for (Product \*p : it->second)          {              if (p->getType() == category)              {                  p->setDiscountRate(discount);                  changed = true;              }          }      }      if (changed)      {          return saveProductsForSeller(sellerUsername);      }      cerr << "错误: 未找到分类为 \"" << category << "\" 的商品" << endl;      return false;  } |

#### Page.h

|  |
| --- |
| #ifndef PAGE\_H  #define PAGE\_H  #include "../store/store.h" // 包含 Store 类定义  #include <vector>  #include <string> // 需要包含 string  // 前向声明 User 类，因为 Page 方法使用了 User\*  // 避免在头文件中包含 user.h，减少编译依赖  class User;  // --- Page 类定义 ---  // 用于管理不同状态下的菜单显示  class Page  {  protected:      // 私有静态辅助方法：清除输入缓冲区 (Page 类内部使用)      // 定义移到 page.cpp      static void clearInputBuffer();  public:      // --- 成员方法：菜单显示 ---      // 主菜单 (未登录)      int pagemain(std::vector<User \*> &users, User \*&currentUser);      // 消费者菜单 (登录后)      int pagemainCustomer(User \*currentUser, std::vector<User \*> &users);      // 商家菜单 (登录后)      int pagemainSeller(User \*currentUser, std::vector<User \*> &users);      // 管理员菜单 (如果需要可以添加)      int pagemainAdmin(User \*currentUser, std::vector<User \*> &users);      virtual int pagestore(User \*currentUser, std::vector<User \*> &users, Store &store) = 0;  };  // --- Page 派生类 ---  class Customer\_page : public Page  {  public:      // 实现商城页面 (确保有 Store& store 参数)      int pagestore(User \*currentUser, std::vector<User \*> &users, Store &store) override;      // 可以覆盖基类的账户菜单方法，如果需要特定行为  };  class Seller\_page : public Page  {  public:      // 实现商城页面 (确保有 Store& store 参数)      int pagestore(User \*currentUser, std::vector<User \*> &users, Store &store) override;      // 可以覆盖基类的账户菜单方法，如果需要特定行为  };  // //应该暂不需要  // class Admin\_page : public Page  // {  //     int pagestore(User \*currentUser, std::vector<User \*> &users);  // };  #endif // PAGE\_H |

#### Page.cpp

|  |
| --- |
| #include "page.h"  #include "../user/user.h"  #include "../store/store.h" // 需要 Store 定义  #include <iostream>  #include <cstdio> // for printf if used  #include <limits>  #include <vector>  #include <string>  using namespace std;  extern const string USER\_FILE;  // STORE\_FILE is managed by the Store object passed in  // --- 基类 Page 静态方法 ---  void Page::clearInputBuffer()  {      cin.clear();      cin.ignore(numeric\_limits<streamsize>::max(), '\n');  }  // --- 基类 Page 菜单实现 ---  // 这些函数现在只处理用户账户菜单，并在选择商城时返回特定值 3  int Page::pagemain(vector<User \*> &users, User \*&currentUser)  {      int choice = 0;      while (true)      {          printf("\n\n\n--- 主菜单 ---\n");          cout << "1. 注册新用户" << endl;          cout << "2. 用户登录" << endl;          cout << "3. 进入商城 (游客)" << endl; // 返回 3 表示进入商城          cout << "4. 退出程序" << endl;          cout << "请选择一个选项: ";          if (!(cin >> choice))          {              cout << "无效输入。" << endl;              clearInputBuffer();              continue;          }          clearInputBuffer();          switch (choice)          {          case 1:              currentUser = User::registerUser(users);              if (currentUser)              {                  User::saveUsersToFile(users, USER\_FILE);                  // 注册成功后可以选择留在主菜单或直接进入用户菜单                  // return 1; // 如果想直接进入用户菜单              }              return 1;              break; // 留在主菜单          case 2:              currentUser = User::userLogin(users);              if (currentUser)                  return 1; // 登录成功，返回 1 给 main 处理              break;        // 登录失败，留在主菜单          case 3:              return 3; // 返回 3，通知 main 调用 pagestore          case 4:              cout << "退出程序..." << endl;              return 0; // 返回 0 给 main 退出循环          default:              cout << "无效选项。" << endl;          }      }  }  int Page::pagemainCustomer(User \*currentUser, vector<User \*> &users)  {      int choice = 0;      while (true)      {          printf("\n\n\n--- 消费者菜单 (%s) ---\n", currentUser->getUsername().c\_str());          cout << "1. 查看个人信息" << endl;          cout << "2. 修改密码" << endl;          cout << "3. 查询余额" << endl;          cout << "4. 充值" << endl;          cout << "5. 进入商城" << endl; // 返回 3 表示进入商城          cout << "6. 退出登录" << endl;          cout << "请选择一个选项: ";          if (!(cin >> choice))          {              cout << "无效输入。" << endl;              clearInputBuffer();              continue;          }          clearInputBuffer();          switch (choice)          {          case 1:              currentUser->displayInfo();              break;          case 2:          {              string oldP, newP;              cout << "请输入原密码: ";              getline(cin >> ws, oldP);              cout << "请输入新密码: ";              getline(cin >> ws, newP);              if (currentUser->changePassword(oldP, newP))              {                  User::saveUsersToFile(users, USER\_FILE);                  cout << "密码修改成功。" << endl;              }              else              {                  cout << "密码修改失败。" << endl;              }          }          break;          case 3:              cout << "当前余额: " << currentUser->checkBalance() << endl;              break;          case 4:          {              double amt;              cout << "请输入充值金额: ";              if (!(cin >> amt) || amt <= 0)              { // 检查无效输入和非正数                  cout << "无效金额。" << endl;                  clearInputBuffer();                  break;              }              clearInputBuffer();              if (currentUser->deposit(amt))              {                  User::saveUsersToFile(users, USER\_FILE);                  cout << "充值成功。当前余额: " << currentUser->checkBalance() << endl;              }              else              {                  // deposit 内部应该已经打印了错误信息              }          }          break;          case 5:              return 3; // 返回 3，通知 main 调用 pagestore          case 6:              cout << "退出登录..." << endl;              return 2; // 返回 2 给 main 处理退出登录          default:              cout << "无效选项。" << endl;          }      }  }  int Page::pagemainSeller(User \*currentUser, vector<User \*> &users)  {      int choice = 0;      while (true)      {          printf("\n\n\n--- 商家菜单 (%s) ---\n", currentUser->getUsername().c\_str());          cout << "1. 查看商户信息" << endl;          cout << "2. 修改密码" << endl;          cout << "3. 进入商城 (管理)" << endl; // 返回 3 表示进入商城          cout << "4. 退出登录" << endl;          cout << "请选择一个选项: ";          if (!(cin >> choice))          {              cout << "无效输入。" << endl;              clearInputBuffer();              continue;          }          clearInputBuffer();          switch (choice)          {          case 1:              currentUser->displayInfo();              break;          case 2:          {              string oldP, newP;              cout << "请输入原密码: ";              getline(cin >> ws, oldP);              cout << "请输入新密码: ";              getline(cin >> ws, newP);              if (currentUser->changePassword(oldP, newP))              {                  User::saveUsersToFile(users, USER\_FILE);                  cout << "密码修改成功。" << endl;              }              else              {                  cout << "密码修改失败。" << endl;              }          }          break;          case 3:              return 3; // 返回 3，通知 main 调用 pagestore          case 4:              cout << "退出登录..." << endl;              return 2; // 返回 2 给 main 处理退出登录          default:              cout << "无效选项。" << endl;          }      }  }  int Page::pagemainAdmin(User \*currentUser, vector<User \*> &users)  {      // 管理员菜单实现      int choice = 0;      while (true)      {          printf("\n\n\n--- 管理员菜单 (%s) ---\n", currentUser->getUsername().c\_str());          cout << "1. 查看所有用户信息" << endl;          // 可以添加其他管理员功能          cout << "2. 退出登录" << endl;          cout << "请选择一个选项: ";          if (!(cin >> choice))          {              cout << "无效输入。" << endl;              clearInputBuffer();              continue;          }          clearInputBuffer();          switch (choice)          {          case 1:              User::displayAllUsers(users); // 假设 User 类有这个静态方法              break;          case 2:              cout << "退出登录..." << endl;              return 2; // 返回 2 给 main 处理退出登录          default:              cout << "无效选项。" << endl;          }      }  }  // --- Customer\_page 商城实现 ---  int Customer\_page::pagestore(User \*currentUser, std::vector<User \*> &users, Store &store)  {      int choice = -1;      string username = currentUser ? currentUser->getUsername() : "游客";      do      {          cout << "\n--- 商城 (用户: " << username << ") ---" << endl;          cout << "1. 显示所有商品" << endl;          cout << "2. 按名称搜索商品" << endl;          cout << "3. 购买商品" << endl; // 新增选项          cout << "0. 退出商城" << endl;          cout << "请选择: ";          if (!(cin >> choice))          {              cout << "无效输入。" << endl;              clearInputBuffer();              choice = -1;              continue;          }          clearInputBuffer();          switch (choice)          {          case 1:              store.displayAllProducts();              break;          case 2:          {              string searchTerm;              cout << "输入搜索名称: ";              getline(cin >> ws, searchTerm);              vector<Product \*> results = store.searchProductsByName(searchTerm);              if (results.empty())              {                  cout << "未找到匹配商品。" << endl;              }              else              {                  cout << "\n--- 搜索结果 ---" << endl;                  for (const auto \*p : results)                  {                      cout << "-----------------" << endl;                      p->display();                  }                  cout << "-----------------" << endl;              }              break;          }          case 3: // 购买商品逻辑          {              if (!currentUser)              {                  cout << "请先登录后再购买商品！" << endl;                  break;              }              string productName;              int quantity;              cout << "输入要购买的商品名称: ";              getline(cin >> ws, productName);              cout << "输入购买数量: ";              if (!(cin >> quantity) || quantity <= 0)              {                  cout << "无效的购买数量！" << endl;                  clearInputBuffer();                  break;              }              clearInputBuffer();              Product \*product = store.findProductByName(productName);              if (!product)              {                  cout << "未找到商品 \"" << productName << "\"！" << endl;                  break;              }              double totalCost = product->getPrice() \* quantity;              if (product->getQuantity() < quantity)              {                  cout << "库存不足！当前库存: " << product->getQuantity() << endl;                  break;              }              if (currentUser->checkBalance() < totalCost)              {                  cout << "余额不足！当前余额: " << currentUser->checkBalance() << "，需要: " << totalCost << endl;                  break;              }              // 扣除用户余额并减少商品库存              currentUser->withdraw(totalCost);              string sellername = product->getSellerUsername();              User \*seller = nullptr;              if (!sellername.empty())              {                  for (User \*u : users)                  {                      if (u->getUsername() == sellername)                      {                          seller = u;                          break;                      }                  }              }              if (seller)              {                  if (!seller->deposit(totalCost))                  {                      cout << "支付给商家失败！交易失败！" << endl;                      break;                  }              }              else if (!sellername.empty())              {                  cout << "未找到商品所属商家 \"" << sellername << "\"！" << endl;                  break;              }              else              {                  cout << "商品没有商家信息，无法支付！" << endl;                  break;              }              product->setQuantity(product->getQuantity() - quantity);              // 保存用户数据和商品数据              User::saveUsersToFile(users, USER\_FILE);              store.saveAllProducts();              cout << "购买成功！已购买 " << quantity << " 件 \"" << productName << "\"，总花费: " << totalCost << endl;              break;          }          case 0:              cout << "正在退出商城..." << endl;              break;          default:              cout << "无效选项。" << endl;          }      } while (choice != 0);      return 1; // 返回 1 表示正常退出商城，继续主循环  }  // 只替换Seller\_page::pagestore方法，保留文件其他部分不变  // --- Seller\_page 商城实现 ---  int Seller\_page::pagestore(User \*currentUser, std::vector<User \*> &users, Store &store)  {      // 确保当前用户是商家      if (!currentUser || currentUser->getUserType() != "seller")      {          cout << "错误：只有商家才能访问商家管理页面。" << endl;          return 1; // 直接退出      }      string sellerUsername = currentUser->getUsername();      int choice = -1;      do      {          cout << "\n--- 商城管理 (商家: " << sellerUsername << ") ---" << endl;          cout << "1. 显示我的商品" << endl;          cout << "2. 按名称搜索我的商品" << endl;          cout << "3. 添加新商品" << endl;          cout << "4. 修改商品价格" << endl;          cout << "5. 修改商品库存" << endl;          cout << "6. 修改商品折扣" << endl;          cout << "7. 应用分类折扣" << endl;          cout << "0. 退出商城管理" << endl;          cout << "请选择: ";          if (!(cin >> choice))          {              cout << "无效输入。" << endl;              clearInputBuffer();              choice = -1;              continue;          }          clearInputBuffer();          switch (choice)          {          case 1:              store.displaySellerProducts(sellerUsername);              break;          case 2:          {              string searchTerm;              cout << "输入搜索名称: ";              getline(cin >> ws, searchTerm);              vector<Product \*> results = store.searchProductsByName(searchTerm, sellerUsername);              if (results.empty())              {                  cout << "未找到匹配商品。" << endl;              }              else              {                  cout << "\n--- 搜索结果 ---" << endl;                  for (const auto \*p : results)                  {                      cout << "-----------------" << endl;                      p->display();                  }                  cout << "-----------------" << endl;              }              break;          }          case 3:          {              // --- 添加商品逻辑 ---              int typeChoice;              cout << "选择商品类型 (1: 图书, 2: 服装, 3: 食品): ";              if (!(cin >> typeChoice) || typeChoice < 1 || typeChoice > 3)              {                  cout << "无效类型。" << endl;                  clearInputBuffer();                  break;              }              clearInputBuffer();              string name, desc, spec1, spec2;              double price;              int qty;              cout << "输入商品名称: ";              getline(cin >> ws, name);              cout << "输入描述: ";              getline(cin >> ws, desc);              cout << "输入价格: ";              while (!(cin >> price) || price < 0)              {                  cout << "无效价格\n";                  clearInputBuffer();                  cout << "输入价格: ";              }              clearInputBuffer();              cout << "输入库存数量: ";              while (!(cin >> qty) || qty < 0)              {                  cout << "无效数量\n";                  clearInputBuffer();                  cout << "输入库存数量: ";              }              clearInputBuffer();              bool success = false;              if (typeChoice == 1)              { // Book                  cout << "输入作者: ";                  getline(cin >> ws, spec1);                  cout << "输入ISBN: ";                  getline(cin >> ws, spec2);                  success = store.createBook(currentUser, name, desc, price, qty, spec1, spec2);              }              else if (typeChoice == 2)              { // Clothing                  cout << "输入尺码: ";                  getline(cin >> ws, spec1);                  cout << "输入颜色: ";                  getline(cin >> ws, spec2);                  success = store.createClothing(currentUser, name, desc, price, qty, spec1, spec2);              }              else              { // Food                  cout << "输入过期日期 (YYYY-MM-DD): ";                  getline(cin >> ws, spec1);                  success = store.createFood(currentUser, name, desc, price, qty, spec1);              }              if (success)                  cout << "商品 \"" << name << "\" 添加成功！" << endl;              else                  cout << "商品添加失败！" << endl;              break;          }          case 4:          { // 修改价格              string name;              double price;              cout << "输入要修改价格的商品名称: ";              getline(cin >> ws, name);              cout << "输入新价格: ";              if (!(cin >> price) || price < 0)              {                  cout << "无效价格\n";                  clearInputBuffer();                  break;              }              clearInputBuffer();              if (store.manageProductPrice(currentUser, name, price))                  cout << "价格修改成功。\n";              else                  cout << "价格修改失败。\n";              break;          }          case 5:          { // 修改库存              string name;              int qty;              cout << "输入要修改库存的商品名称: ";              getline(cin >> ws, name);              cout << "输入新库存: ";              if (!(cin >> qty) || qty < 0)              {                  cout << "无效库存\n";                  clearInputBuffer();                  break;              }              clearInputBuffer();              if (store.manageProductQuantity(currentUser, name, qty))                  cout << "库存修改成功。\n";              else                  cout << "库存修改失败。\n";              break;          }          case 6:          { // 修改折扣              string name;              double discount;              cout << "输入要修改折扣的商品名称: ";              getline(cin >> ws, name);              cout << "输入新折扣率 (0 到 100): ";              if (!(cin >> discount) || discount < 0 || discount > 100)              {                  cout << "无效折扣率\n";                  clearInputBuffer();                  break;              }              clearInputBuffer();              if (store.manageProductDiscount(currentUser, name, discount / 100))                  cout << "折扣修改成功。\n";              else                  cout << "折扣修改失败。\n";              break;          }          case 7:          { // 应用分类折扣              string category;              double discount;              cout << "输入要应用折扣的分类 (Book, Clothing, Food): ";              getline(cin >> ws, category);              cout << "输入折扣率 (0 到 100): ";              if (!(cin >> discount) || discount < 0 || discount > 100)              {                  cout << "无效折扣率\n";                  clearInputBuffer();                  break;              }              clearInputBuffer();              if (store.applyCategoryDiscount(currentUser, category, discount / 100))                  cout << "分类折扣应用成功。\n";              else                  cout << "分类折扣应用失败。\n";              break;          }          case 0:              cout << "正在退出商城管理..." << endl;              break;          default:              cout << "无效选项。" << endl;          }      } while (choice != 0);      return 1; // 返回1表示正常退出商城，继续主循环  } |

### imgUI

#### ui\_main.cpp

|  |
| --- |
| #include <iostream>  #include <vector>  #include <string>  #include "user/user.h"  #include "store/store.h"  #include "ui/ui.h"  // 文件路径常量  extern const std::string USER\_FILE = "./user/users.txt";  extern const std::string STORE\_FILE = "./store";  int main()  {      std::vector<User \*> users;      User \*currentUser = nullptr;      // 创建 Store 对象      Store store(STORE\_FILE);      // 加载用户数据      users = User::loadUsersFromFile(USER\_FILE);      std::cout << "已加载 " << users.size() << " 个用户数据。" << std::endl;      // 加载商店数据      store.loadAllProducts();      // 创建并初始化UI      UI ui(800, 600);      if (!ui.init())      {          std::cerr << "UI初始化失败，程序退出" << std::endl;          return 1;      }      // 启动UI主循环      ui.mainLoop(users, store);      // 程序结束，释放资源      for (User \*user : users)      {          delete user;      }      users.clear();      return 0;  } |

#### User.h

|  |
| --- |
| #ifndef USER\_H  #define USER\_H  #include <iostream>  #include <string>  #include <vector>  #include <fstream>  #include <limits>  class User  {  protected: // protected成员可被派生类访问      std::string username;      std::string password;      std::string userType; // 用户类型      double balance;      static void clearInputBuffer() // 清空输入缓冲区      {          std::cin.clear();                                                   // 清除错误标志          std::cin.ignore(std::numeric\_limits<std::streamsize>::max(), '\n'); // 忽略剩余输入      }  public:      // 构造函数      User(std::string username, std::string password, double balance = 0.0,           bool customer = false, bool seller = false, bool admin = false);      User();      virtual ~User() = default; // 虚析构函数      // 纯虚函数，使User成为抽象类      virtual std::string getUserType() const = 0;      // 账户操作函数      void displayInfo(int index = -1) const;      bool changePassword(const std::string &oldPassword, const std::string &newPassword);      bool deposit(double amount);      bool withdraw(double amount);      double checkBalance() const;      //      static User \*registerUser(std::vector<User \*> &users);    // 注册新用户      static User \*userLogin(const std::vector<User \*> &users); // 用户登录 (获取输入并验证)      // 文件操作函数      static bool saveUsersToFile(const std::vector<User \*> &users, const std::string &filename);      static std::vector<User \*> loadUsersFromFile(const std::string &filename);      static User \*findUser(const std::vector<User \*> &users, const std::string &username);      static void displayAllUsers(const std::vector<User \*> &users);      static bool isUsernameExists(const std::vector<User \*> &users, const std::string &username);      // Getter 函数      std::string getUsername() const;      std::string getPassword() const;      // Setter 函数      void setUsername(const std::string &newUsername);      void setPassword(const std::string &newPassword);      void setBalance(double newBalance);      // 用户验证      static User \*login(const std::vector<User \*> &users, const std::string &username, const std::string &password);  };  // 消费者类  class Customer : public User  {  public:      Customer(std::string username, std::string password, double balance = 0.0);      Customer();      std::string getUserType() const override { return "customer"; }  };  // 商家类  class Seller : public User  {  public:      Seller(std::string username, std::string password, double balance = 0.0);      Seller();      std::string getUserType() const override { return "seller"; }  };  // 管理员类  class Admin : public User  {  public:      Admin(std::string username, std::string password, double balance = 0.0);      Admin();      std::string getUserType() const override { return "admin"; }  };  #endif // USER\_H |

#### User.cpp

|  |
| --- |
| #include "user.h"  #include <iostream>  #include <limits>  #include <string>  #include <fstream>  #include <sstream>  using namespace std;  // User 构造函数实现  User::User(string username, string password, double balance, bool customer, bool seller, bool admin)      : username(username), password(password), balance(balance) {}  User::User()  {      username = "defaultUser";      password = "12345678";      balance = 0.0;      userType = "customer"; // 默认用户类型为消费者  }  // Customer 构造函数实现  Customer::Customer(string username, string password, double balance)      : User(username, password, balance, true, false, false) {}  Customer::Customer() : User()  {      userType = "customer"; // 默认用户类型为消费者  }  // Seller 构造函数实现  Seller::Seller(string username, string password, double balance)      : User(username, password, balance, false, true, false) {}  Seller::Seller() : User()  {      userType = "seller"; // 默认用户类型为商家  }  // Admin 构造函数实现  Admin::Admin(string username, string password, double balance)      : User(username, password, balance, false, false, true) {}  Admin::Admin() : User()  {      userType = "admin"; // 默认用户类型为管理员  }  // --- 静态方法：用户注册实现 ---  User \*User::registerUser(vector<User \*> &users)  {      string username, password;      int userTypeChoice;      cout << "\n--- 用户注册 ---" << endl;      // 获取并验证用户名      while (true)      {          cout << "请输入用户名: ";          getline(cin >> ws, username); // ws 忽略前导空格          if (username.empty())          {              cout << "用户名不能为空，请重新输入！" << endl;              continue;          }          // 调用 User 类的静态方法检查用户名是否存在          if (User::isUsernameExists(users, username))          {              cout << "该用户名已存在，请选择其他用户名！" << endl;              continue;          }          break;      }      // 获取密码      cout << "请输入密码: ";      getline(cin >> ws, password);      // 选择用户类型      cout << "请选择用户类型 (1: 消费者, 2: 商家): ";      while (!(cin >> userTypeChoice) || (userTypeChoice != 1 && userTypeChoice != 2))      {          cout << "无效选择，请输入 1 或 2: ";          User::clearInputBuffer(); // 调用 User 类的静态 clearInputBuffer      }      User::clearInputBuffer(); // 清除 cin 留下的换行符      User \*newUser = nullptr;      // 根据选择创建 Customer 或 Seller 对象 (User 的派生类)      if (userTypeChoice == 1)      {          newUser = new Customer(username, password);      }      else // userTypeChoice == 2      {          newUser = new Seller(username, password);      }      users.push\_back(newUser); // 将新用户指针添加到向量      // 注意：这里不再自动保存文件，让调用者决定何时保存      // User::saveUsersToFile(users, USER\_FILE); // 移除或注释掉      cout << "注册成功！" << endl;      return newUser; // 返回新注册的用户指针  }  // --- 静态方法：用户登录实现 ---  User \*User::userLogin(const vector<User \*> &users)  {      string username, password;      cout << "\n--- 用户登录 ---" << endl;      cout << "请输入用户名: ";      getline(cin >> ws, username);      cout << "请输入密码: ";      getline(cin >> ws, password);      User \*user = findUser(users, username);      if (user == nullptr)      {          cout << "用户不存在！" << endl;          return nullptr;      }      if (user->getPassword() != password)      {          cout << "密码错误！" << endl;          return nullptr;      }      cout << "登录成功！" << endl;      cout << "欢迎回来，" << user->getUsername() << "！" << endl;      return user; // 返回登录成功的用户指针  }  // 显示单个用户信息  void User::displayInfo(int index) const  {      if (index > 0)      {          cout << "用户 " << index << ":" << endl;      }      cout << "  用户名: " << username << endl;      cout << "  密码: " << password << endl;      cout << "  用户类型: " << getUserType() << endl;      cout << "  账户余额: " << balance << endl;      cout << "---------------------" << endl;      // 需改为按下enter也能返回      cout << "按任意键返回" << endl;      cin.get();                                           // 等待用户输入      cin.ignore(numeric\_limits<streamsize>::max(), '\n'); // 忽略剩余输入  }  // 静态方法：显示所有用户信息  void User::displayAllUsers(const vector<User \*> &users)  {      cout << "\n--- 所有用户信息 ---" << endl;      if (users.empty())      {          cout << "当前没有已注册的用户。" << endl;      }      else      {          for (size\_t i = 0; i < users.size(); ++i)          {              users[i]->displayInfo(i + 1);          }      }  }  // 修改密码  bool User::changePassword(const string &oldPassword, const string &newPassword)  {      if (password != oldPassword)      {          cout << "原密码错误！" << endl;          return false;      }      password = newPassword;      cout << "密码修改成功！" << endl;      return true;  }  // 充值余额  bool User::deposit(double amount)  {      if (amount <= 0)      {          cout << "充值金额必须大于0！" << endl;          return false;      }      balance += amount;      // cout << "充值成功！当前余额: " << balance << endl;      return true;  }  // 消费余额  bool User::withdraw(double amount)  {      if (amount <= 0)      {          cout << "消费金额必须大于0！" << endl;          return false;      }      if (balance < amount)      {          cout << "余额不足！当前余额: " << balance << endl;          return false;      }      balance -= amount;      cout << "支付成功！当前余额: " << balance << endl;      return true;  }  // 查询余额  double User::checkBalance() const  {      return balance;  }  // 将用户信息保存到文件  bool User::saveUsersToFile(const vector<User \*> &users, const string &filename)  {      ofstream file(filename);      if (!file.is\_open())      {          cerr << "无法打开文件: " << filename << endl;          return false;      }      for (const auto &user : users)      {          file << user->getUserType() << ","               << user->getUsername() << ","               << user->getPassword() << ","               << user->checkBalance() << endl;      }      file.close();      return true;  }  // 从文件加载用户信息  vector<User \*> User::loadUsersFromFile(const string &filename)  {      vector<User \*> users;      ifstream file(filename);      if (!file.is\_open())      {          cerr << "无法打开文件: " << filename << " (可能是首次运行)" << endl;          return users;      }      string line;      while (getline(file, line))      {          stringstream ss(line);          string userType, username, password;          double balance;          getline(ss, userType, ',');          getline(ss, username, ',');          getline(ss, password, ',');          ss >> balance;          User \*user = nullptr;          if (userType == "customer")          {              user = new Customer(username, password, balance);          }          else if (userType == "seller")          {              user = new Seller(username, password, balance);          }          else if (userType == "admin")          {              user = new Admin(username, password, balance);          }          if (user != nullptr)          {              users.push\_back(user);          }      }      file.close();      return users;  }  // 查找指定用户名的用户  User \*User::findUser(const vector<User \*> &users, const string &username)  {      for (auto user : users)      {          if (user->getUsername() == username)          {              return user;          }      }      return nullptr;  }  // 检查用户名是否已存在  bool User::isUsernameExists(const vector<User \*> &users, const string &username)  {      return findUser(users, username) != nullptr;  }  // 用户登录验证  User \*User::login(const vector<User \*> &users, const string &username, const string &password)  {      User \*user = findUser(users, username);      if (user && user->getPassword() == password)      {          return user;      }      return nullptr;  }  // Getter 函数实现  string User::getUsername() const { return username; }  string User::getPassword() const { return password; }  // Setter 函数实现  void User::setUsername(const string &newUsername) { username = newUsername; }  void User::setPassword(const string &newPassword) { password = newPassword; }  void User::setBalance(double newBalance) { balance = newBalance; } |

#### Ui.h

|  |
| --- |
| #ifndef UI\_H  #define UI\_H  #include <string>  #include <vector>  #include <functional>  // 前向声明  class User;  class Store;  class Product;  // UI系统主类  class UI  {  private:      // 窗口尺寸      int width = 800;      int height = 600;      // UI状态      bool shouldClose = false;      int currentPage = 0; // 0=主菜单, 1=用户菜单, 2=商城      int userType = 0;    // 0=游客, 1=消费者, 2=商家, 3=管理员      // 商品滚动位置      float scrollY = 0.0f;      // 登录输入缓冲区      char usernameBuffer[128] = {0};      char passwordBuffer[128] = {0};      // 用户信息和商店引用      User \*currentUser = nullptr;      std::vector<User \*> \*users = nullptr;      Store \*store = nullptr;      // 商品搜索      char searchBuffer[128] = {0};      std::vector<Product \*> searchResults;      // 购买商品      char buyProductName[128] = {0};      int buyQuantity = 1;      // 商品管理      char newProductName[128] = {0};      char newProductDesc[256] = {0};      float newProductPrice = 0.0f;      int newProductQuantity = 0;      char newProductAttr1[128] = {0}; // 作者/尺寸/过期日期      char newProductAttr2[128] = {0}; // ISBN/颜色      int newProductType = 0;          // 0=图书, 1=服装, 2=食品      // 充值金额      float depositAmount = 0.0f;      // UI辅助函数      void renderMainMenu();      void renderUserMenu();      void renderStore();      void renderUserRegister();      void renderUserLogin();      void renderStoreCustomer();      void renderStoreSeller();      void renderProductList(const std::vector<Product \*> &products, const std::string &title);  public:      UI(int width = 800, int height = 600);      ~UI();      // 初始化UI系统      bool init();      // 主循环      void mainLoop(std::vector<User \*> &users, Store &store);      // 是否应该关闭      bool isClosing() const { return shouldClose; }      // 设置当前用户      void setCurrentUser(User \*user);      // 获取菜单返回值 (兼容原代码)      int getMenuResult();  };  #endif // UI\_H |

#### Ui.cpp

|  |
| --- |
| #include "ui.h"  #include "../user/user.h"  #include "../store/store.h"  #include "../imgui/imgui.h"  #include "../imgui/backends/imgui\_impl\_glfw.h"  #include "../imgui/backends/imgui\_impl\_opengl3.h"  #include <GLFW/glfw3.h>  #include <iostream>  #include <algorithm>  #include <string>  // 全局GLFW窗口指针  GLFWwindow \*window = nullptr;  // UI构造函数  UI::UI(int width, int height)      : width(width), height(height) {}  // UI析构函数  UI::~UI()  {      // 清理ImGui      ImGui\_ImplOpenGL3\_Shutdown();      ImGui\_ImplGlfw\_Shutdown();      ImGui::DestroyContext();      // 清理GLFW      if (window)      {          glfwDestroyWindow(window);          window = nullptr;      }      glfwTerminate();  }  // 初始化UI  bool UI::init()  {      // 初始化GLFW      if (!glfwInit())      {          std::cerr << "GLFW初始化失败" << std::endl;          return false;      }      // 创建窗口      window = glfwCreateWindow(width, height, "电子商城系统", nullptr, nullptr);      if (!window)      {          std::cerr << "创建窗口失败" << std::endl;          glfwTerminate();          return false;      }      // 设置当前上下文      glfwMakeContextCurrent(window);      glfwSwapInterval(1); // 启用垂直同步      // 初始化ImGui      IMGUI\_CHECKVERSION();      ImGui::CreateContext();      ImGuiIO &io = ImGui::GetIO();      io.ConfigFlags |= ImGuiConfigFlags\_NavEnableKeyboard; // 启用键盘控制      io.Fonts->AddFontFromFileTTF("c:\\Windows\\Fonts\\simhei.ttf", 16.0f, NULL, io.Fonts->GetGlyphRangesChineseFull());      // 设置ImGui风格      ImGui::StyleColorsLight();      // 平台/渲染器后端      ImGui\_ImplGlfw\_InitForOpenGL(window, true);      ImGui\_ImplOpenGL3\_Init("#version 130");      return true;  }  // 设置当前用户  void UI::setCurrentUser(User \*user)  {      currentUser = user;      if (user)      {          // 设置用户类型          std::string type = user->getUserType();          if (type == "customer")              userType = 1;          else if (type == "seller")              userType = 2;          else if (type == "admin")              userType = 3;          else              userType = 0;      }      else      {          userType = 0;      }  }  // 主循环  void UI::mainLoop(std::vector<User \*> &usersRef, Store &storeRef)  {      users = &usersRef;      store = &storeRef;      // 主循环      while (!glfwWindowShouldClose(window) && !shouldClose)      {          glfwPollEvents();          // 开始新帧          ImGui\_ImplOpenGL3\_NewFrame();          ImGui\_ImplGlfw\_NewFrame();          ImGui::NewFrame();          // 全屏窗口          ImGui::SetNextWindowPos(ImVec2(0, 0));          ImGui::SetNextWindowSize(ImVec2(width, height));          ImGui::Begin("商城系统", nullptr,                       ImGuiWindowFlags\_NoTitleBar |                           ImGuiWindowFlags\_NoResize |                           ImGuiWindowFlags\_NoMove |                           ImGuiWindowFlags\_NoScrollbar);          // 根据当前页面渲染不同的UI          switch (currentPage)          {          case 0: // 主菜单              renderMainMenu();              break;          case 1: // 用户菜单              renderUserMenu();              break;          case 2: // 商城              renderStore();              break;          case 3: // 注册页面              renderUserRegister();              break;          case 4: // 登录页面              renderUserLogin();              break;          }          ImGui::End();          // 渲染          ImGui::Render();          int displayW, displayH;          glfwGetFramebufferSize(window, &displayW, &displayH);          glViewport(0, 0, displayW, displayH);          glClearColor(0.45f, 0.55f, 0.60f, 1.00f);          glClear(GL\_COLOR\_BUFFER\_BIT);          ImGui\_ImplOpenGL3\_RenderDrawData(ImGui::GetDrawData());          glfwSwapBuffers(window);      }  }  // 渲染主菜单  void UI::renderMainMenu()  {      ImGui::SetWindowFontScale(1.5f);      ImGui::Text("电子商城系统");      ImGui::SetWindowFontScale(1.0f);      ImGui::Separator();      ImGui::Spacing();      ImGui::Spacing();      ImGui::SetCursorPosX((width - 200) / 2);      if (ImGui::Button("注册新用户", ImVec2(200, 50)))      {          currentPage = 3; // 切换到注册页面      }      ImGui::Spacing();      ImGui::SetCursorPosX((width - 200) / 2);      if (ImGui::Button("用户登录", ImVec2(200, 50)))      {          currentPage = 4; // 切换到登录页面          memset(usernameBuffer, 0, sizeof(usernameBuffer));          memset(passwordBuffer, 0, sizeof(passwordBuffer));      }      ImGui::Spacing();      ImGui::SetCursorPosX((width - 200) / 2);      if (ImGui::Button("进入商城 (游客)", ImVec2(200, 50)))      {          currentPage = 2; // 切换到商城      }      ImGui::Spacing();      ImGui::SetCursorPosX((width - 200) / 2);      if (ImGui::Button("退出程序", ImVec2(200, 50)))      {          shouldClose = true;      }  }  // 渲染用户菜单  void UI::renderUserMenu()  {      if (!currentUser)      {          currentPage = 0; // 回到主菜单          return;      }      std::string userTypeStr;      if (userType == 1)          userTypeStr = "消费者";      else if (userType == 2)          userTypeStr = "商家";      else if (userType == 3)          userTypeStr = "管理员";      else          userTypeStr = "未知";      ImGui::SetWindowFontScale(1.5f);      ImGui::Text("%s菜单 - %s", userTypeStr.c\_str(), currentUser->getUsername().c\_str());      ImGui::SetWindowFontScale(1.0f);      ImGui::Separator();      // 用户信息      ImGui::BeginChild("UserInfo", ImVec2(width \* 0.4f, height - 150), true);      ImGui::Text("用户名: %s", currentUser->getUsername().c\_str());      ImGui::Text("用户类型: %s", userTypeStr.c\_str());      ImGui::Text("账户余额: CNY%.2f", currentUser->checkBalance());      ImGui::EndChild();      // 操作区域      ImGui::SameLine();      ImGui::BeginChild("UserActions", ImVec2(0, height - 150), true);      // 修改密码      static char oldPass[128] = {0};      static char newPass[128] = {0};      ImGui::SetNextItemWidth(200);      ImGui::InputText("原密码", oldPass, IM\_ARRAYSIZE(oldPass), ImGuiInputTextFlags\_Password);      ImGui::SetNextItemWidth(200);      ImGui::InputText("新密码", newPass, IM\_ARRAYSIZE(newPass), ImGuiInputTextFlags\_Password);      if (ImGui::Button("修改密码", ImVec2(100, 30)))      {          if (currentUser->changePassword(oldPass, newPass))          {              User::saveUsersToFile(\*users, "user/users.txt");              ImGui::OpenPopup("密码修改成功");              memset(oldPass, 0, sizeof(oldPass));              memset(newPass, 0, sizeof(newPass));          }          else          {              ImGui::OpenPopup("密码修改失败");          }      }      // 弹窗      if (ImGui::BeginPopupModal("密码修改成功", NULL, ImGuiWindowFlags\_AlwaysAutoResize))      {          ImGui::Text("密码修改成功！");          if (ImGui::Button("确定", ImVec2(120, 0)))          {              ImGui::CloseCurrentPopup();          }          ImGui::EndPopup();      }      if (ImGui::BeginPopupModal("密码修改失败", NULL, ImGuiWindowFlags\_AlwaysAutoResize))      {          ImGui::Text("密码修改失败，请检查原密码是否正确。");          if (ImGui::Button("确定", ImVec2(120, 0)))          {              ImGui::CloseCurrentPopup();          }          ImGui::EndPopup();      }      ImGui::Separator();      // 消费者特有 - 充值      if (userType == 1)      {          ImGui::SetNextItemWidth(200);          ImGui::InputFloat("充值金额", &depositAmount, 10.0f, 100.0f, "%.2f");          if (ImGui::Button("充值", ImVec2(100, 30)))          {              if (depositAmount > 0)              {                  if (currentUser->deposit(depositAmount))                  {                      User::saveUsersToFile(\*users, "user/users.txt");                      ImGui::OpenPopup("充值成功");                      depositAmount = 0.0f;                  }              }              else              {                  ImGui::OpenPopup("充值失败");              }          }          if (ImGui::BeginPopupModal("充值成功", NULL, ImGuiWindowFlags\_AlwaysAutoResize))          {              ImGui::Text("充值成功！当前余额: CNY%.2f", currentUser->checkBalance());              if (ImGui::Button("确定", ImVec2(120, 0)))              {                  ImGui::CloseCurrentPopup();              }              ImGui::EndPopup();          }          if (ImGui::BeginPopupModal("充值失败", NULL, ImGuiWindowFlags\_AlwaysAutoResize))          {              ImGui::Text("充值失败，请确保充值金额大于0。");              if (ImGui::Button("确定", ImVec2(120, 0)))              {                  ImGui::CloseCurrentPopup();              }              ImGui::EndPopup();          }      }      ImGui::EndChild();      // 底部按钮      ImGui::Separator();      if (ImGui::Button("进入商城", ImVec2(120, 40)))      {          currentPage = 2; // 切换到商城          searchResults.clear();          memset(searchBuffer, 0, sizeof(searchBuffer));      }      ImGui::SameLine();      if (ImGui::Button("退出登录", ImVec2(120, 40)))      {          currentUser = nullptr;          userType = 0;          currentPage = 0; // 回到主菜单      }  }  // 渲染商城  void UI::renderStore()  {      ImGui::SetWindowFontScale(1.5f);      if (currentUser)      {          if (userType == 2)          {              ImGui::Text("商城管理 - %s", currentUser->getUsername().c\_str());          }          else          {              ImGui::Text("商城 - %s", currentUser->getUsername().c\_str());          }      }      else      {          ImGui::Text("商城 - 游客模式");      }      ImGui::SetWindowFontScale(1.0f);      ImGui::Separator();      // 根据用户类型显示不同的商城界面      if (userType == 2)      { // 商家          renderStoreSeller();      }      else      { // 消费者或游客          renderStoreCustomer();      }      // 底部按钮      ImGui::Separator();      if (ImGui::Button("返回", ImVec2(120, 40)))      {          if (currentUser)          {              currentPage = 1; // 返回用户菜单          }          else          {              currentPage = 0; // 返回主菜单          }      }  }  // 渲染注册页面  void UI::renderUserRegister()  {      ImGui::SetWindowFontScale(1.5f);      ImGui::Text("用户注册");      ImGui::SetWindowFontScale(1.0f);      ImGui::Separator();      static char regUsername[128] = {0};      static char regPassword[128] = {0};      static int regUserType = 0; // 0=消费者, 1=商家      ImGui::SetNextItemWidth(250);      ImGui::InputText("用户名", regUsername, IM\_ARRAYSIZE(regUsername));      ImGui::SetNextItemWidth(250);      ImGui::InputText("密码", regPassword, IM\_ARRAYSIZE(regPassword), ImGuiInputTextFlags\_Password);      ImGui::RadioButton("消费者", &regUserType, 0);      ImGui::SameLine();      ImGui::RadioButton("商家", &regUserType, 1);      if (ImGui::Button("注册", ImVec2(120, 40)))      {          if (strlen(regUsername) == 0)          {              ImGui::OpenPopup("注册失败");          }          else if (User::isUsernameExists(\*users, regUsername))          {              ImGui::OpenPopup("用户名已存在");          }          else          {              User \*newUser = nullptr;              if (regUserType == 0)              {                  newUser = new Customer(regUsername, regPassword);              }              else              {                  newUser = new Seller(regUsername, regPassword);              }              users->push\_back(newUser);              User::saveUsersToFile(\*users, "user/users.txt");              currentUser = newUser;              setCurrentUser(newUser);              currentPage = 1; // 跳转至用户菜单              // 清空输入              memset(regUsername, 0, sizeof(regUsername));              memset(regPassword, 0, sizeof(regPassword));          }      }      ImGui::SameLine();      if (ImGui::Button("返回", ImVec2(120, 40)))      {          currentPage = 0; // 回到主菜单      }      // 弹窗      if (ImGui::BeginPopupModal("注册失败", NULL, ImGuiWindowFlags\_AlwaysAutoResize))      {          ImGui::Text("注册失败，用户名不能为空！");          if (ImGui::Button("确定", ImVec2(120, 0)))          {              ImGui::CloseCurrentPopup();          }          ImGui::EndPopup();      }      if (ImGui::BeginPopupModal("用户名已存在", NULL, ImGuiWindowFlags\_AlwaysAutoResize))      {          ImGui::Text("该用户名已存在，请选择其他用户名！");          if (ImGui::Button("确定", ImVec2(120, 0)))          {              ImGui::CloseCurrentPopup();          }          ImGui::EndPopup();      }  }  // 渲染登录页面  void UI::renderUserLogin()  {      ImGui::SetWindowFontScale(1.5f);      ImGui::Text("用户登录");      ImGui::SetWindowFontScale(1.0f);      ImGui::Separator();      ImGui::SetNextItemWidth(250);      ImGui::InputText("用户名", usernameBuffer, IM\_ARRAYSIZE(usernameBuffer));      ImGui::SetNextItemWidth(250);      ImGui::InputText("密码", passwordBuffer, IM\_ARRAYSIZE(passwordBuffer), ImGuiInputTextFlags\_Password);      if (ImGui::Button("登录", ImVec2(120, 40)))      {          User \*user = User::login(\*users, usernameBuffer, passwordBuffer);          if (user)          {              currentUser = user;              setCurrentUser(user);              currentPage = 1; // 跳转至用户菜单          }          else          {              ImGui::OpenPopup("登录失败");          }      }      ImGui::SameLine();      if (ImGui::Button("返回", ImVec2(120, 40)))      {          currentPage = 0; // 回到主菜单      }      // 弹窗      if (ImGui::BeginPopupModal("登录失败", NULL, ImGuiWindowFlags\_AlwaysAutoResize))      {          ImGui::Text("登录失败，用户名或密码错误！");          if (ImGui::Button("确定", ImVec2(120, 0)))          {              ImGui::CloseCurrentPopup();          }          ImGui::EndPopup();      }  }  // 渲染消费者商城  void UI::renderStoreCustomer()  {      ImGui::BeginChild("StoreControls", ImVec2(width, 50));      ImGui::SetNextItemWidth(300);      if (ImGui::InputText("搜索商品", searchBuffer, IM\_ARRAYSIZE(searchBuffer), ImGuiInputTextFlags\_EnterReturnsTrue))      {          searchResults = store->searchProductsByName(searchBuffer);      }      ImGui::SameLine();      if (ImGui::Button("搜索", ImVec2(80, 0)))      {          searchResults = store->searchProductsByName(searchBuffer);      }      ImGui::SameLine();      if (ImGui::Button("显示所有商品", ImVec2(150, 0)))      {          searchResults.clear();          memset(searchBuffer, 0, sizeof(searchBuffer));      }      ImGui::EndChild();      ImGui::Separator();      // 商品列表      if (!searchResults.empty())      {          renderProductList(searchResults, "搜索结果");      }      else      {          renderProductList(store->getProducts(), "所有商品");      }      // 购买商品区域      if (currentUser && userType == 1)      {          ImGui::Separator();          ImGui::Text("购买商品");          ImGui::SetNextItemWidth(250);          ImGui::InputText("商品名称", buyProductName, IM\_ARRAYSIZE(buyProductName));          ImGui::SetNextItemWidth(100);          ImGui::InputInt("购买数量", &buyQuantity);          if (buyQuantity < 1)              buyQuantity = 1;          if (ImGui::Button("购买", ImVec2(100, 30)))          {              if (strlen(buyProductName) == 0)              {                  ImGui::OpenPopup("购买失败");              }              else              {                  Product \*product = store->findProductByName(buyProductName);                  if (!product)                  {                      ImGui::OpenPopup("商品不存在");                  }                  else if (product->getQuantity() < buyQuantity)                  {                      ImGui::OpenPopup("库存不足");                  }                  else                  {                      double totalCost = product->getPrice() \* buyQuantity;                      if (currentUser->checkBalance() < totalCost)                      {                          ImGui::OpenPopup("余额不足");                      }                      else                      {                          // 扣除用户余额                          currentUser->withdraw(totalCost);                          // 添加给商家余额                          std::string sellerName = product->getSellerUsername();                          if (!sellerName.empty())                          {                              User \*seller = User::findUser(\*users, sellerName);                              if (seller)                              {                                  seller->deposit(totalCost);                              }                          }                          // 减少库存                          product->setQuantity(product->getQuantity() - buyQuantity);                          // 保存更改                          User::saveUsersToFile(\*users, "user/users.txt");                          store->saveAllProducts();                          ImGui::OpenPopup("购买成功");                          memset(buyProductName, 0, sizeof(buyProductName));                          buyQuantity = 1;                      }                  }              }          }          // 弹窗          if (ImGui::BeginPopupModal("购买失败", NULL, ImGuiWindowFlags\_AlwaysAutoResize))          {              ImGui::Text("请输入要购买的商品名称！");              if (ImGui::Button("确定", ImVec2(120, 0)))              {                  ImGui::CloseCurrentPopup();              }              ImGui::EndPopup();          }          if (ImGui::BeginPopupModal("商品不存在", NULL, ImGuiWindowFlags\_AlwaysAutoResize))          {              ImGui::Text("找不到商品 \"%s\"！", buyProductName);              if (ImGui::Button("确定", ImVec2(120, 0)))              {                  ImGui::CloseCurrentPopup();              }              ImGui::EndPopup();          }          if (ImGui::BeginPopupModal("库存不足", NULL, ImGuiWindowFlags\_AlwaysAutoResize))          {              Product \*product = store->findProductByName(buyProductName);              if (product)              {                  ImGui::Text("库存不足！当前库存: %d", product->getQuantity());              }              if (ImGui::Button("确定", ImVec2(120, 0)))              {                  ImGui::CloseCurrentPopup();              }              ImGui::EndPopup();          }          if (ImGui::BeginPopupModal("余额不足", NULL, ImGuiWindowFlags\_AlwaysAutoResize))          {              Product \*product = store->findProductByName(buyProductName);              if (product)              {                  double totalCost = product->getPrice() \* buyQuantity;                  ImGui::Text("余额不足！需要: CNY%.2f, 当前余额: CNY%.2f",                              totalCost, currentUser->checkBalance());              }              if (ImGui::Button("确定", ImVec2(120, 0)))              {                  ImGui::CloseCurrentPopup();              }              ImGui::EndPopup();          }          if (ImGui::BeginPopupModal("购买成功", NULL, ImGuiWindowFlags\_AlwaysAutoResize))          {              ImGui::Text("购买成功！当前余额: CNY%.2f", currentUser->checkBalance());              if (ImGui::Button("确定", ImVec2(120, 0)))              {                  ImGui::CloseCurrentPopup();              }              ImGui::EndPopup();          }      }  }  // 渲染商家商城  void UI::renderStoreSeller()  {      if (!currentUser || userType != 2)          return;      std::string sellerName = currentUser->getUsername();      // 显示选项卡      static int currentTab = 0;      ImGui::BeginTabBar("SellerTabs");      if (ImGui::BeginTabItem("我的商品"))      {          ImGui::BeginChild("MyProducts", ImVec2(0, height - 200));          renderProductList(store->getSellerProducts(sellerName), "我的商品");          ImGui::EndChild();          ImGui::EndTabItem();      }      if (ImGui::BeginTabItem("添加商品"))      {          ImGui::BeginChild("AddProduct", ImVec2(0, height - 200));          ImGui::SetNextItemWidth(250);          ImGui::InputText("商品名称", newProductName, IM\_ARRAYSIZE(newProductName));          ImGui::SetNextItemWidth(350);          ImGui::InputTextMultiline("商品描述", newProductDesc, IM\_ARRAYSIZE(newProductDesc));          ImGui::SetNextItemWidth(150);          ImGui::InputFloat("商品价格", &newProductPrice, 1.0f, 10.0f, "CNY%.2f");          if (newProductPrice < 0)              newProductPrice = 0;          ImGui::SetNextItemWidth(150);          ImGui::InputInt("库存数量", &newProductQuantity);          if (newProductQuantity < 0)              newProductQuantity = 0;          const char \*productTypes[] = {"图书", "服装", "食品", "其他"};          ImGui::SetNextItemWidth(150);          ImGui::Combo("商品类型", &newProductType, productTypes, IM\_ARRAYSIZE(productTypes));          // 根据不同的商品类型显示不同的输入字段          if (newProductType == 0)          { // 图书              ImGui::SetNextItemWidth(250);              ImGui::InputText("作者", newProductAttr1, IM\_ARRAYSIZE(newProductAttr1));              ImGui::SetNextItemWidth(250);              ImGui::InputText("ISBN", newProductAttr2, IM\_ARRAYSIZE(newProductAttr2));          }          else if (newProductType == 1)          { // 服装              ImGui::SetNextItemWidth(150);              ImGui::InputText("尺寸", newProductAttr1, IM\_ARRAYSIZE(newProductAttr1));              ImGui::SetNextItemWidth(150);              ImGui::InputText("颜色", newProductAttr2, IM\_ARRAYSIZE(newProductAttr2));          }          else if (newProductType == 2)          { // 食品              ImGui::SetNextItemWidth(150);              ImGui::InputText("过期日期", newProductAttr1, IM\_ARRAYSIZE(newProductAttr1));          }          else          {              ImGui::SetNextItemWidth(250);              ImGui::InputText("分类名称", newProductAttr1, IM\_ARRAYSIZE(newProductAttr1));          }          if (ImGui::Button("添加商品", ImVec2(120, 30)))          {              if (strlen(newProductName) == 0)              {                  ImGui::OpenPopup("商品添加失败");              }              else              {                  bool success = false;                  if (newProductType == 0)                  { // 图书                      success = store->createBook(                          currentUser, newProductName, newProductDesc,                          newProductPrice, newProductQuantity,                          newProductAttr1, newProductAttr2);                  }                  else if (newProductType == 1)                  { // 服装                      success = store->createClothing(                          currentUser, newProductName, newProductDesc,                          newProductPrice, newProductQuantity,                          newProductAttr1, newProductAttr2);                  }                  else if (newProductType == 2)                  { // 食品                      success = store->createFood(                          currentUser, newProductName, newProductDesc,                          newProductPrice, newProductQuantity,                          newProductAttr1);                  }                  else                  { // 其他                      success = store->createGeneral(                          currentUser, newProductName, newProductDesc,                          newProductPrice, newProductQuantity,                          newProductAttr1);                  }                  if (success)                  {                      ImGui::OpenPopup("商品添加成功");                      memset(newProductName, 0, sizeof(newProductName));                      memset(newProductDesc, 0, sizeof(newProductDesc));                      memset(newProductAttr1, 0, sizeof(newProductAttr1));                      memset(newProductAttr2, 0, sizeof(newProductAttr2));                      newProductPrice = 0.0f;                      newProductQuantity = 0;                  }                  else                  {                      ImGui::OpenPopup("商品添加失败");                  }              }          }          if (ImGui::BeginPopupModal("商品添加成功", NULL, ImGuiWindowFlags\_AlwaysAutoResize))          {              ImGui::Text("商品添加成功！");              if (ImGui::Button("确定", ImVec2(120, 0)))              {                  ImGui::CloseCurrentPopup();              }              ImGui::EndPopup();          }          ImGui::EndChild();          ImGui::EndTabItem();      }      if (ImGui::BeginTabItem("管理商品"))      {          ImGui::BeginChild("ManageProducts", ImVec2(0, height - 200));          ImGui::Text("修改商品价格");          static char priceProductName[128] = {0};          static float newPrice = 0.0f;          ImGui::SetNextItemWidth(250);          ImGui::InputText("商品名称##price", priceProductName, IM\_ARRAYSIZE(priceProductName));          ImGui::SetNextItemWidth(150);          ImGui::InputFloat("新价格", &newPrice, 1.0f, 10.0f, "CNY%.2f");          if (newPrice < 0)              newPrice = 0;          if (ImGui::Button("修改价格", ImVec2(120, 30)))          {              if (strlen(priceProductName) == 0)              {                  ImGui::OpenPopup("修改失败");              }              else              {                  if (store->manageProductPrice(currentUser, priceProductName, newPrice))                  {                      ImGui::OpenPopup("价格修改成功");                      memset(priceProductName, 0, sizeof(priceProductName));                      newPrice = 0.0f;                  }                  else                  {                      ImGui::OpenPopup("价格修改失败");                  }              }          }          if (ImGui::BeginPopupModal("修改失败", NULL, ImGuiWindowFlags\_AlwaysAutoResize))          {              ImGui::Text("请输入商品名称！");              if (ImGui::Button("确定", ImVec2(120, 0)))              {                  ImGui::CloseCurrentPopup();              }              ImGui::EndPopup();          }          if (ImGui::BeginPopupModal("价格修改成功", NULL, ImGuiWindowFlags\_AlwaysAutoResize))          {              ImGui::Text("商品价格修改成功！");              if (ImGui::Button("确定", ImVec2(120, 0)))              {                  ImGui::CloseCurrentPopup();              }              ImGui::EndPopup();          }          if (ImGui::BeginPopupModal("价格修改失败", NULL, ImGuiWindowFlags\_AlwaysAutoResize))          {              ImGui::Text("价格修改失败！请确认商品名称正确且为您所有。");              if (ImGui::Button("确定", ImVec2(120, 0)))              {                  ImGui::CloseCurrentPopup();              }              ImGui::EndPopup();          }          ImGui::Separator();          ImGui::Text("修改商品库存");          static char qtyProductName[128] = {0};          static int newQty = 0;          ImGui::SetNextItemWidth(250);          ImGui::InputText("商品名称##qty", qtyProductName, IM\_ARRAYSIZE(qtyProductName));          ImGui::SetNextItemWidth(150);          ImGui::InputInt("新库存", &newQty);          if (newQty < 0)              newQty = 0;          if (ImGui::Button("修改库存", ImVec2(120, 30)))          {              if (strlen(qtyProductName) == 0)              {                  ImGui::OpenPopup("修改失败");              }              else              {                  if (store->manageProductQuantity(currentUser, qtyProductName, newQty))                  {                      ImGui::OpenPopup("库存修改成功");                      memset(qtyProductName, 0, sizeof(qtyProductName));                      newQty = 0;                  }                  else                  {                      ImGui::OpenPopup("库存修改失败");                  }              }          }          if (ImGui::BeginPopupModal("库存修改成功", NULL, ImGuiWindowFlags\_AlwaysAutoResize))          {              ImGui::Text("商品库存修改成功！");              if (ImGui::Button("确定", ImVec2(120, 0)))              {                  ImGui::CloseCurrentPopup();              }              ImGui::EndPopup();          }          if (ImGui::BeginPopupModal("库存修改失败", NULL, ImGuiWindowFlags\_AlwaysAutoResize))          {              ImGui::Text("库存修改失败！请确认商品名称正确且为您所有。");              if (ImGui::Button("确定", ImVec2(120, 0)))              {                  ImGui::CloseCurrentPopup();              }              ImGui::EndPopup();          }          ImGui::Separator();          ImGui::Text("修改商品折扣");          static char discountProductName[128] = {0};          static float newDiscount = 0.0f;          ImGui::SetNextItemWidth(250);          ImGui::InputText("商品名称##discount", discountProductName, IM\_ARRAYSIZE(discountProductName));          ImGui::SetNextItemWidth(150);          ImGui::SliderFloat("折扣 (%)", &newDiscount, 0.0f, 100.0f, "%.1f%%");          if (ImGui::Button("应用折扣", ImVec2(120, 30)))          {              if (strlen(discountProductName) == 0)              {                  ImGui::OpenPopup("修改失败");              }              else              {                  if (store->manageProductDiscount(currentUser, discountProductName, newDiscount / 100.0f))                  {                      ImGui::OpenPopup("折扣修改成功");                      memset(discountProductName, 0, sizeof(discountProductName));                      newDiscount = 0.0f;                  }                  else                  {                      ImGui::OpenPopup("折扣修改失败");                  }              }          }          if (ImGui::BeginPopupModal("折扣修改成功", NULL, ImGuiWindowFlags\_AlwaysAutoResize))          {              ImGui::Text("商品折扣修改成功！");              if (ImGui::Button("确定", ImVec2(120, 0)))              {                  ImGui::CloseCurrentPopup();              }              ImGui::EndPopup();          }          if (ImGui::BeginPopupModal("折扣修改失败", NULL, ImGuiWindowFlags\_AlwaysAutoResize))          {              ImGui::Text("折扣修改失败！请确认商品名称正确且为您所有。");              if (ImGui::Button("确定", ImVec2(120, 0)))              {                  ImGui::CloseCurrentPopup();              }              ImGui::EndPopup();          }          ImGui::Separator();          ImGui::Text("修改商品类型折扣");          static char categoryName[128] = {0};          static float categoryDiscount = 0.0f;          static int selectedCategory = 0;          const char \*categories[] = {"Book", "Clothing", "Food", "其他"};          ImGui::SetNextItemWidth(250);          ImGui::Combo("商品类型", &selectedCategory, categories, IM\_ARRAYSIZE(categories));          // 如果选择"其他"类型，则显示输入框          if (selectedCategory == 3)          { // "其他"选项的索引              ImGui::SetNextItemWidth(250);              ImGui::InputText("自定义类型", categoryName, IM\_ARRAYSIZE(categoryName));          }          ImGui::SetNextItemWidth(150);          ImGui::SliderFloat("类别折扣 (%)", &categoryDiscount, 0.0f, 100.0f, "%.1f%%");          if (ImGui::Button("应用类别折扣", ImVec2(140, 30)))          {              std::string category;              if (selectedCategory == 3 && strlen(categoryName) > 0)              {                  // 如果选择"其他"且输入了自定义类型                  category = categoryName;              }              else              {                  category = categories[selectedCategory];              }              if (store->applyCategoryDiscount(currentUser, category, categoryDiscount / 100.0f))              {                  ImGui::OpenPopup("类别折扣修改成功");                  categoryDiscount = 0.0f;                  if (selectedCategory == 3)                  {                      memset(categoryName, 0, sizeof(categoryName));                  }              }              else              {                  ImGui::OpenPopup("类别折扣修改失败");              }          }          if (ImGui::BeginPopupModal("类别折扣修改成功", NULL, ImGuiWindowFlags\_AlwaysAutoResize))          {              ImGui::Text("商品类型折扣修改成功！");              if (ImGui::Button("确定", ImVec2(120, 0)))              {                  ImGui::CloseCurrentPopup();              }              ImGui::EndPopup();          }          if (ImGui::BeginPopupModal("类别折扣修改失败", NULL, ImGuiWindowFlags\_AlwaysAutoResize))          {              ImGui::Text("类别折扣修改失败！请确认您拥有该类别的商品。");              if (ImGui::Button("确定", ImVec2(120, 0)))              {                  ImGui::CloseCurrentPopup();              }              ImGui::EndPopup();          }          ImGui::EndChild();          ImGui::EndTabItem();      }      ImGui::EndTabBar();  }  // 渲染商品列表  void UI::renderProductList(const std::vector<Product \*> &products, const std::string &title)  {      ImGui::Text("%s (%d 件商品)", title.c\_str(), products.size());      if (products.empty())      {          ImGui::TextColored(ImVec4(0.7f, 0.7f, 0.7f, 1.0f), "没有商品");          return;      }      // 商品表格      if (ImGui::BeginTable("ProductTable", 6, ImGuiTableFlags\_Borders | ImGuiTableFlags\_RowBg))      {          ImGui::TableSetupColumn("名称");          ImGui::TableSetupColumn("描述");          ImGui::TableSetupColumn("分类");          ImGui::TableSetupColumn("价格");          ImGui::TableSetupColumn("库存");          ImGui::TableSetupColumn("商家");          ImGui::TableHeadersRow();          for (const auto &product : products)          {              ImGui::TableNextRow();              ImGui::TableSetColumnIndex(0);              ImGui::Text("%s", product->getName().c\_str());              ImGui::TableSetColumnIndex(1);              ImGui::TextWrapped("%s", product->getDescription().c\_str());              ImGui::TableSetColumnIndex(2);              ImGui::Text("%s", product->getType().c\_str());              ImGui::TableSetColumnIndex(3);              if (product->getDiscountRate() > 0)              {                  ImGui::Text("CNY%.2f (%.0f%%折)", product->getPrice(),                              (1.0 - product->getDiscountRate()) \* 100);              }              else              {                  ImGui::Text("CNY%.2f", product->getPrice());              }              ImGui::TableSetColumnIndex(4);              ImGui::Text("%d", product->getQuantity());              ImGui::TableSetColumnIndex(5);              ImGui::Text("%s", product->getSellerUsername().c\_str());          }          ImGui::EndTable();      }  }  // 获取菜单结果 (兼容原代码)  int UI::getMenuResult()  {      if (shouldClose)          return 0; // 退出程序      if (currentPage == 2)          return 3; // 进入商城      if (!currentUser)          return 1; // 继续主循环      return 1;     // 默认继续主循环  } |

#### Store.h

|  |
| --- |
| #ifndef STORE\_H  #define STORE\_H  #include <string>  #include <vector>  #include <iostream>  #include <fstream>  #include <sstream>  #include <iomanip>  #include <limits>  #include <map>  #include <filesystem>  // 前向声明 User  class User;  // --- Product 基类和派生类 ---  class Product  {  protected:      std::string name;      std::string description;      double originalPrice;      int quantity;      double discountRate;      std::string sellerUsername; // 添加商品所属商家  public:      Product(std::string name, std::string desc, double price, int qty, std::string seller = "")          : name(name), description(desc), originalPrice(price), quantity(qty),            discountRate(0.0), sellerUsername(seller) {}      virtual ~Product() = default;      virtual double getPrice() const { return originalPrice \* (1.0 - discountRate); }      virtual void display() const;      virtual std::string getType() const = 0;      virtual void save(std::ofstream &ofs) const;      // Getters and Setters      std::string getName() const { return name; }      std::string getDescription() const { return description; }      double getOriginalPrice() const { return originalPrice; }      int getQuantity() const { return quantity; }      double getDiscountRate() const { return discountRate; }      std::string getSellerUsername() const { return sellerUsername; } // 获取商品所属商家      void setName(const std::string &newName) { name = newName; }      void setDescription(const std::string &newDesc) { description = newDesc; }      void setOriginalPrice(double newPrice)      {          if (newPrice >= 0)              originalPrice = newPrice;      }      void setQuantity(int newQuantity)      {          if (newQuantity >= 0)              quantity = newQuantity;      }      void setDiscountRate(double newRate)      {          if (newRate >= 0.0 && newRate <= 1.0)              discountRate = newRate;      }      void setSellerUsername(const std::string &seller) { sellerUsername = seller; } // 设置商品所属商家  };  class Book : public Product  {  private:      std::string author;      std::string isbn;  public:      Book(std::string name, std::string desc, double price, int qty,           std::string auth, std::string isbn\_num, std::string seller = "")          : Product(name, desc, price, qty, seller), author(auth), isbn(isbn\_num) {}      std::string getType() const override { return "Book"; }      void display() const override;      void save(std::ofstream &ofs) const override;      // 新增的 getters      std::string getAuthor() const { return author; }      std::string getIsbn() const { return isbn; }  };  class Clothing : public Product  {  private:      std::string size;      std::string color;  public:      Clothing(std::string name, std::string desc, double price, int qty,               std::string sz, std::string clr, std::string seller = "")          : Product(name, desc, price, qty, seller), size(sz), color(clr) {}      std::string getType() const override { return "Clothing"; }      void display() const override;      void save(std::ofstream &ofs) const override;      // 新增的 getters      std::string getSize() const { return size; }      std::string getColor() const { return color; }  };  class Food : public Product  {  private:      std::string expirationDate;  public:      Food(std::string name, std::string desc, double price, int qty,           std::string expDate, std::string seller = "")          : Product(name, desc, price, qty, seller), expirationDate(expDate) {}      std::string getType() const override { return "Food"; }      void display() const override;      void save(std::ofstream &ofs) const override;      // 新增的 getters      std::string getExpirationDate() const { return expirationDate; }  };  class General : public Product  {  private:      std::string generalType;  public:      General(std::string name, std::string desc, double price, int qty,              std::string generalType, std::string seller = "")          : Product(name, desc, price, qty, seller), generalType(generalType) {}      std::string getType() const override { return generalType; }      void display() const override;      void save(std::ofstream &ofs) const override;      // 新增的 getters      std::string getGeneralType() const { return generalType; }  };  // --- Store Class ---  class Store  {  private:      std::vector<Product \*> allProducts;                           // 存储所有商家的商品      std::map<std::string, std::vector<Product \*>> sellerProducts; // 每个商家的商品映射      std::string storeDirectory;                                   // 商品文件所在目录      // 辅助方法      std::string getSellerFilename(const std::string &username) const;      bool saveProductsForSeller(const std::string &sellerUsername);      bool ensureDirectoryExists(const std::string &path) const;  public:      // 构造函数接收目录名      Store(const std::string &directory);      ~Store();      Product \*findProductByName(const std::string &name, const std::string &sellerUsername = "");      // 文件操作      bool loadAllProducts();      bool loadSellerProducts(const std::string &sellerUsername);      bool saveAllProducts();      // 显示功能      void displayAllProducts() const;                                     // 显示所有商品      void displaySellerProducts(const std::string &sellerUsername) const; // 显示指定商家的商品      // 搜索功能      std::vector<Product \*> searchProductsByName(const std::string &searchTerm, const std::string &sellerUsername = "") const;      // 商家商品管理功能      bool createBook(User \*currentUser, const std::string &name, const std::string &desc,                      double price, int qty, const std::string &author, const std::string &isbn);      bool createClothing(User \*currentUser, const std::string &name, const std::string &desc,                          double price, int qty, const std::string &size, const std::string &color);      bool createFood(User \*currentUser, const std::string &name, const std::string &desc,                      double price, int qty, const std::string &expDate);      bool createGeneral(User \*currentUser, const std::string &name, const std::string &desc,                         double price, int qty, const std::string &generalType);      bool manageProductPrice(User \*currentUser, const std::string &productName, double newPrice);      bool manageProductQuantity(User \*currentUser, const std::string &productName, int newQuantity);      bool manageProductDiscount(User \*currentUser, const std::string &productName, double newDiscount);      bool applyCategoryDiscount(User \*currentUser, const std::string &category, double discount);      // 获取商品      const std::vector<Product \*> &getProducts() const { return allProducts; }      std::vector<Product \*> getSellerProducts(const std::string &sellerUsername) const;  };  #endif // STORE\_H |

#### Store.cpp

|  |
| --- |
| #include "store.h"  #include "../user/user.h"  #include <iostream>  #include <vector>  #include <string>  #include <fstream>  #include <sstream>  #include <iomanip>  #include <algorithm>  #include <filesystem>  #include <set>  #include <direct.h> // Windows 系统特定的目录操作  namespace fs = std::filesystem;  using namespace std;  // --- Product 方法实现 ---  void Product::display() const  {      std::cout << "  商户名称：" << sellerUsername << std::endl;      std::cout << "  商品名称: " << name << std::endl;      std::cout << "  描述: " << description << std::endl;      if (discountRate > 0)      {          std::cout << "  原价: ¥" << std::fixed << std::setprecision(2) << originalPrice << std::endl;          std::cout << "  折扣: " << (discountRate \* 100) << "%" << std::endl;          std::cout << "  现价: ¥" << std::fixed << std::setprecision(2) << getPrice() << std::endl;      }      else      {          std::cout << "  价格: ¥" << std::fixed << std::setprecision(2) << getPrice() << std::endl;      }      std::cout << "  库存: " << quantity << " 件" << std::endl;      if (!sellerUsername.empty())      {          std::cout << "  商家: " << sellerUsername << std::endl;      }  }  void Product::save(std::ofstream &ofs) const  {      ofs << getType() << "," << name << "," << description << ","          << originalPrice << "," << quantity << "," << discountRate << "," << sellerUsername;  }  void Book::display() const // 复用父类  {      Product::display();      std::cout << "  作者: " << author << std::endl;      std::cout << "  ISBN: " << isbn << std::endl;  }  void Book::save(std::ofstream &ofs) const  {      Product::save(ofs);      ofs << "," << author << "," << isbn;  }  void Clothing::display() const  {      Product::display();      std::cout << "  尺寸: " << size << std::endl;      std::cout << "  颜色: " << color << std::endl;  }  void Clothing::save(std::ofstream &ofs) const  {      Product::save(ofs);      ofs << "," << size << "," << color;  }  void Food::display() const  {      Product::display();      std::cout << "  过期日期: " << expirationDate << std::endl;  }  void Food::save(std::ofstream &ofs) const  {      Product::save(ofs);      ofs << "," << expirationDate;  }  void General::display() const  {      Product::display();      // std::cout << "  类型: " << generalType << std::endl;  }  void General::save(std::ofstream &ofs) const  {      Product::save(ofs);      ofs << "," << generalType;  }  // --- Store 类实现 ---  // 构造函数  Store::Store(const string &directory) : storeDirectory(directory)  {      // 确保商家目录存在      string sellersDir = storeDirectory + "/sellers";      ensureDirectoryExists(sellersDir);      // 加载所有商品数据      if (!loadAllProducts())      {          cerr << "警告: 无法加载商店数据。商店可能为空或文件丢失/损坏。" << endl;      }      else      {          cout << "已从商店加载 " << allProducts.size() << " 件商品。" << endl;      }  }  // 析构函数  Store::~Store()  {      for (Product \*p : allProducts)      {          delete p;      }      allProducts.clear();      sellerProducts.clear(); // 清除映射  }  // 确保目录存在  bool Store::ensureDirectoryExists(const string &path) const  {      try      {          if (!fs::exists(path))          {              fs::create\_directories(path);              cout << "创建目录: " << path << endl;          }          return true;      }      catch (const fs::filesystem\_error &e)      {          cerr << "创建目录失败 " << path << ": " << e.what() << endl;          // 尝试使用传统方法创建目录          if (\_mkdir(path.c\_str()) == 0)          {              cout << "使用传统方法创建目录: " << path << endl;              return true;          }          return false;      }      catch (...)      {          cerr << "创建目录时发生未知错误: " << path << endl;          return false;      }  }  // 获取商家文件名  string Store::getSellerFilename(const string &username) const  {      return storeDirectory + "/sellers/" + username + ".txt";  }  // 加载所有商品  bool Store::loadAllProducts()  {      // 清空现有商品      for (Product \*p : allProducts)      {          delete p;      }      allProducts.clear();      sellerProducts.clear();      // 获取sellers目录下的所有文件      string sellersDir = storeDirectory + "/sellers";      try      {          // 如果目录不存在，创建目录并返回成功（表示商店为空）          if (!fs::exists(sellersDir))          {              ensureDirectoryExists(sellersDir);              return true;          }          // 遍历目录中的所有文件          for (const auto &entry : fs::directory\_iterator(sellersDir))          {              if (entry.is\_regular\_file())              {                  string filename = entry.path().string();                  string sellerUsername = entry.path().stem().string(); // 获取不带扩展名的文件名                  loadSellerProducts(sellerUsername);              }          }          return true;      }      catch (const fs::filesystem\_error &e)      {          cerr << "读取商家目录失败: " << e.what() << endl;          return false;      }      catch (...)      {          cerr << "读取商家数据时发生未知错误" << endl;          return false;      }  }  // 加载指定商家的商品  bool Store::loadSellerProducts(const string &sellerUsername)  {      string filename = getSellerFilename(sellerUsername);      ifstream file(filename);      if (!file.is\_open())      {          // 对于新商家，文件不存在是正常的          return true;      }      vector<Product \*> sellerProductsList;      string line;      while (getline(file, line))      {          stringstream ss(line);          string segment;          vector<string> seglist;          while (getline(ss, segment, ','))          {              seglist.push\_back(segment);          }          // 格式检查: 至少需要7个字段 (类型,名称,描述,价格,数量,折扣,商家)          if (seglist.size() < 7)          {              cerr << "无效的产品数据行: " << line << endl;              continue;          }          try          {              string type = seglist[0];              string name = seglist[1];              string desc = seglist[2];              double price = stod(seglist[3]);              int qty = stoi(seglist[4]);              double discount = stod(seglist[5]);              string seller = seglist[6]; // 第7个字段是商家名              // 确认商家名称匹配              if (seller != sellerUsername)              {                  cerr << "警告: 文件 " << filename << " 中的商品所属商家 \""                       << seller << "\" 与预期的 \"" << sellerUsername << "\" 不符" << endl;                  seller = sellerUsername; // 强制使用文件名指示的商家              }              Product \*newProduct = nullptr;              if (type == "Book" && seglist.size() >= 9)              {                  // Book需要额外的author和isbn字段                  newProduct = new Book(name, desc, price, qty, seglist[7], seglist[8], seller);              }              else if (type == "Clothing" && seglist.size() >= 9)              {                  // Clothing需要额外的size和color字段                  newProduct = new Clothing(name, desc, price, qty, seglist[7], seglist[8], seller);              }              else if (type == "Food" && seglist.size() >= 8)              {                  // Food需要额外的expirationDate字段                  newProduct = new Food(name, desc, price, qty, seglist[7], seller);              }              if (newProduct)              {                  newProduct->setDiscountRate(discount);                  allProducts.push\_back(newProduct);                  sellerProductsList.push\_back(newProduct);              }          }          catch (const exception &e)          {              cerr << "解析商品数据时出错: " << e.what() << " 行: " << line << endl;          }      }      file.close();      // 更新商家商品映射      sellerProducts[sellerUsername] = sellerProductsList;      cout << "已加载商家 \"" << sellerUsername << "\" 的 "           << sellerProductsList.size() << " 件商品" << endl;      return true;  }  // 保存所有商品  bool Store::saveAllProducts()  {      // 对于每个商家，保存其商品      set<string> processedSellers; // 用于跟踪已处理的商家      for (const auto &product : allProducts)      {          string seller = product->getSellerUsername();          if (!seller.empty() && processedSellers.find(seller) == processedSellers.end())          {              saveProductsForSeller(seller);              processedSellers.insert(seller);          }      }      return true;  }  // 保存指定商家的商品  bool Store::saveProductsForSeller(const string &sellerUsername)  {      if (sellerUsername.empty())      {          cerr << "错误: 尝试保存无效商家的商品" << endl;          return false;      }      vector<Product \*> products;      // 获取该商家的所有商品      for (Product \*p : allProducts)      {          if (p->getSellerUsername() == sellerUsername)          {              products.push\_back(p);          }      }      string filename = getSellerFilename(sellerUsername);      ofstream file(filename);      if (!file.is\_open())      {          cerr << "错误: 无法打开文件保存商品: " << filename << endl;          return false;      }      for (const Product \*p : products)      {          p->save(file);          file << endl;      }      file.close();      // cout << "商家 \"" << sellerUsername << "\" 的 " << products.size()      //      << " 件商品已保存至 " << filename << endl;      return true;  }  // 查找商品  Product \*Store::findProductByName(const string &name, const string &sellerUsername)  {      if (!sellerUsername.empty())      {          // 只在指定商家的商品中查找          auto it = sellerProducts.find(sellerUsername);          if (it != sellerProducts.end())          {              for (Product \*p : it->second)              {                  if (p->getName() == name)                  {                      return p;                  }              }          }          return nullptr;      }      else      {          // 在所有商品中查找          for (Product \*p : allProducts)          {              if (p->getName() == name)              {                  return p;              }          }          return nullptr;      }  }  // 显示所有商品  void Store::displayAllProducts() const  {      if (allProducts.empty())      {          cout << "\n--- 商品列表为空 ---" << endl;          return;      }      cout << "\n--- 所有商品列表 ---" << endl;      int count = 0;      for (const Product \*p : allProducts)      {          cout << "--------------------------" << endl;          p->display();          count++;      }      cout << "--------------------------" << endl;      cout << "共显示 " << count << " 件商品" << endl;  }  // 显示指定商家的商品  void Store::displaySellerProducts(const string &sellerUsername) const  {      auto it = sellerProducts.find(sellerUsername);      if (it == sellerProducts.end() || it->second.empty())      {          cout << "\n--- 商家 \"" << sellerUsername << "\" 没有商品 ---" << endl;          return;      }      cout << "\n--- 商家 \"" << sellerUsername << "\" 的商品列表 ---" << endl;      int count = 0;      for (const Product \*p : it->second)      {          cout << "--------------------------" << endl;          p->display();          count++;      }      cout << "--------------------------" << endl;      cout << "共显示 " << count << " 件商品" << endl;  }  // 按名称搜索商品  vector<Product \*> Store::searchProductsByName(const string &searchTerm, const string &sellerUsername) const  {      vector<Product \*> results;      const vector<Product \*> &productsToSearch = sellerUsername.empty() ? allProducts : (sellerProducts.count(sellerUsername) ? sellerProducts.at(sellerUsername) : vector<Product \*>());      for (Product \*p : productsToSearch)      {          // 简单的包含搜索 (不区分大小写)          string productName = p->getName();          string searchTermLower = searchTerm;          // 将两个字符串转为小写进行比较          transform(productName.begin(), productName.end(), productName.begin(), ::tolower);          transform(searchTermLower.begin(), searchTermLower.end(), searchTermLower.begin(), ::tolower);          if (productName.find(searchTermLower) != string::npos)          {              results.push\_back(p);          }      }      return results;  }  // 获取商家的商品  vector<Product \*> Store::getSellerProducts(const string &sellerUsername) const  {      auto it = sellerProducts.find(sellerUsername);      return (it != sellerProducts.end()) ? it->second : vector<Product \*>();  }  // 创建图书  bool Store::createBook(User \*currentUser, const string &name, const string &desc,                         double price, int qty, const string &author, const string &isbn)  {      if (!currentUser)      {          cerr << "错误: 用户未登录" << endl;          return false;      }      if (currentUser->getUserType() != "seller")      {          cerr << "错误: 只有商家可以添加商品" << endl;          return false;      }      string sellerUsername = currentUser->getUsername();      // 检查该商家是否已有同名商品      if (findProductByName(name, sellerUsername) != nullptr)      {          cerr << "错误: 您已有名为 \"" << name << "\" 的商品" << endl;          return false;      }      if (price < 0 || qty < 0)      {          cerr << "错误: 价格和数量不能为负数" << endl;          return false;      }      try      {          Book \*newBook = new Book(name, desc, price, qty, author, isbn, sellerUsername);          allProducts.push\_back(newBook);          // 更新商家商品映射          sellerProducts[sellerUsername].push\_back(newBook);          // 保存商家的商品          if (saveProductsForSeller(sellerUsername))          {              cout << "商品 \"" << name << "\" 添加成功！" << endl;              return true;          }          else          {              cout << "商品 \"" << name << "\" 添加失败！" << endl;              return false;          }      }      catch (const bad\_alloc &e)      {          cerr << "错误: 无法分配内存: " << e.what() << endl;          return false;      }      catch (const exception &e)      {          cerr << "错误: " << e.what() << endl;          return false;      }      catch (...)      {          cerr << "错误: 添加商品时发生未知错误" << endl;          return false;      }  }  // 创建服装  bool Store::createClothing(User \*currentUser, const string &name, const string &desc,                             double price, int qty, const string &size, const string &color)  {      if (!currentUser)      {          cerr << "错误: 用户未登录" << endl;          return false;      }      if (currentUser->getUserType() != "seller")      {          cerr << "错误: 只有商家可以添加商品" << endl;          return false;      }      string sellerUsername = currentUser->getUsername();      // 检查该商家是否已有同名商品      if (findProductByName(name, sellerUsername) != nullptr)      {          cerr << "错误: 您已有名为 \"" << name << "\" 的商品" << endl;          return false;      }      if (price < 0 || qty < 0)      {          cerr << "错误: 价格和数量不能为负数" << endl;          return false;      }      try      {          Clothing \*newClothing = new Clothing(name, desc, price, qty, size, color, sellerUsername);          allProducts.push\_back(newClothing);          // 更新商家商品映射          sellerProducts[sellerUsername].push\_back(newClothing);          // 保存商家的商品          if (saveProductsForSeller(sellerUsername))          {              cout << "商品 \"" << name << "\" 添加成功！" << endl;              return true;          }          else          {              cout << "商品 \"" << name << "\" 添加失败！" << endl;              return false;          }      }      catch (const bad\_alloc &e)      {          cerr << "错误: 无法分配内存: " << e.what() << endl;          return false;      }      catch (const exception &e)      {          cerr << "错误: " << e.what() << endl;          return false;      }      catch (...)      {          cerr << "错误: 添加商品时发生未知错误" << endl;          return false;      }  }  // 创建食品  bool Store::createFood(User \*currentUser, const string &name, const string &desc,                         double price, int qty, const string &expDate)  {      if (!currentUser)      {          cerr << "错误: 用户未登录" << endl;          return false;      }      if (currentUser->getUserType() != "seller")      {          cerr << "错误: 只有商家可以添加商品" << endl;          return false;      }      string sellerUsername = currentUser->getUsername();      // 检查该商家是否已有同名商品      if (findProductByName(name, sellerUsername) != nullptr)      {          cerr << "错误: 您已有名为 \"" << name << "\" 的商品" << endl;          return false;      }      if (price < 0 || qty < 0)      {          cerr << "错误: 价格和数量不能为负数" << endl;          return false;      }      try      {          Food \*newFood = new Food(name, desc, price, qty, expDate, sellerUsername);          allProducts.push\_back(newFood);          // 更新商家商品映射          sellerProducts[sellerUsername].push\_back(newFood);          // 保存商家的商品          if (saveProductsForSeller(sellerUsername))          {              cout << "商品 \"" << name << "\" 添加成功！" << endl;              return true;          }          else          {              cout << "商品 \"" << name << "\" 添加失败！" << endl;              return false;          }      }      catch (const bad\_alloc &e)      {          cerr << "错误: 无法分配内存: " << e.what() << endl;          return false;      }      catch (const exception &e)      {          cerr << "错误: " << e.what() << endl;          return false;      }      catch (...)      {          cerr << "错误: 添加商品时发生未知错误" << endl;          return false;      }  }  bool Store::createGeneral(User \*currentUser, const string &name, const string &desc,                            double price, int qty, const string &generalType)  {      if (!currentUser)      {          cerr << "错误: 用户未登录" << endl;          return false;      }      if (currentUser->getUserType() != "seller")      {          cerr << "错误: 只有商家可以添加商品" << endl;          return false;      }      string sellerUsername = currentUser->getUsername();      // 检查该商家是否已有同名商品      if (findProductByName(name, sellerUsername) != nullptr)      {          cerr << "错误: 您已有名为 \"" << name << "\" 的商品" << endl;          return false;      }      if (price < 0 || qty < 0)      {          cerr << "错误: 价格和数量不能为负数" << endl;          return false;      }      try      {          General \*newGeneral = new General(name, desc, price, qty, generalType, sellerUsername);          allProducts.push\_back(newGeneral);          // 更新商家商品映射          sellerProducts[sellerUsername].push\_back(newGeneral);          // 保存商家的商品          if (saveProductsForSeller(sellerUsername))          {              cout << "商品 \"" << name << "\" 添加成功！" << endl;              return true;          }          else          {              cout << "商品 \"" << name << "\" 添加失败！" << endl;              return false;          }      }      catch (const bad\_alloc &e)      {          cerr << "错误: 无法分配内存: " << e.what() << endl;          return false;      }      catch (const exception &e)      {          cerr << "错误: " << e.what() << endl;          return false;      }  }  // 修改商品价格  bool Store::manageProductPrice(User \*currentUser, const string &productName, double newPrice)  {      if (!currentUser)      {          cerr << "错误: 用户未登录" << endl;          return false;      }      if (currentUser->getUserType() != "seller")      {          cerr << "错误: 只有商家可以修改商品价格" << endl;          return false;      }      string sellerUsername = currentUser->getUsername();      Product \*product = findProductByName(productName, sellerUsername);      if (!product)      {          cerr << "错误: 未找到您的商品 \"" << productName << "\"" << endl;          return false;      }      if (newPrice < 0)      {          cerr << "错误: 价格不能为负数" << endl;          return false;      }      product->setOriginalPrice(newPrice);      // 保存商家的商品      if (saveProductsForSeller(sellerUsername))      {          cout << "价格保存成功！" << endl;          return true;      }      else      {          cout << "价格保存失败！" << endl;          return false;      }  }  // 修改商品库存  bool Store::manageProductQuantity(User \*currentUser, const string &productName, int newQuantity)  {      if (!currentUser)      {          cerr << "错误: 用户未登录" << endl;          return false;      }      if (currentUser->getUserType() != "seller")      {          cerr << "错误: 只有商家可以修改商品库存" << endl;          return false;      }      string sellerUsername = currentUser->getUsername();      Product \*product = findProductByName(productName, sellerUsername);      if (!product)      {          cerr << "错误: 未找到您的商品 \"" << productName << "\"" << endl;          return false;      }      if (newQuantity < 0)      {          cerr << "错误: 库存不能为负数" << endl;          return false;      }      product->setQuantity(newQuantity);      if (saveProductsForSeller(sellerUsername))      {          cout << "库存修改成功！" << endl;          return true;      }      else      {          cout << "库存修改失败！" << endl;          return false;      }  }  // 修改商品折扣  bool Store::manageProductDiscount(User \*currentUser, const string &productName, double newDiscount)  {      if (!currentUser)      {          cerr << "错误: 用户未登录" << endl;          return false;      }      if (currentUser->getUserType() != "seller")      {          cerr << "错误: 只有商家可以修改商品折扣" << endl;          return false;      }      string sellerUsername = currentUser->getUsername();      Product \*product = findProductByName(productName, sellerUsername);      if (!product)      {          cerr << "错误: 未找到您的商品 \"" << productName << "\"" << endl;          return false;      }      if (newDiscount < 0 || newDiscount > 100)      {          cerr << "错误: 折扣必须在0到100之间" << endl;          return false;      }      product->setDiscountRate(newDiscount);      return saveProductsForSeller(sellerUsername);  }  // 应用分类折扣  bool Store::applyCategoryDiscount(User \*currentUser, const string &category, double discount)  {      if (!currentUser)      {          cerr << "错误: 用户未登录" << endl;          return false;      }      if (currentUser->getUserType() != "seller")      {          cerr << "错误: 只有商家可以应用折扣" << endl;          return false;      }      if (discount < 0.0 || discount > 1.0)      {          cerr << "错误: 折扣必须在0到100之间" << endl;          return false;      }      string sellerUsername = currentUser->getUsername();      bool changed = false; // 只应用折扣到该商家的商品      auto it = sellerProducts.find(sellerUsername);      if (it != sellerProducts.end())      {          for (Product \*p : it->second)          {              // 处理标准类型以及自定义类型              if (p->getType() == category)              {                  p->setDiscountRate(discount);                  changed = true;              }              // 处理特殊情况："其他"选择时应用于所有非标准类型商品              else if (category == "其他" && p->getType() != "Book" && p->getType() != "Clothing" && p->getType() != "Food")              {                  p->setDiscountRate(discount);                  changed = true;              }          }      }      if (changed)      {          return saveProductsForSeller(sellerUsername);      }      cerr << "错误: 未找到分类为 \"" << category << "\" 的商品" << endl;      return false;  } |

## 综合2.cpp

### Terminal

#### Start.cpp

|  |
| --- |
| #include <iostream>  #include <vector>  #include <string>  #include <limits>  #include "user/user.h"  #include "page/page.h"   // 包含 Page 基类和派生类  #include "store/store.h" // 包含 Store 类  #include "ordermanager/ordermanager.h"  #include <cmath>  #include <windows.h> // Windows 特有的头文件，用于 Sleep 函数  using namespace std;  // 文件路径常量  extern const string USER\_FILE = "./user/users.txt";  extern const string STORE\_FILE = "./store";  extern const string CART\_FILE = "./user/carts";  extern const string ORDER\_DIR = "./order/orders"; // 订单目录  bool startProcessing;                             // 用于控制订单处理线程的启动状态  OrderManager g\_orderManager(ORDER\_DIR); // 创建订单管理器对象  int main()  {      SetConsoleOutputCP(65001); // 设置控制台输出编码为 UTF-8      SetConsoleCP(65001);       // 设置控制台输入编码为 UTF-8      vector<User \*> users;      User \*currentUser = nullptr;      // 创建 Store 对象      Store store(STORE\_FILE);      // 创建具体的 Page 派生类对象      Customer\_page customerPage;      Seller\_page sellerPage;      // Page\* currentPageManager = &customerPage; // 如果需要基类指针进行多态调用      // 加载用户数据      users = User::loadUsersFromFile(USER\_FILE);      cout << "已加载 " << users.size() << " 个用户数据。" << endl;      // 启动订单处理线程      // startProcessing = false;      g\_orderManager.startProcessingThread(store, users);      // cout << "订单处理线程已启动。" << endl;      int menuResult = 1; // 1=继续, 0=退出程序, 2=退出登录, 3=进入商城      while (menuResult != 0)      {          // if (g\_orderManager.getPendingOrderCount() > 0)          // {          //     g\_orderManager.processNextOrder(store, users);          // }          // 根据 menuResult 处理状态          if (menuResult == 2)          { // 处理退出登录              currentUser = nullptr;              menuResult = 1;          }          if (menuResult == 3)          { // 处理进入商城              if (currentUser == nullptr || dynamic\_cast<Customer \*>(currentUser))              {                  menuResult = customerPage.pagestore(currentUser, users, store);              }              else if (dynamic\_cast<Seller \*>(currentUser))              {                  menuResult = sellerPage.pagestore(currentUser, users, store);              }              else              {                  cout << "当前用户类型无法进入商城。" << endl;                  menuResult = 1;              }              if (menuResult != 0 && menuResult != 2)                  menuResult = 1; // 确保返回后继续菜单循环              continue;          }          // 显示用户账户菜单          if (currentUser == nullptr) // 未登录          {              // 使用 customerPage 对象显示主菜单              menuResult = customerPage.pagemain(users, currentUser);          }          else // 已登录          {              if (dynamic\_cast<Customer \*>(currentUser))              {                  menuResult = customerPage.pagemainCustomer(currentUser, users);              }              else if (dynamic\_cast<Seller \*>(currentUser))              {                  menuResult = sellerPage.pagemainSeller(currentUser, users);              }              // else if (dynamic\_cast<Admin\*>(currentUser)) { ... }              else // 使用字符串匹配作为后备              {                  string type = currentUser->getUserType();                  if (type == "消费者")                  {                      menuResult = customerPage.pagemainCustomer(currentUser, users);                  }                  else if (type == "商家")                  {                      menuResult = sellerPage.pagemainSeller(currentUser, users);                  }                  // else if (type == "管理员") { ... }                  else                  {                      cout << "未知的用户类型 (" << type << ")，将退出登录。" << endl;                      menuResult = 2;                  }              }          }      } // End while loop      // ... (程序结束部分保持不变) ...      User::saveUsersToFile(users, USER\_FILE);      cout << "用户数据已保存。" << endl;      for (auto \*user : users)      {          delete user;      }      users.clear();      return 0;  } |

#### User.h

|  |
| --- |
| #ifndef USER\_H  #define USER\_H  #include <iostream>  #include <string>  #include <vector>  #include <fstream>  #include <limits>  #include <iomanip>          // For output formatting in cart display  #include "../order/order.h" // For OrderItem  // 前向声明  class Product;  class Store;  // 购物车中的商品项 (现在作为 Customer 类的内部结构或辅助结构)  struct CartItem  {      std::string productId; // 商品名称或唯一ID      std::string productName;      int quantity;      double priceAtAddition; // 添加到购物车时的价格      std::string sellerUsername;      CartItem(std::string pid = "", std::string pName = "", int qty = 0, double price = 0.0, std::string sName = "")          : productId(pid), productName(pName), quantity(qty), priceAtAddition(price), sellerUsername(sName) {}  };  class User  {  protected:      std::string username;      std::string password;      std::string userType;      double balance;      static void clearInputBuffer()      {          std::cin.clear();          std::cin.ignore(std::numeric\_limits<std::streamsize>::max(), '\n');      }  public:      User(std::string username, std::string password, double balance = 0.0,           bool is\_customer = false, bool is\_seller = false, bool is\_admin = false);      User();      virtual ~User() = default;      virtual std::string getUserType() const = 0;      void displayInfo(int index = -1) const;      bool changePassword(const std::string &oldPassword, const std::string &newPassword);      bool deposit(double amount);      bool withdraw(double amount);      double checkBalance() const;      static User \*registerUser(std::vector<User \*> &users);      static User \*userLogin(const std::vector<User \*> &users);      static bool saveUsersToFile(const std::vector<User \*> &users, const std::string &filename);      static std::vector<User \*> loadUsersFromFile(const std::string &filename);      static User \*findUser(const std::vector<User \*> &users, const std::string &username);      static void displayAllUsers(const std::vector<User \*> &users);      static bool isUsernameExists(const std::vector<User \*> &users, const std::string &username);      std::string getUsername() const;      std::string getPassword() const;      void setUsername(const std::string &newUsername);      void setPassword(const std::string &newPassword);      void setBalance(double newBalance);      static User \*login(const std::vector<User \*> &users, const std::string &username, const std::string &password);  };  // 消费者类  class Customer : public User  {  private:      std::string cartDirectoryPath; // 存储购物车文件的目录路径      // 购物车文件操作的辅助方法      std::string getCartFilePath() const;  public:      Customer(std::string username, std::string password, const std::string &cartDir, double balance = 0.0);      Customer(const std::string &cartDir); // 默认构造函数      std::string getUserType() const override { return "customer"; }      std::vector<CartItem> shoppingCartItems; // 直接在 Customer 中存储购物车项      void loadCartFromFile();      bool saveCartToFile() const;      // 购物车管理方法      bool addToCart(const Product &product, int quantity);      bool removeCartItem(const std::string &productName);      bool updateCartItemQuantity(const std::string &productName, int newQuantity, Store &store); // 需要 store 检查库存      void viewCart(Store &store, std::vector<User \*> &allUsers);                                 // 查看并管理购物车，结算也在这里处理      bool isCartEmpty() const;      void clearCartAndFile(); // 结算后清空购物车及文件  };  // 商家类  class Seller : public User  {  public:      Seller(std::string username, std::string password, double balance = 0.0);      Seller();      std::string getUserType() const override { return "seller"; }  };  // 管理员类  class Admin : public User  {  public:      Admin(std::string username, std::string password, double balance = 0.0);      Admin();      std::string getUserType() const override { return "admin"; }  };  #endif // USER\_H |

#### User.cpp

|  |
| --- |
| #include "user.h"  #include "../store/store.h" // 需要 Product 和 Store 的完整定义  #include "../order/order.h"  #include "../ordermanager/ordermanager.h"  #include <iostream>  #include <limits>  #include <string>  #include <fstream>  #include <sstream>  #include <filesystem> // 用于文件操作  #include <algorithm>  // for std::remove\_if  #include <iomanip>    // for std::fixed, std::setprecision  #include <thread>  using namespace std;  // 声明在 start.cpp 中定义的全局常量  extern const string CART\_FILE; // 用于 Customer 构造函数  extern const string USER\_FILE; // 用于保存用户信息  // User 构造函数实现  User::User(std::string u\_name, std::string pwd, double bal, bool is\_customer, bool is\_seller, bool is\_admin)      : username(u\_name), password(pwd), balance(bal)  {      // userType 在派生类中设置  }  User::User() : username(""), password(""), balance(0.0), userType("unknown") {}  // --- Customer 类购物车相关方法实现 ---  std::string Customer::getCartFilePath() const  {      if (this->username.empty())      { // 防止用户名为空时生成无效路径          // cerr << "警告: Customer 用户名为空，无法生成购物车路径。" << endl;          return ""; // 或者抛出异常，或者返回一个默认的无效路径      }      return this->cartDirectoryPath + "/" + this->username + "\_cart.txt";  }  void Customer::loadCartFromFile()  {      shoppingCartItems.clear();      std::string cartPath = getCartFilePath();      if (cartPath.empty())          return; // 用户名为空，无法加载      std::ifstream file(cartPath);      if (!file.is\_open())      {          return; // 文件不存在或无法打开是正常情况（购物车为空）      }      std::string line;      while (getline(file, line))      {          std::stringstream ss(line);          std::string productId, productName, sellerUsernameStr;          int quantity;          double priceAtAddition;          if (getline(ss, productId, ',') &&              getline(ss, productName, ',') &&              (ss >> quantity) &&              (ss.ignore(1) && ss >> priceAtAddition) &&              (ss.ignore(1) && getline(ss, sellerUsernameStr)))          {              shoppingCartItems.emplace\_back(productId, productName, quantity, priceAtAddition, sellerUsernameStr);          }          else          {              std::cerr << "警告: 购物车文件 " << cartPath << " 中存在格式错误的行: " << line << std::endl;          }      }      file.close();  }  bool Customer::saveCartToFile() const  {      std::string cartPath = getCartFilePath();      if (cartPath.empty())      {          std::cerr << "错误: Customer 用户名为空，无法保存购物车。" << std::endl;          return false;      }      std::ofstream file(cartPath); // 会覆盖旧文件      if (!file.is\_open())      {          std::cerr << "错误: 无法打开购物车文件进行写入: " << cartPath << std::endl;          return false;      }      for (const auto &item : shoppingCartItems)      {          file << item.productId << ","               << item.productName << ","               << item.quantity << ","               << item.priceAtAddition << ","               << item.sellerUsername << std::endl;      }      file.close();      return true;  }  // Customer 构造函数实现  Customer::Customer(std::string uname, std::string pwd, const std::string &cartDir, double bal)      : User(uname, pwd, bal, true, false, false), cartDirectoryPath(cartDir)  {      this->userType = "customer";      if (!this->username.empty())      { // 只有当用户名有效时才加载购物车          loadCartFromFile();      }  }  // 默认构造函数  Customer::Customer(const std::string &cartDir)      : User(), cartDirectoryPath(cartDir)  {      this->userType = "customer";      // 注意: 此时 username 为空，loadCartFromFile 不会执行。      // 如果后续通过 setUsername 设置了用户名，需要一种机制来加载对应的购物车。      // 或者，强制 Customer 总是在有用户名的情况下创建。  }  bool Customer::addToCart(const Product &product, int quantity)  {      if (this->username.empty())      {          cout << "错误：无法为匿名用户添加购物车。" << endl;          return false;      }      if (quantity <= 0)      {          cout << "添加数量必须大于0。" << endl;          return false;      }      bool found = false;      for (auto &item : shoppingCartItems)      {          if (item.productId == product.getName())          { // 假设商品名称是唯一ID              item.quantity += quantity;              // 可选：更新 priceAtAddition 为最新价格或保持不变              found = true;              break;          }      }      if (!found)      {          shoppingCartItems.emplace\_back(product.getName(), product.getName(), quantity, product.getPrice(), product.getSellerUsername());      }      cout << "\"" << product.getName() << "\" 已成功加入购物车。" << endl;      return saveCartToFile();  }  bool Customer::removeCartItem(const std::string &productName)  {      if (this->username.empty())          return false;      auto initial\_size = shoppingCartItems.size();      shoppingCartItems.erase(std::remove\_if(shoppingCartItems.begin(), shoppingCartItems.end(),                                             [&](const CartItem &item)                                             { return item.productName == productName; }),                              shoppingCartItems.end());      if (shoppingCartItems.size() < initial\_size)      {          cout << "商品 \"" << productName << "\" 已从购物车移除。" << endl;          return saveCartToFile();      }      else      {          cout << "未在购物车中找到商品: " << productName << endl;          return false;      }  }  bool Customer::updateCartItemQuantity(const std::string &productName, int newQuantity, Store &store)  {      if (this->username.empty())          return false;      if (newQuantity < 0)      {          cout << "数量不能为负。" << endl;          return false;      }      bool updated = false;      for (auto &item : shoppingCartItems)      {          if (item.productName == productName)          {              Product \*p\_info = store.findProductByName(productName); // 检查最新库存              if (p\_info && newQuantity > p\_info->getQuantity())              {                  cout << "库存不足！无法将购物车中 \"" << productName << "\" 的数量修改为 " << newQuantity                       << "。当前库存: " << p\_info->getQuantity() << endl;                  return false;              }              item.quantity = newQuantity;              updated = true;              break;          }      }      if (!updated)      {          cout << "未在购物车中找到商品: " << productName << endl;          return false;      }      // 移除数量为0的商品      shoppingCartItems.erase(std::remove\_if(shoppingCartItems.begin(), shoppingCartItems.end(),                                             [](const CartItem &item)                                             { return item.quantity <= 0; }),                              shoppingCartItems.end());      cout << "购物车商品 \"" << productName << "\" 数量已更新。" << endl;      return saveCartToFile();  }  void Customer::clearCartAndFile()  {      if (this->username.empty())          return;      shoppingCartItems.clear();      std::string cartPath = getCartFilePath();      if (!cartPath.empty() && std::filesystem::exists(cartPath))      {          try          {              std::filesystem::remove(cartPath);          }          catch (const std::filesystem::filesystem\_error &err)          {              std::cerr << "错误: 删除购物车文件失败: " << err.what() << std::endl;          }      }  }  bool Customer::isCartEmpty() const  {      return shoppingCartItems.empty();  }  void Customer::viewCart(Store &store, std::vector<User \*> &allUsers)  {      if (this->username.empty())      {          cout << "错误：匿名用户无法查看购物车。" << endl;          return;      }      // 确保购物车是最新的（如果其他地方可能修改文件但未更新内存）      // loadCartFromFile(); // 通常在每次操作后保存，所以这里可能不需要，除非有外部修改购物车的可能      if (shoppingCartItems.empty())      {          cout << "\n您的购物车是空的。" << endl;          return;      }      double totalCartPriceBasedOnCurrent = 0;      cout << "\n--- 我的购物车 ---" << endl;      cout << std::fixed << std::setprecision(2); // 设置输出格式      for (size\_t i = 0; i < shoppingCartItems.size(); ++i)      {          const auto &item = shoppingCartItems[i];          Product \*p\_info = store.findProductByName(item.productId);          double currentItemPrice = p\_info ? p\_info->getPrice() : item.priceAtAddition;          cout << i + 1 << ". 商品: " << item.productName               << " | 数量: " << item.quantity               << " | 当前单价: ¥" << currentItemPrice               << " | 小计: ¥" << (item.quantity \* currentItemPrice) << endl;          if (p\_info)          {              if (p\_info->getQuantity() < item.quantity)              {                  cout << "   注意: " << item.productName << " 当前库存(" << p\_info->getQuantity() << ")不足购物车数量(" << item.quantity << ")!" << endl;              }              if (p\_info->getPrice() != item.priceAtAddition)              {                  cout << "   提示: " << item.productName << " 加入时价格为 ¥" << item.priceAtAddition << ", 当前价格已变为 ¥" << p\_info->getPrice() << endl;              }          }          else          {              cout << "   警告: 商品 " << item.productName << " 可能已从商店下架!" << endl;          }          totalCartPriceBasedOnCurrent += item.quantity \* currentItemPrice;      }      cout << "--------------------" << endl;      cout << "购物车总计 (按当前价格): ¥" << totalCartPriceBasedOnCurrent << endl;      cout << "--------------------" << endl;      int cartChoice = -1;      do      {          cout << "\n购物车操作:" << endl;          cout << "1. 修改商品数量" << endl;          cout << "2. 移除商品" << endl;          cout << "3. 生成订单并结算" << endl;          cout << "0. 返回商城" << endl;          cout << "请选择: ";          if (!(cin >> cartChoice))          {              cout << "无效输入。" << endl;              User::clearInputBuffer();              cartChoice = -1;              continue;          }          User::clearInputBuffer();          switch (cartChoice)          {          case 1:          {              string nameToUpdate;              int newQty;              cout << "输入要修改数量的商品名称: ";              getline(cin >> ws, nameToUpdate);              cout << "输入新的数量 (输入0将移除该商品): ";              if (!(cin >> newQty))              {                  cout << "无效数量输入。" << endl;                  User::clearInputBuffer();                  break;              }              User::clearInputBuffer();              if (updateCartItemQuantity(nameToUpdate, newQty, store))              {                  // 成功后重新加载并显示购物车 (递归调用或重新加载数据)                  // 为了避免无限递归，这里直接返回，让外层循环重新调用 viewCart                  cout << "购物车已更新。" << endl;              }              return viewCart(store, allUsers); // 重新显示更新后的购物车          }          case 2:          {              string nameToRemove;              cout << "输入要移除的商品名称: ";              getline(cin >> ws, nameToRemove);              if (removeCartItem(nameToRemove))              {                  cout << "购物车已更新。" << endl;              }              return viewCart(store, allUsers); // 重新显示更新后的购物车          }          case 3: // Generate Order and Checkout          {              if (shoppingCartItems.empty())              {                  cout << "购物车是空的，无法生成订单。" << endl;                  break;              }              Order currentOrder(this->getUsername()); // Create a new order object              currentOrder.setStatus("PENDING\_VALIDATION");              // 第一阶段：验证商品并构建订单（预检查）              bool preCheckOk = true;              for (const auto &cart\_item : shoppingCartItems)              {                  Product \*product = store.findProductByName(cart\_item.productId);                  if (!product)                  {                      cout << "错误: 购物车商品 \"" << cart\_item.productName << "\" 已不存在。" << endl;                      currentOrder.setStatus("CANCELLED\_PRODUCT\_NOT\_FOUND");                      preCheckOk = false;                      break;                  }                  if (product->getQuantity() < cart\_item.quantity)                  {                      cout << "错误: 购物车商品 \"" << cart\_item.productName << "\" 库存不足 (需要 "                           << cart\_item.quantity << ", 现有 " << product->getQuantity() << ")。" << endl;                      currentOrder.setStatus("CANCELLED\_INSUFFICIENT\_STOCK");                      preCheckOk = false;                      break;                  }                  currentOrder.addItemFromProduct(\*product, cart\_item.quantity);              }              if (!preCheckOk)              {                  currentOrder.displaySummary(); // Display order with cancelled status                  cout << "订单无法继续，请调整购物车后重试。" << endl;                  break; // Back to cart menu              }              // 第二阶段：验证客户余额（只是初步检查，队列处理时还会再次检查）              if (this->checkBalance() < currentOrder.getTotalAmount())              {                  cout << "错误: 您的余额不足以支付此订单！" << endl;                  currentOrder.setStatus("CANCELLED\_INSUFFICIENT\_FUNDS");                  // currentOrder.displaySummary();                  cout << "您的余额:   ¥" << this->checkBalance() << endl;                  break; // Back to cart menu              }              // 第三阶段：显示订单并请求确认              currentOrder.setStatus("PENDING\_CONFIRMATION");              currentOrder.displaySummary();              cout << "您的当前余额: ¥" << std::fixed << std::setprecision(2) << this->checkBalance() << endl;              cout << "支付后余额将为: ¥" << (this->checkBalance() - currentOrder.getTotalAmount()) << endl;              cout << "--------------------" << endl;              char confirmChoice;              cout << "确认支付并完成订单吗? (y/n): ";              cin >> confirmChoice;              User::clearInputBuffer();              if (tolower(confirmChoice) != 'y')              {                  currentOrder.setStatus("CANCELLED\_BY\_USER");                  currentOrder.displaySummary();                  cout << "订单已取消。" << endl;                  break; // Back to cart menu              }              // Stage 4: Process payment and update inventory (Actual "lock" and transaction)              cout << "\n正在处理订单，请稍候..." << endl;              currentOrder.setStatus("PROCESSING");              // 访问OrderManager，提交订单 - 修改这部分              extern OrderManager g\_orderManager;              auto submittedOrder = g\_orderManager.submitOrderRequest(currentOrder);              cout << "订单已成功提交到处理队列，订单ID: " << submittedOrder->getOrderId() << endl;              cout << "正在等待订单处理，请稍候..." << endl;              // 等待订单处理完成              int waitCounter = 0;              const int MAX\_WAIT\_CYCLES = 100; // 最多等待10秒              while (!submittedOrder->getProcessed() && waitCounter < MAX\_WAIT\_CYCLES)              {                  // 短暂休眠，避免过度消耗CPU                  std::this\_thread::sleep\_for(std::chrono::milliseconds(100));                  waitCounter++;                  // 每秒显示等待状态                  if (waitCounter % 10 == 0)                  {                      cout << "已等待 " << (waitCounter / 10) << " 秒..." << endl;                  }              }              // 检查是否超时              if (waitCounter >= MAX\_WAIT\_CYCLES)              {                  cout << "等待订单处理超时！订单可能仍在处理中。" << endl;              }              else              {                  cout << "订单已处理完成! 状态: " << submittedOrder->getStatus() << endl;                  if (submittedOrder->getStatus() == "COMPLETED" ||                      submittedOrder->getStatus() == "COMPLETED\_WITH\_PAYMENT\_ISSUES")                  {                      clearCartAndFile(); // 成功提交后清空购物车                  }              }              cout << std::resetiosflags(std::ios\_base::fixed | std::ios\_base::floatfield);              return; // 退出viewCart，返回Customer\_page商店菜单              // if (!this->withdraw(currentOrder.getTotalAmount()))              // {              //     currentOrder.setStatus("FAILED\_PAYMENT\_WITHDRAWAL");              //     currentOrder.displaySummary();              //     cout << "错误: 支付失败。请检查余额或联系客服。" << endl;              //     // Potentially log this critical failure              //     break;              // }              // for (const auto &order\_item : currentOrder.getItems())              // {              //     Product \*product = store.findProductByName(order\_item.productId); // Re-fetch, though should be same              //     if (product)              //     { // Should always be true due to pre-checks              //         product->setQuantity(product->getQuantity() - order\_item.quantity);              //         User \*seller = User::findUser(allUsers, order\_item.sellerUsername);              //         if (seller)              //         {              //             if (!seller->deposit(order\_item.priceAtPurchase \* order\_item.quantity))              //             {              //                 cerr << "警告: 向商家 " << seller->getUsername() << " 转账失败 (商品: "              //                      << order\_item.productName << "). 订单号: " << currentOrder.getOrderId() << endl;              //                 // Log this for reconciliation              //             }              //         }              //         else if (!order\_item.sellerUsername.empty())              //         {              //             cerr << "警告: 未找到商品 \"" << order\_item.productName << "\" 的商家 \""              //                  << order\_item.sellerUsername << "\". 订单号: " << currentOrder.getOrderId() << endl;              //         }              //         else              //         {              //             cerr << "警告: 商品 \"" << order\_item.productName << "\" 没有商家信息. 订单号: " << currentOrder.getOrderId() << endl;              //         }              //     }              // }              // User::saveUsersToFile(allUsers, USER\_FILE);              // store.saveAllProducts(); // Or save only affected sellers              // currentOrder.setStatus("COMPLETED");              // cout << "\n订单处理完成！" << endl;              // currentOrder.displaySummary(); // Display final completed order              // // Optional: currentOrder.saveToFile(ORDER\_DIRECTORY\_BASE + "/" + this->getUsername());              // clearCartAndFile(); // Clear cart after successful order              // cout << std::resetiosflags(std::ios\_base::fixed | std::ios\_base::floatfield);              // return; // Exit viewCart, back to Customer\_page store menu          }          case 0:              // 返回商城，什么也不做，循环会结束              break;          default:              cout << "无效选项。" << endl;          }      } while (cartChoice != 0);      cout << std::resetiosflags(std::ios\_base::fixed | std::ios\_base::floatfield); // 重置输出格式  }  // Seller 构造函数实现  Seller::Seller(std::string uname, std::string pwd, double bal)      : User(uname, pwd, bal, false, true, false)  {      this->userType = "seller";  }  Seller::Seller() : User() { this->userType = "seller"; }  // Admin 构造函数实现  Admin::Admin(std::string uname, std::string pwd, double bal)      : User(uname, pwd, bal, false, false, true)  {      this->userType = "admin";  }  Admin::Admin() : User() { this->userType = "admin"; }  // --- 静态方法：用户注册实现 ---  User \*User::registerUser(std::vector<User \*> &users)  {      string newUsername, newPassword;      double initialBalance = 0.0;      int userTypeChoice;      cout << "请输入新用户名: ";      getline(cin >> ws, newUsername);      if (isUsernameExists(users, newUsername))      {          cout << "用户名已存在！" << endl;          return nullptr;      }      cout << "请输入密码: ";      getline(cin >> ws, newPassword);      cout << "选择用户类型 (1: 消费者, 2: 商家): ";      if (!(cin >> userTypeChoice) || (userTypeChoice != 1 && userTypeChoice != 2))      {          cout << "无效的用户类型选择。" << endl;          clearInputBuffer();          return nullptr;      }      clearInputBuffer();      User \*newUser = nullptr;      if (userTypeChoice == 1)      {                                                                                // Customer          newUser = new Customer(newUsername, newPassword, CART\_FILE, initialBalance); // 使用 CART\_FILE      }      else      { // Seller          newUser = new Seller(newUsername, newPassword, initialBalance);      }      if (newUser)      {          users.push\_back(newUser);          cout << (userTypeChoice == 1 ? "消费者" : "商家") << "用户 \"" << newUsername << "\" 注册成功！" << endl;      }      return newUser;  }  // 从文件加载用户信息  std::vector<User \*> User::loadUsersFromFile(const std::string &filename)  {      std::vector<User \*> loadedUsers;      std::ifstream file(filename);      if (!file.is\_open())      {          return loadedUsers;      }      std::string line;      while (getline(file, line))      {          std::stringstream ss(line);          std::string uname, pwd, type;          double bal;          if (              getline(ss, uname, ',') &&              getline(ss, pwd, ',') &&              getline(ss, type, ',') &&              (ss >> bal))          {              if (type == "customer")              {                  loadedUsers.push\_back(new Customer(uname, pwd, CART\_FILE, bal)); // 使用 CART\_FILE              }              else if (type == "seller")              {                  loadedUsers.push\_back(new Seller(uname, pwd, bal));              }              else if (type == "admin")              {                  loadedUsers.push\_back(new Admin(uname, pwd, bal));              }          }      }      file.close();      return loadedUsers;  }  // --- User 类其他成员函数实现 ---  // getUsername, getPassword, displayInfo, changePassword, deposit, withdraw, checkBalance,  // saveUsersToFile, findUser, isUsernameExists, login  // 这些方法的实现应该保持不变，除非它们需要与新的购物车逻辑交互（但看起来不需要）。  std::string User::getUsername() const { return username; }  std::string User::getPassword() const { return password; } // 密码不应这样直接返回，仅为示例  void User::setUsername(const std::string &newUsername) { username = newUsername; }  void User::setPassword(const std::string &newPassword) { password = newPassword; }  void User::setBalance(double newBalance)  {      if (newBalance >= 0)          balance = newBalance;  }  void User::displayInfo(int index) const  {      if (index != -1)      {          cout << "用户 " << index << ": ";      }      cout << "用户名: " << username << ", 类型: " << userType << ", 余额: " << balance << endl;  }  bool User::changePassword(const std::string &oldPassword, const std::string &newPassword)  {      if (this->password == oldPassword)      {          this->password = newPassword;          return true;      }      cout << "原密码错误！" << endl;      return false;  }  bool User::deposit(double amount)  {      if (amount <= 0)      {          cout << "充值金额必须为正！" << endl;          return false;      }      balance += amount;      return true;  }  bool User::withdraw(double amount)  {      if (amount <= 0)      {          cout << "消费金额必须为正！" << endl;          return false;      }      if (balance < amount)      {          cout << "余额不足！" << endl;          return false;      }      balance -= amount;      return true;  }  double User::checkBalance() const  {      return balance;  }  bool User::saveUsersToFile(const std::vector<User \*> &users, const std::string &filename)  {      ofstream file(filename);      if (!file.is\_open())      {          cerr << "错误: 无法打开用户文件进行写入: " << filename << endl;          return false;      }      for (const auto \*user : users)      {          if (user)          {              file << user->getUsername() << ","                   << user->getPassword() << ","                   << user->getUserType() << ","                   << user->checkBalance() << endl;          }      }      file.close();      return true;  }  User \*User::findUser(const std::vector<User \*> &users, const std::string &findUsername)  {      for (User \*user : users)      {          if (user->getUsername() == findUsername)          {              return user;          }      }      return nullptr;  }  bool User::isUsernameExists(const std::vector<User \*> &users, const std::string &checkUsername)  {      return findUser(users, checkUsername) != nullptr;  }  User \*User::userLogin(const std::vector<User \*> &users)  {      string loginUsername, loginPassword;      cout << "请输入用户名: ";      getline(cin >> ws, loginUsername);      cout << "请输入密码: ";      getline(cin >> ws, loginPassword);      User \*foundUser = findUser(users, loginUsername);      if (foundUser && foundUser->getPassword() == loginPassword)      {          cout << "登录成功！欢迎, " << loginUsername << "!" << endl;          // 如果是 Customer，并且之前用户名为空（例如通过默认构造后设置用户名），则加载其购物车          if (Customer \*cust = dynamic\_cast<Customer \*>(foundUser))          {              if (cust->shoppingCartItems.empty() && !cust->getUsername().empty())              {                             // 假设 shoppingCartItems 为空表示未加载                                            // cust->cartDirectoryPath 应该在构造时已设置                  cust->loadCartFromFile(); // 尝试加载购物车              }          }          return foundUser;      }      cout << "用户名或密码错误！" << endl;      return nullptr;  }  User \*User::login(const std::vector<User \*> &users, const std::string &username\_param, const std::string &password\_param)  {      User \*foundUser = findUser(users, username\_param);      if (foundUser && foundUser->getPassword() == password\_param)      {          return foundUser;      }      return nullptr;  }  void User::displayAllUsers(const std::vector<User \*> &users)  {      if (users.empty())      {          cout << "当前没有注册用户。" << endl;          return;      }      cout << "\n--- 所有用户信息 ---" << endl;      for (size\_t i = 0; i < users.size(); ++i)      {          users[i]->displayInfo(i + 1);      }      cout << "--------------------" << endl;  } |

#### Store.h

|  |
| --- |
| #ifndef STORE\_H  #define STORE\_H  #include <string>  #include <vector>  #include <iostream>  #include <fstream>  #include <sstream>  #include <iomanip>  #include <limits>  #include <map>  #include <filesystem>  // 前向声明 User  class User;  // --- Product 基类和派生类 ---  class Product  {  protected:      std::string name;      std::string description;      double originalPrice;      int quantity;      double discountRate;      std::string sellerUsername; // 添加商品所属商家  public:      Product(std::string name, std::string desc, double price, int qty, std::string seller = "")          : name(name), description(desc), originalPrice(price), quantity(qty),            discountRate(0.0), sellerUsername(seller) {}      virtual ~Product() = default;      virtual double getPrice() const { return originalPrice \* (1.0 - discountRate); }      virtual void display() const;      virtual std::string getType() const = 0;      virtual std::string getUserCategory() const { return ""; } // 默认返回空字符串      virtual void save(std::ofstream &ofs) const;      // Getters and Setters      std::string getName() const { return name; }      std::string getDescription() const { return description; }      double getOriginalPrice() const { return originalPrice; }      int getQuantity() const { return quantity; }      double getDiscountRate() const { return discountRate; }      std::string getSellerUsername() const { return sellerUsername; } // 获取商品所属商家      void setName(const std::string &newName) { name = newName; }      void setDescription(const std::string &newDesc) { description = newDesc; }      void setOriginalPrice(double newPrice)      {          if (newPrice >= 0)              originalPrice = newPrice;      }      void setQuantity(int newQuantity)      {          if (newQuantity >= 0)              quantity = newQuantity;      }      void setDiscountRate(double newRate)      {          if (newRate >= 0.0 && newRate <= 1.0)              discountRate = newRate;      }      void setSellerUsername(const std::string &seller) { sellerUsername = seller; } // 设置商品所属商家  };  class Book : public Product  {  private:      std::string author;      std::string isbn;  public:      Book(std::string name, std::string desc, double price, int qty,           std::string auth, std::string isbn\_num, std::string seller = "")          : Product(name, desc, price, qty, seller), author(auth), isbn(isbn\_num) {}      std::string getType() const override { return "Book"; }      std::string getUserCategory() const override { return "Book"; } // 固定类型为 "Book"      void display() const override;      void save(std::ofstream &ofs) const override;      // 新增的 getters      std::string getAuthor() const { return author; }      std::string getIsbn() const { return isbn; }  };  class Clothing : public Product  {  private:      std::string size;      std::string color;  public:      Clothing(std::string name, std::string desc, double price, int qty,               std::string sz, std::string clr, std::string seller = "")          : Product(name, desc, price, qty, seller), size(sz), color(clr) {}      std::string getType() const override { return "Clothing"; }      std::string getUserCategory() const override { return "Clothing"; } // 固定类型为 "Clothing"      void display() const override;      void save(std::ofstream &ofs) const override;      // 新增的 getters      std::string getSize() const { return size; }      std::string getColor() const { return color; }  };  class Food : public Product  {  private:      std::string expirationDate;  public:      Food(std::string name, std::string desc, double price, int qty,           std::string expDate, std::string seller = "")          : Product(name, desc, price, qty, seller), expirationDate(expDate) {}      std::string getType() const override { return "Food"; }      std::string getUserCategory() const override { return "Food"; } // 固定类型为 "Food"      void display() const override;      void save(std::ofstream &ofs) const override;      // 新增的 getters      std::string getExpirationDate() const { return expirationDate; }  };  // 自定义商品类  class GenericProduct : public Product  {  private:      std::string categoryTag; // 用户自定义的分类标签  public:      GenericProduct(std::string name, std::string desc, double price, int qty,                     std::string tag, std::string seller = "")          : Product(name, desc, price, qty, seller), categoryTag(tag) {}      std::string getType() const override { return "Generic"; }           // 固定类型为 "Generic"      std::string getUserCategory() const override { return categoryTag; } // 返回用户自定义标签      void display() const override;      void save(std::ofstream &ofs) const override;      // Getter for categoryTag      std::string getCategoryTag() const { return categoryTag; }      void setCategoryTag(const std::string &tag) { categoryTag = tag; }  };  // --- Store Class ---  class Store  {  private:      std::vector<Product \*> allProducts;                           // 存储所有商家的商品      std::map<std::string, std::vector<Product \*>> sellerProducts; // 每个商家的商品映射      std::string storeDirectory;                                   // 商品文件所在目录      // 辅助方法      std::string getSellerFilename(const std::string &username) const;      bool saveProductsForSeller(const std::string &sellerUsername);      bool ensureDirectoryExists(const std::string &path) const;  public:      // 构造函数接收目录名      Store(const std::string &directory);      ~Store();      Product \*findProductByName(const std::string &name, const std::string &sellerUsername = "");      // 文件操作      bool loadAllProducts();      bool loadSellerProducts(const std::string &sellerUsername);      bool saveAllProducts();      // 显示功能      void displayAllProducts() const;                                     // 显示所有商品      void displaySellerProducts(const std::string &sellerUsername) const; // 显示指定商家的商品      // 搜索功能      std::vector<Product \*> searchProductsByName(const std::string &searchTerm, const std::string &sellerUsername = "") const;      // 商家商品管理功能      bool createBook(User \*currentUser, const std::string &name, const std::string &desc,                      double price, int qty, const std::string &author, const std::string &isbn);      bool createClothing(User \*currentUser, const std::string &name, const std::string &desc,                          double price, int qty, const std::string &size, const std::string &color);      bool createFood(User \*currentUser, const std::string &name, const std::string &desc,                      double price, int qty, const std::string &expDate);      bool createGenericProduct(User \*currentUser, const std::string &name, const std::string &desc,                                double price, int qty, const std::string &categoryTag);      bool manageProductPrice(User \*currentUser, const std::string &productName, double newPrice);      bool manageProductQuantity(User \*currentUser, const std::string &productName, int newQuantity);      bool manageProductDiscount(User \*currentUser, const std::string &productName, double newDiscount);      bool applyCategoryDiscount(User \*currentUser, const std::string &category, double discount);      // 获取商品      const std::vector<Product \*> &getProducts() const { return allProducts; }      std::vector<Product \*> getSellerProducts(const std::string &sellerUsername) const;      // 获取商家商品的唯一分类      std::vector<std::string> getUniqueCategoriesForSeller(const std::string &sellerUsername) const;  };  #endif // STORE\_H |

#### Store.cpp

|  |
| --- |
| #include "store.h"  #include "../user/user.h"  #include <iostream>  #include <vector>  #include <string>  #include <fstream>  #include <sstream>  #include <iomanip>  #include <algorithm>  #include <filesystem>  #include <set>  #include <direct.h> // Windows 系统特定的目录操作  namespace fs = std::filesystem;  using namespace std;  // --- Product 方法实现 ---  void Product::display() const  {      std::cout << "  商户名称：" << sellerUsername << std::endl;      std::cout << "  商品名称: " << name << std::endl;      std::cout << "  描述: " << description << std::endl;      if (!getUserCategory().empty())      {          std::cout << "  分类标签: " << getUserCategory() << std::endl;      }      if (discountRate > 0)      {          std::cout << "  原价: ¥" << std::fixed << std::setprecision(2) << originalPrice << std::endl;          std::cout << "  折扣: " << (discountRate \* 100) << "%" << std::endl;          std::cout << "  现价: ¥" << std::fixed << std::setprecision(2) << getPrice() << std::endl;      }      else      {          std::cout << "  价格: ¥" << std::fixed << std::setprecision(2) << getPrice() << std::endl;      }      std::cout << "  库存: " << quantity << " 件" << std::endl;      if (!sellerUsername.empty())      {          std::cout << "  商家: " << sellerUsername << std::endl;      }  }  void Product::save(std::ofstream &ofs) const  {      ofs << getType() << "," << name << "," << description << ","          << originalPrice << "," << quantity << "," << discountRate << "," << sellerUsername;  }  void Book::display() const // 复用父类  {      Product::display();      std::cout << "  作者: " << author << std::endl;      std::cout << "  ISBN: " << isbn << std::endl;  }  void Book::save(std::ofstream &ofs) const  {      Product::save(ofs);      ofs << "," << author << "," << isbn;  }  void Clothing::display() const  {      Product::display();      std::cout << "  尺寸: " << size << std::endl;      std::cout << "  颜色: " << color << std::endl;  }  void Clothing::save(std::ofstream &ofs) const  {      Product::save(ofs);      ofs << "," << size << "," << color;  }  void Food::display() const  {      Product::display();      std::cout << "  保质期限: " << expirationDate << std::endl;  }  void Food::save(std::ofstream &ofs) const  {      Product::save(ofs);      ofs << "," << expirationDate;  }  void GenericProduct::display() const  {      Product::display();  }  void GenericProduct::save(std::ofstream &ofs) const  {      Product::save(ofs);      ofs << "," << categoryTag;  }  // --- Store 类实现 ---  // 构造函数  Store::Store(const string &directory) : storeDirectory(directory)  {      // 确保商家目录存在      string sellersDir = storeDirectory + "/sellers";      ensureDirectoryExists(sellersDir);      // 加载所有商品数据      if (!loadAllProducts())      {          cerr << "警告: 无法加载商店数据。商店可能为空或文件丢失/损坏。" << endl;      }      else      {          cout << "已从商店加载 " << allProducts.size() << " 件商品。" << endl;      }  }  // 析构函数  Store::~Store()  {      for (Product \*p : allProducts)      {          delete p;      }      allProducts.clear();      sellerProducts.clear(); // 清除映射  }  // 确保目录存在  bool Store::ensureDirectoryExists(const string &path) const  {      try      {          if (!fs::exists(path))          {              fs::create\_directories(path);              cout << "创建目录: " << path << endl;          }          return true;      }      catch (const fs::filesystem\_error &e)      {          cerr << "创建目录失败 " << path << ": " << e.what() << endl;          // 尝试使用传统方法创建目录          if (\_mkdir(path.c\_str()) == 0)          {              cout << "使用传统方法创建目录: " << path << endl;              return true;          }          return false;      }      catch (...)      {          cerr << "创建目录时发生未知错误: " << path << endl;          return false;      }  }  // 获取商家文件名  string Store::getSellerFilename(const string &username) const  {      return storeDirectory + "/sellers/" + username + ".txt";  }  // 加载所有商品  bool Store::loadAllProducts()  {      // 清空现有商品      for (Product \*p : allProducts)      {          delete p;      }      allProducts.clear();      sellerProducts.clear();      // 获取sellers目录下的所有文件      string sellersDir = storeDirectory + "/sellers";      try      {          // 如果目录不存在，创建目录并返回成功（表示商店为空）          if (!fs::exists(sellersDir))          {              ensureDirectoryExists(sellersDir);              return true;          }          // 遍历目录中的所有文件          for (const auto &entry : fs::directory\_iterator(sellersDir))          {              if (entry.is\_regular\_file())              {                  string filename = entry.path().string();                  string sellerUsername = entry.path().stem().string(); // 获取不带扩展名的文件名                  loadSellerProducts(sellerUsername);              }          }          return true;      }      catch (const fs::filesystem\_error &e)      {          cerr << "读取商家目录失败: " << e.what() << endl;          return false;      }      catch (...)      {          cerr << "读取商家数据时发生未知错误" << endl;          return false;      }  }  // 加载指定商家的商品  bool Store::loadSellerProducts(const string &sellerUsername)  {      string filename = getSellerFilename(sellerUsername);      ifstream file(filename);      if (!file.is\_open())      {          // 对于新商家，文件不存在是正常的          return true;      }      vector<Product \*> sellerProductsList;      string line;      while (getline(file, line))      {          stringstream ss(line);          string segment;          vector<string> seglist;          while (getline(ss, segment, ','))          {              seglist.push\_back(segment);          }          // 格式检查: 至少需要7个字段 (类型,名称,描述,价格,数量,折扣,商家)          if (seglist.size() < 7)          {              cerr << "无效的产品数据行: " << line << endl;              continue;          }          try          {              string type = seglist[0];              string name = seglist[1];              string desc = seglist[2];              double price = stod(seglist[3]);              int qty = stoi(seglist[4]);              double discount = stod(seglist[5]);              string seller = seglist[6]; // 第7个字段是商家名              // 确认商家名称匹配              if (seller != sellerUsername)              {                  cerr << "警告: 文件 " << filename << " 中的商品所属商家 \""                       << seller << "\" 与预期的 \"" << sellerUsername << "\" 不符" << endl;                  seller = sellerUsername; // 强制使用文件名指示的商家              }              Product \*newProduct = nullptr;              if (type == "Book" && seglist.size() >= 9)              {                  // Book需要额外的author和isbn字段                  newProduct = new Book(name, desc, price, qty, seglist[7], seglist[8], seller);              }              else if (type == "Clothing" && seglist.size() >= 9)              {                  // Clothing需要额外的size和color字段                  newProduct = new Clothing(name, desc, price, qty, seglist[7], seglist[8], seller);              }              else if (type == "Food" && seglist.size() >= 8)              {                  // Food需要额外的expirationDate字段                  newProduct = new Food(name, desc, price, qty, seglist[7], seller);              }              else if (type == "Generic" && seglist.size() >= 8)              {                  // GenericProduct需要额外的categoryTag字段                  newProduct = new GenericProduct(name, desc, price, qty, seglist[7], seller);              }              else              {                  cerr << "未知商品类型或缺少必要字段: " << line << endl;                  continue;              }              if (newProduct)              {                  newProduct->setDiscountRate(discount);                  allProducts.push\_back(newProduct);                  sellerProductsList.push\_back(newProduct);              }          }          catch (const exception &e)          {              cerr << "解析商品数据时出错: " << e.what() << " 行: " << line << endl;          }      }      file.close();      // 更新商家商品映射      sellerProducts[sellerUsername] = sellerProductsList;      cout << "已加载商家 \"" << sellerUsername << "\" 的 "           << sellerProductsList.size() << " 件商品" << endl;      return true;  }  // 保存所有商品  bool Store::saveAllProducts()  {      // 对于每个商家，保存其商品      set<string> processedSellers; // 用于跟踪已处理的商家      for (const auto &product : allProducts)      {          string seller = product->getSellerUsername();          if (!seller.empty() && processedSellers.find(seller) == processedSellers.end())          {              saveProductsForSeller(seller);              processedSellers.insert(seller);          }      }      return true;  }  // 保存指定商家的商品  bool Store::saveProductsForSeller(const string &sellerUsername)  {      if (sellerUsername.empty())      {          cerr << "错误: 尝试保存无效商家的商品" << endl;          return false;      }      vector<Product \*> products;      // 获取该商家的所有商品      for (Product \*p : allProducts)      {          if (p->getSellerUsername() == sellerUsername)          {              products.push\_back(p);          }      }      string filename = getSellerFilename(sellerUsername);      ofstream file(filename);      if (!file.is\_open())      {          cerr << "错误: 无法打开文件保存商品: " << filename << endl;          return false;      }      for (const Product \*p : products)      {          p->save(file);          file << endl;      }      file.close();      // cout << "商家 \"" << sellerUsername << "\" 的 " << products.size()      //      << " 件商品已保存至 " << filename << endl;      return true;  }  // 查找商品  Product \*Store::findProductByName(const string &name, const string &sellerUsername)  {      if (!sellerUsername.empty())      {          // 只在指定商家的商品中查找          auto it = sellerProducts.find(sellerUsername);          if (it != sellerProducts.end())          {              for (Product \*p : it->second)              {                  if (p->getName() == name)                  {                      return p;                  }              }          }          return nullptr;      }      else      {          // 在所有商品中查找          for (Product \*p : allProducts)          {              if (p->getName() == name)              {                  return p;              }          }          return nullptr;      }  }  // 显示所有商品  void Store::displayAllProducts() const  {      if (allProducts.empty())      {          cout << "\n--- 商品列表为空 ---" << endl;          return;      }      cout << "\n--- 所有商品列表 ---" << endl;      int count = 0;      for (const Product \*p : allProducts)      {          cout << "--------------------------" << endl;          p->display();          count++;      }      cout << "--------------------------" << endl;      cout << "共显示 " << count << " 件商品" << endl;  }  // 显示指定商家的商品  void Store::displaySellerProducts(const string &sellerUsername) const  {      auto it = sellerProducts.find(sellerUsername);      if (it == sellerProducts.end() || it->second.empty())      {          cout << "\n--- 商家 \"" << sellerUsername << "\" 没有商品 ---" << endl;          return;      }      cout << "\n--- 商家 \"" << sellerUsername << "\" 的商品列表 ---" << endl;      int count = 0;      for (const Product \*p : it->second)      {          cout << "--------------------------" << endl;          p->display();          count++;      }      cout << "--------------------------" << endl;      cout << "共显示 " << count << " 件商品" << endl;  }  // 按名称搜索商品  vector<Product \*> Store::searchProductsByName(const string &searchTerm, const string &sellerUsername) const  {      vector<Product \*> results;      const vector<Product \*> &productsToSearch = sellerUsername.empty() ? allProducts : (sellerProducts.count(sellerUsername) ? sellerProducts.at(sellerUsername) : vector<Product \*>());      for (Product \*p : productsToSearch)      {          // 简单的包含搜索 (不区分大小写)          string productName = p->getName();          string searchTermLower = searchTerm;          // 将两个字符串转为小写进行比较          transform(productName.begin(), productName.end(), productName.begin(), ::tolower);          transform(searchTermLower.begin(), searchTermLower.end(), searchTermLower.begin(), ::tolower);          if (productName.find(searchTermLower) != string::npos)          {              results.push\_back(p);          }      }      return results;  }  // 获取商家的商品  vector<Product \*> Store::getSellerProducts(const string &sellerUsername) const  {      auto it = sellerProducts.find(sellerUsername);      return (it != sellerProducts.end()) ? it->second : vector<Product \*>();  }  // 创建图书  bool Store::createBook(User \*currentUser, const string &name, const string &desc,                         double price, int qty, const string &author, const string &isbn)  {      if (!currentUser)      {          cerr << "错误: 用户未登录" << endl;          return false;      }      if (currentUser->getUserType() != "seller")      {          cerr << "错误: 只有商家可以添加商品" << endl;          return false;      }      string sellerUsername = currentUser->getUsername();      // 检查该商家是否已有同名商品      if (findProductByName(name, sellerUsername) != nullptr)      {          cerr << "错误: 您已有名为 \"" << name << "\" 的商品" << endl;          return false;      }      if (price < 0 || qty < 0)      {          cerr << "错误: 价格和数量不能为负数" << endl;          return false;      }      try      {          Book \*newBook = new Book(name, desc, price, qty, author, isbn, sellerUsername);          allProducts.push\_back(newBook);          // 更新商家商品映射          sellerProducts[sellerUsername].push\_back(newBook);          // 保存商家的商品          if (saveProductsForSeller(sellerUsername))          {              cout << "商品 \"" << name << "\" 添加成功！" << endl;              return true;          }          else          {              cout << "商品 \"" << name << "\" 添加失败！" << endl;              return false;          }      }      catch (const bad\_alloc &e)      {          cerr << "错误: 无法分配内存: " << e.what() << endl;          return false;      }      catch (const exception &e)      {          cerr << "错误: " << e.what() << endl;          return false;      }      catch (...)      {          cerr << "错误: 添加商品时发生未知错误" << endl;          return false;      }  }  // 创建服装  bool Store::createClothing(User \*currentUser, const string &name, const string &desc,                             double price, int qty, const string &size, const string &color)  {      if (!currentUser)      {          cerr << "错误: 用户未登录" << endl;          return false;      }      if (currentUser->getUserType() != "seller")      {          cerr << "错误: 只有商家可以添加商品" << endl;          return false;      }      string sellerUsername = currentUser->getUsername();      // 检查该商家是否已有同名商品      if (findProductByName(name, sellerUsername) != nullptr)      {          cerr << "错误: 您已有名为 \"" << name << "\" 的商品" << endl;          return false;      }      if (price < 0 || qty < 0)      {          cerr << "错误: 价格和数量不能为负数" << endl;          return false;      }      try      {          Clothing \*newClothing = new Clothing(name, desc, price, qty, size, color, sellerUsername);          allProducts.push\_back(newClothing);          // 更新商家商品映射          sellerProducts[sellerUsername].push\_back(newClothing);          // 保存商家的商品          if (saveProductsForSeller(sellerUsername))          {              cout << "商品 \"" << name << "\" 添加成功！" << endl;              return true;          }          else          {              cout << "商品 \"" << name << "\" 添加失败！" << endl;              return false;          }      }      catch (const bad\_alloc &e)      {          cerr << "错误: 无法分配内存: " << e.what() << endl;          return false;      }      catch (const exception &e)      {          cerr << "错误: " << e.what() << endl;          return false;      }      catch (...)      {          cerr << "错误: 添加商品时发生未知错误" << endl;          return false;      }  }  // 创建食品  bool Store::createFood(User \*currentUser, const string &name, const string &desc,                         double price, int qty, const string &expDate)  {      if (!currentUser)      {          cerr << "错误: 用户未登录" << endl;          return false;      }      if (currentUser->getUserType() != "seller")      {          cerr << "错误: 只有商家可以添加商品" << endl;          return false;      }      string sellerUsername = currentUser->getUsername();      // 检查该商家是否已有同名商品      if (findProductByName(name, sellerUsername) != nullptr)      {          cerr << "错误: 您已有名为 \"" << name << "\" 的商品" << endl;          return false;      }      if (price < 0 || qty < 0)      {          cerr << "错误: 价格和数量不能为负数" << endl;          return false;      }      try      {          Food \*newFood = new Food(name, desc, price, qty, expDate, sellerUsername);          allProducts.push\_back(newFood);          // 更新商家商品映射          sellerProducts[sellerUsername].push\_back(newFood);          // 保存商家的商品          if (saveProductsForSeller(sellerUsername))          {              cout << "商品 \"" << name << "\" 添加成功！" << endl;              return true;          }          else          {              cout << "商品 \"" << name << "\" 添加失败！" << endl;              return false;          }      }      catch (const bad\_alloc &e)      {          cerr << "错误: 无法分配内存: " << e.what() << endl;          return false;      }      catch (const exception &e)      {          cerr << "错误: " << e.what() << endl;          return false;      }      catch (...)      {          cerr << "错误: 添加商品时发生未知错误" << endl;          return false;      }  }  bool Store::createGenericProduct(User \*currentUser, const string &name, const string &desc,                                   double price, int qty, const string &categoryTag)  {      if (!currentUser)      {          cerr << "错误: 用户未登录" << endl;          return false;      }      if (currentUser->getUserType() != "seller")      {          cerr << "错误: 只有商家可以添加商品" << endl;          return false;      }      string sellerUsername = currentUser->getUsername();      if (findProductByName(name, sellerUsername) != nullptr)      {          cerr << "错误: 您已有名为 \"" << name << "\" 的商品" << endl;          return false;      }      if (price < 0 || qty < 0)      {          cerr << "错误: 价格和数量不能为负数" << endl;          return false;      }      if (categoryTag.empty())      {          cerr << "错误: 通用商品必须指定分类标签" << endl;          return false;      }      try      {          GenericProduct \*newGenericProduct = new GenericProduct(name, desc, price, qty, categoryTag, sellerUsername);          allProducts.push\_back(newGenericProduct);          sellerProducts[sellerUsername].push\_back(newGenericProduct);          if (saveProductsForSeller(sellerUsername))          {              cout << "通用商品 \"" << name << "\" (分类: " << categoryTag << ") 添加成功！" << endl;              return true;          }          else          {              // 如果保存失败，应该从内存中移除，以保持一致性              allProducts.pop\_back();              sellerProducts[sellerUsername].pop\_back();              delete newGenericProduct;              cout << "通用商品 \"" << name << "\" 添加失败（保存错误）！" << endl;              return false;          }      }      catch (const bad\_alloc &e)      {          cerr << "错误: 无法分配内存: " << e.what() << endl;          return false;      }      catch (const exception &e)      {          cerr << "错误: " << e.what() << endl;          return false;      }      catch (...)      {          cerr << "错误: 添加通用商品时发生未知错误" << endl;          return false;      }  }  // 修改商品价格  bool Store::manageProductPrice(User \*currentUser, const string &productName, double newPrice)  {      if (!currentUser)      {          cerr << "错误: 用户未登录" << endl;          return false;      }      if (currentUser->getUserType() != "seller")      {          cerr << "错误: 只有商家可以修改商品价格" << endl;          return false;      }      string sellerUsername = currentUser->getUsername();      Product \*product = findProductByName(productName, sellerUsername);      if (!product)      {          cerr << "错误: 未找到您的商品 \"" << productName << "\"" << endl;          return false;      }      if (newPrice < 0)      {          cerr << "错误: 价格不能为负数" << endl;          return false;      }      product->setOriginalPrice(newPrice);      // 保存商家的商品      if (saveProductsForSeller(sellerUsername))      {          cout << "价格保存成功！" << endl;          return true;      }      else      {          cout << "价格保存失败！" << endl;          return false;      }  }  // 修改商品库存  bool Store::manageProductQuantity(User \*currentUser, const string &productName, int newQuantity)  {      if (!currentUser)      {          cerr << "错误: 用户未登录" << endl;          return false;      }      if (currentUser->getUserType() != "seller")      {          cerr << "错误: 只有商家可以修改商品库存" << endl;          return false;      }      string sellerUsername = currentUser->getUsername();      Product \*product = findProductByName(productName, sellerUsername);      if (!product)      {          cerr << "错误: 未找到您的商品 \"" << productName << "\"" << endl;          return false;      }      if (newQuantity < 0)      {          cerr << "错误: 库存不能为负数" << endl;          return false;      }      product->setQuantity(newQuantity);      if (saveProductsForSeller(sellerUsername))      {          cout << "库存修改成功！" << endl;          return true;      }      else      {          cout << "库存修改失败！" << endl;          return false;      }  }  // 修改商品折扣  bool Store::manageProductDiscount(User \*currentUser, const string &productName, double newDiscount)  {      if (!currentUser)      {          cerr << "错误: 用户未登录" << endl;          return false;      }      if (currentUser->getUserType() != "seller")      {          cerr << "错误: 只有商家可以修改商品折扣" << endl;          return false;      }      string sellerUsername = currentUser->getUsername();      Product \*product = findProductByName(productName, sellerUsername);      if (!product)      {          cerr << "错误: 未找到您的商品 \"" << productName << "\"" << endl;          return false;      }      if (newDiscount < 0 || newDiscount > 100)      {          cerr << "错误: 折扣必须在0到100之间" << endl;          return false;      }      product->setDiscountRate(newDiscount);      return saveProductsForSeller(sellerUsername);  }  std::vector<std::string> Store::getUniqueCategoriesForSeller(const std::string &sellerUsername) const  {      std::set<std::string> uniqueCategories;      auto it = sellerProducts.find(sellerUsername);      if (it != sellerProducts.end())      {          for (const Product \*p : it->second)          {              if (p)              {                                                // Ensure product pointer is not null                  std::string category = p->getUserCategory(); // This gets "Book", "Clothing", "Food", or custom tag                  if (!category.empty())                  { // Ensure not to add empty strings                      uniqueCategories.insert(category);                  }              }          }      }      return std::vector<std::string>(uniqueCategories.begin(), uniqueCategories.end());  }  // 应用分类折扣  bool Store::applyCategoryDiscount(User \*currentUser, const string &category, double discount)  {      if (!currentUser)      {          cerr << "错误: 用户未登录" << endl;          return false;      }      if (currentUser->getUserType() != "seller")      {          cerr << "错误: 只有商家可以应用折扣" << endl;          return false;      }      if (discount < 0.0 || discount > 1.0)      {          cerr << "错误: 折扣必须在0到100之间" << endl;          return false;      }      string sellerUsername = currentUser->getUsername();      bool changed = false;      // 只应用折扣到该商家的商品      auto it = sellerProducts.find(sellerUsername);      if (it != sellerProducts.end())      {          for (Product \*p : it->second)          {              // 使用 getUserCategory() 进行比较              if (p && p->getUserCategory() == category)              {                  p->setDiscountRate(discount);                  changed = true;              }          }      }      if (changed)      {          cout << "已对分类 \"" << category << "\" 下的商品应用 " << (discount \* 100) << "% 折扣。" << endl;          return saveProductsForSeller(sellerUsername);      }      cerr << "错误: 未找到分类为 \"" << category << "\" 的商品" << endl;      return false;  } |

#### Page.h

|  |
| --- |
| #ifndef PAGE\_H  #define PAGE\_H  #include "../store/store.h" // 包含 Store 类定义  #include <vector>  #include <string> // 需要包含 string  // 前向声明 User 类，因为 Page 方法使用了 User\*  // 避免在头文件中包含 user.h，减少编译依赖  class User;  // --- Page 类定义 ---  // 用于管理不同状态下的菜单显示  class Page  {  protected:      // 私有静态辅助方法：清除输入缓冲区 (Page 类内部使用)      // 定义移到 page.cpp      static void clearInputBuffer();  public:      // --- 成员方法：菜单显示 ---      // 主菜单 (未登录)      int pagemain(std::vector<User \*> &users, User \*&currentUser);      // 消费者菜单 (登录后)      int pagemainCustomer(User \*currentUser, std::vector<User \*> &users);      // 商家菜单 (登录后)      int pagemainSeller(User \*currentUser, std::vector<User \*> &users);      // 管理员菜单 (如果需要可以添加)      int pagemainAdmin(User \*currentUser, std::vector<User \*> &users);      virtual int pagestore(User \*currentUser, std::vector<User \*> &users, Store &store) = 0;  };  // --- Page 派生类 ---  class Customer\_page : public Page  {  public:      // 实现商城页面 (确保有 Store& store 参数)      int pagestore(User \*currentUser, std::vector<User \*> &users, Store &store) override;      // 可以覆盖基类的账户菜单方法，如果需要特定行为  };  class Seller\_page : public Page  {  public:      // 实现商城页面 (确保有 Store& store 参数)      int pagestore(User \*currentUser, std::vector<User \*> &users, Store &store) override;      // 可以覆盖基类的账户菜单方法，如果需要特定行为  };  // //应该暂不需要  // class Admin\_page : public Page  // {  //     int pagestore(User \*currentUser, std::vector<User \*> &users);  // };  #endif // PAGE\_H |

#### Page.cpp

|  |
| --- |
| #include "page.h"  #include "../user/user.h"  #include "../store/store.h" // 需要 Store 定义  #include "../ordermanager/ordermanager.h"  #include <iostream>  #include <cstdio> // for printf if used  #include <limits>  #include <vector>  #include <string>  #include <iomanip>  using namespace std;  extern const string USER\_FILE;  // STORE\_FILE is managed by the Store object passed in  // --- 基类 Page 静态方法 ---  void Page::clearInputBuffer()  {      cin.clear();      cin.ignore(numeric\_limits<streamsize>::max(), '\n');  }  // --- 基类 Page 菜单实现 ---  // 这些函数现在只处理用户账户菜单，并在选择商城时返回特定值 3  int Page::pagemain(vector<User \*> &users, User \*&currentUser)  {      int choice = 0;      while (true)      {          printf("\n\n\n--- 主菜单 ---\n");          cout << "1. 注册新用户" << endl;          cout << "2. 用户登录" << endl;          cout << "3. 进入商城 (游客)" << endl; // 返回 3 表示进入商城          cout << "4. 退出程序" << endl;          cout << "请选择一个选项: ";          if (!(cin >> choice))          {              cout << "无效输入。" << endl;              clearInputBuffer();              continue;          }          clearInputBuffer();          switch (choice)          {          case 1:              currentUser = User::registerUser(users);              if (currentUser)              {                  User::saveUsersToFile(users, USER\_FILE);                  // 注册成功后可以选择留在主菜单或直接进入用户菜单                  // return 1; // 如果想直接进入用户菜单              }              return 1;              break; // 留在主菜单          case 2:              currentUser = User::userLogin(users);              if (currentUser)                  return 1; // 登录成功，返回 1 给 main 处理              break;        // 登录失败，留在主菜单          case 3:              return 3; // 返回 3，通知 main 调用 pagestore          case 4:              cout << "退出程序..." << endl;              return 0; // 返回 0 给 main 退出循环          default:              cout << "无效选项。" << endl;          }      }  }  int Page::pagemainCustomer(User \*currentUser, vector<User \*> &users)  {      int choice = 0;      while (true)      {          printf("\n\n\n--- 消费者菜单 (%s) ---\n", currentUser->getUsername().c\_str());          cout << "1. 查看个人信息" << endl;          cout << "2. 修改密码" << endl;          cout << "3. 查询余额" << endl;          cout << "4. 充值" << endl;          cout << "5. 进入商城" << endl; // 返回 3 表示进入商城          cout << "6. 退出登录" << endl;          cout << "请选择一个选项: ";          if (!(cin >> choice))          {              cout << "无效输入。" << endl;              clearInputBuffer();              continue;          }          clearInputBuffer();          switch (choice)          {          case 1:              currentUser->displayInfo();              break;          case 2:          {              string oldP, newP;              cout << "请输入原密码: ";              getline(cin >> ws, oldP);              cout << "请输入新密码: ";              getline(cin >> ws, newP);              if (currentUser->changePassword(oldP, newP))              {                  User::saveUsersToFile(users, USER\_FILE);                  cout << "密码修改成功。" << endl;              }              else              {                  cout << "密码修改失败。" << endl;              }          }          break;          case 3:          {              cout << fixed << setprecision(2); // 设置输出格式              cout << "当前余额: " << currentUser->checkBalance() << endl;              break;          }          case 4:          {              double amt;              cout << "请输入充值金额: ";              if (!(cin >> amt) || amt <= 0)              { // 检查无效输入和非正数                  cout << "无效金额。" << endl;                  clearInputBuffer();                  break;              }              clearInputBuffer();              if (currentUser->deposit(amt))              {                  User::saveUsersToFile(users, USER\_FILE);                  cout << "充值成功。当前余额: " << currentUser->checkBalance() << endl;              }              else              {                  // deposit 内部应该已经打印了错误信息              }          }          break;          case 5:              return 3; // 返回 3，通知 main 调用 pagestore          case 6:              cout << "退出登录..." << endl;              return 2; // 返回 2 给 main 处理退出登录          default:              cout << "无效选项。" << endl;          }      }  }  int Page::pagemainSeller(User \*currentUser, vector<User \*> &users)  {      int choice = 0;      while (true)      {          printf("\n\n\n--- 商家菜单 (%s) ---\n", currentUser->getUsername().c\_str());          cout << "1. 查看商户信息" << endl;          cout << "2. 修改密码" << endl;          cout << "3. 进入商城 (管理)" << endl; // 返回 3 表示进入商城          cout << "4. 查看收入" << endl;          cout << "0. 退出登录" << endl;          cout << "请选择一个选项: ";          if (!(cin >> choice))          {              cout << "无效输入。" << endl;              clearInputBuffer();              continue;          }          clearInputBuffer();          switch (choice)          {          case 1:              currentUser->displayInfo();              break;          case 2:          {              string oldP, newP;              cout << "请输入原密码: ";              getline(cin >> ws, oldP);              cout << "请输入新密码: ";              getline(cin >> ws, newP);              if (currentUser->changePassword(oldP, newP))              {                  User::saveUsersToFile(users, USER\_FILE);                  cout << "密码修改成功。" << endl;              }              else              {                  cout << "密码修改失败。" << endl;              }          }          break;          case 3:              return 3; // 返回 3，通知 main 调用 pagestore          case 4:          {              double income = currentUser->checkBalance(); // 假设收入就是余额              cout << fixed << setprecision(2);            // 设置输出格式              cout << "当前收入: " << income << endl;              break;          }          case 0:              cout << "退出登录..." << endl;              return 2; // 返回 2 给 main 处理退出登录          default:              cout << "无效选项。" << endl;          }      }  }  int Page::pagemainAdmin(User \*currentUser, vector<User \*> &users)  {      // 管理员菜单实现      int choice = 0;      while (true)      {          printf("\n\n\n--- 管理员菜单 (%s) ---\n", currentUser->getUsername().c\_str());          cout << "1. 查看所有用户信息" << endl;          // 可以添加其他管理员功能          cout << "2. 退出登录" << endl;          cout << "请选择一个选项: ";          if (!(cin >> choice))          {              cout << "无效输入。" << endl;              clearInputBuffer();              continue;          }          clearInputBuffer();          switch (choice)          {          case 1:              User::displayAllUsers(users); // 假设 User 类有这个静态方法              break;          case 2:              cout << "退出登录..." << endl;              return 2; // 返回 2 给 main 处理退出登录          default:              cout << "无效选项。" << endl;          }      }  }  // --- Customer\_page 商城实现 ---  int Customer\_page::pagestore(User \*currentUser, std::vector<User \*> &users, Store &store)  {      int choice = -1;      Customer \*customer = dynamic\_cast<Customer \*>(currentUser); // 用于调用购物车方法      string username\_display = "游客";      if (currentUser)      {          username\_display = currentUser->getUsername();      }      do      {          cout << "\n--- 商城 (用户: " << username\_display << ") ---" << endl;          cout << "1. 显示所有商品" << endl;          cout << "2. 按名称搜索商品" << endl;          cout << "3. 直接购买商品" << endl; // 保持不变          cout << "4. 加入购物车" << endl;          cout << "5. 查看我的购物车" << endl;          cout << "0. 退出商城" << endl;          cout << "请选择: ";          if (!(cin >> choice))          {              cout << "无效输入。" << endl;              Page::clearInputBuffer();              choice = -1;              continue;          }          Page::clearInputBuffer();          switch (choice)          {          case 1:              store.displayAllProducts();              break;          case 2:          {              string searchTerm;              cout << "输入搜索名称: ";              getline(cin >> ws, searchTerm);              vector<Product \*> results = store.searchProductsByName(searchTerm);              if (results.empty())              {                  cout << "未找到匹配商品。" << endl;              }              else              {                  cout << "\n--- 搜索结果 ---" << endl;                  for (const auto \*p : results)                  {                      cout << "-----------------" << endl;                      p->display();                  }                  cout << "-----------------" << endl;              }              break;          }          case 3: // 直接购买商品逻辑 - 现在通过订单系统处理          {              if (!customer)              { // 必须是登录的 Customer                  cout << "请先以消费者身份登录后再购买商品！" << endl;                  break;              }              // --- 开始直接购买逻辑 ---              string productName\_direct;              int quantity\_direct;              cout << "输入要购买的商品名称: ";              getline(cin >> ws, productName\_direct);              Product \*product\_direct = store.findProductByName(productName\_direct);              if (!product\_direct)              {                  cout << "未找到商品 \"" << productName\_direct << "\"！" << endl;                  break;              }              cout << "输入购买数量: ";              if (!(cin >> quantity\_direct) || quantity\_direct <= 0)              {                  cout << "无效购买数量！" << endl;                  Page::clearInputBuffer();                  break;              }              Page::clearInputBuffer();              double totalCost\_direct = product\_direct->getPrice() \* quantity\_direct;              if (product\_direct->getQuantity() < quantity\_direct)              {                  cout << "库存不足！" << endl;                  break;              }              if (customer->checkBalance() < totalCost\_direct)              {                  cout << "余额不足！" << endl;                  break;              }              // 创建订单对象              Order currentOrder(customer->getUsername());              currentOrder.setStatus("PENDING\_VALIDATION");              // 添加商品到订单              if (!currentOrder.addItemFromProduct(\*product\_direct, quantity\_direct))              {                  cout << "添加商品到订单失败！" << endl;                  break;              }              // 计算订单总额              currentOrder.calculateTotalAmount();              // 显示订单信息并请求确认              currentOrder.setStatus("PENDING\_CONFIRMATION");              currentOrder.displaySummary();              cout << "您的当前余额: ¥" << std::fixed << std::setprecision(2) << customer->checkBalance() << endl;              cout << "支付后余额将为: ¥" << (customer->checkBalance() - currentOrder.getTotalAmount()) << endl;              cout << "--------------------" << endl;              char confirmChoice;              cout << "确认支付并完成订单吗? (y/n): ";              cin >> confirmChoice;              clearInputBuffer();              if (tolower(confirmChoice) != 'y')              {                  currentOrder.setStatus("CANCELLED\_BY\_USER");                  cout << "订单已取消。" << endl;                  break;              }              // 提交订单到处理队列              cout << "正在处理订单，请稍候..." << endl;              extern OrderManager g\_orderManager;              auto submittedOrder = g\_orderManager.submitOrderRequest(currentOrder);              cout << "订单已成功提交到处理队列，订单ID: " << submittedOrder->getOrderId() << endl;              cout << "正在等待订单处理，请稍候..." << endl;              // 等待订单处理完成              int waitCounter = 0;              const int MAX\_WAIT\_CYCLES = 100; // 最多等待10秒              while (!submittedOrder->getProcessed() && waitCounter < MAX\_WAIT\_CYCLES)              {                  // 短暂休眠，避免过度消耗CPU                  std::this\_thread::sleep\_for(std::chrono::milliseconds(100));                  waitCounter++;                  // 每秒显示等待状态                  if (waitCounter % 10 == 0)                  {                      cout << "已等待 " << (waitCounter / 10) << " 秒..." << endl;                  }              }              // 检查是否超时              if (waitCounter >= MAX\_WAIT\_CYCLES)              {                  cout << "等待订单处理超时！订单可能仍在处理中。" << endl;                  cout << "您可以稍后查询订单状态。" << endl;              }              else              {                  cout << "订单已处理完成! 状态: " << submittedOrder->getStatus() << endl;                  if (submittedOrder->getStatus() == "COMPLETED" ||                      submittedOrder->getStatus() == "COMPLETED\_WITH\_PAYMENT\_ISSUES")                  {                      cout << "购买成功！已购买 " << quantity\_direct << " 件 \"" << productName\_direct << "\"" << endl;                  }                  else if (submittedOrder->getStatus().find("FAILED") != std::string::npos)                  {                      cout << "订单处理失败! 原因: ";                      if (submittedOrder->getStatus() == "FAILED\_PAYMENT\_ERROR")                          cout << "支付处理时出错" << endl;                      else if (submittedOrder->getStatus() == "FAILED\_INSUFFICIENT\_FUNDS")                          cout << "账户余额不足" << endl;                      else if (submittedOrder->getStatus() == "FAILED\_PRODUCT\_NOT\_FOUND")                          cout << "商品不存在或已下架" << endl;                      else if (submittedOrder->getStatus() == "FAILED\_INSUFFICIENT\_STOCK")                          cout << "商品库存不足" << endl;                      else                          cout << "未知原因，请联系客服" << endl;                  }              }              break;          }          case 4: // 加入购物车          {              if (!customer)              {                  cout << "请先以消费者身份登录后再使用购物车功能！" << endl;                  break;              }              string productName\_cart;              int quantity\_to\_add;              cout << "输入要加入购物车的商品名称: ";              getline(cin >> ws, productName\_cart);              Product \*product\_to\_add = store.findProductByName(productName\_cart);              if (!product\_to\_add)              {                  cout << "未找到商品 \"" << productName\_cart << "\"！" << endl;                  break;              }              cout << "输入数量: ";              if (!(cin >> quantity\_to\_add) || quantity\_to\_add <= 0)              {                  cout << "无效的数量！" << endl;                  Page::clearInputBuffer();                  break;              }              Page::clearInputBuffer();              customer->addToCart(\*product\_to\_add, quantity\_to\_add);              break;          }          case 5: // 查看我的购物车          {              if (!customer)              {                  cout << "请先以消费者身份登录后再查看购物车！" << endl;                  break;              }              customer->viewCart(store, users); // viewCart 现在是 Customer 的成员函数              break;          }          case 0:              cout << "正在退出商城..." << endl;              break;          default:              cout << "无效选项。" << endl;          }      } while (choice != 0);      return 1;  }  // 只替换Seller\_page::pagestore方法，保留文件其他部分不变  // --- Seller\_page 商城实现 ---  int Seller\_page::pagestore(User \*currentUser, std::vector<User \*> &users, Store &store)  {      // 确保当前用户是商家      if (!currentUser || currentUser->getUserType() != "seller")      {          cout << "错误：只有商家才能访问商家管理页面。" << endl;          return 1; // 直接退出      }      string sellerUsername = currentUser->getUsername();      int choice = -1;      do      {          cout << "\n--- 商城管理 (商家: " << sellerUsername << ") ---" << endl;          cout << "1. 显示我的商品" << endl;          cout << "2. 按名称搜索我的商品" << endl;          cout << "3. 添加新商品" << endl;          cout << "4. 修改商品价格" << endl;          cout << "5. 修改商品库存" << endl;          cout << "6. 修改商品折扣" << endl;          cout << "7. 应用分类折扣" << endl;          // cout << "8. 查看余额" << endl;          cout << "0. 退出商城管理" << endl;          cout << "请选择: ";          if (!(cin >> choice))          {              cout << "无效输入。" << endl;              clearInputBuffer();              choice = -1;              continue;          }          clearInputBuffer();          switch (choice)          {          case 1:              store.displaySellerProducts(sellerUsername);              break;          case 2:          {              string searchTerm;              cout << "输入搜索名称: ";              getline(cin >> ws, searchTerm);              vector<Product \*> results = store.searchProductsByName(searchTerm, sellerUsername);              if (results.empty())              {                  cout << "未找到匹配商品。" << endl;              }              else              {                  cout << "\n--- 搜索结果 ---" << endl;                  for (const auto \*p : results)                  {                      cout << "-----------------" << endl;                      p->display();                  }                  cout << "-----------------" << endl;              }              break;          }          case 3:          {              // --- 添加商品逻辑 ---              int typeChoice;              cout << "选择商品类型 (1: 图书, 2: 服装, 3: 食品, 4: 其他类型): ";              if (!(cin >> typeChoice) || typeChoice < 1 || typeChoice > 4)              {                  cout << "无效类型。" << endl;                  clearInputBuffer();                  break;              }              clearInputBuffer();              string name, desc, spec1, spec2;              double price;              int qty;              cout << "输入商品名称: ";              getline(cin >> ws, name);              cout << "输入描述: ";              getline(cin >> ws, desc);              cout << "输入价格: ";              while (!(cin >> price) || price < 0)              {                  cout << "无效价格\n";                  clearInputBuffer();                  cout << "输入价格: ";              }              clearInputBuffer();              cout << "输入库存数量: ";              while (!(cin >> qty) || qty < 0)              {                  cout << "无效数量\n";                  clearInputBuffer();                  cout << "输入库存数量: ";              }              clearInputBuffer();              bool success = false;              if (typeChoice == 1)              { // Book                  cout << "输入作者: ";                  getline(cin >> ws, spec1);                  cout << "输入ISBN: ";                  getline(cin >> ws, spec2);                  success = store.createBook(currentUser, name, desc, price, qty, spec1, spec2);              }              else if (typeChoice == 2)              { // Clothing                  cout << "输入尺码: ";                  getline(cin >> ws, spec1);                  cout << "输入颜色: ";                  getline(cin >> ws, spec2);                  success = store.createClothing(currentUser, name, desc, price, qty, spec1, spec2);              }              else if (typeChoice == 3)              { // Food                  cout << "输入过期日期 (YYYY-MM-DD): ";                  getline(cin >> ws, spec1);                  success = store.createFood(currentUser, name, desc, price, qty, spec1);              }              else if (typeChoice == 4)              { // Generic                  cout << "输入分类标签: ";                  getline(cin >> ws, spec1);                  success = store.createGenericProduct(currentUser, name, desc, price, qty, spec1);              }              else              {                  cout << "无效类型。" << endl;                  break;              }              if (success)                  cout << "商品 \"" << name << "\" 添加成功！" << endl;              else                  cout << "商品添加失败！" << endl;              break;          }          case 4:          { // 修改价格              string name;              double price;              cout << "输入要修改价格的商品名称: ";              getline(cin >> ws, name);              cout << "输入新价格: ";              if (!(cin >> price) || price < 0)              {                  cout << "无效价格\n";                  clearInputBuffer();                  break;              }              clearInputBuffer();              if (store.manageProductPrice(currentUser, name, price))                  cout << "价格修改成功。\n";              else                  cout << "价格修改失败。\n";              break;          }          case 5:          { // 修改库存              string name;              int qty;              cout << "输入要修改库存的商品名称: ";              getline(cin >> ws, name);              cout << "输入新库存: ";              if (!(cin >> qty) || qty < 0)              {                  cout << "无效库存\n";                  clearInputBuffer();                  break;              }              clearInputBuffer();              if (store.manageProductQuantity(currentUser, name, qty))                  cout << "库存修改成功。\n";              else                  cout << "库存修改失败。\n";              break;          }          case 6:          { // 修改折扣              string name;              double discount;              cout << "输入要修改折扣的商品名称: ";              getline(cin >> ws, name);              cout << "输入新折扣率 (0 到 100): ";              if (!(cin >> discount) || discount < 0 || discount > 100)              {                  cout << "无效折扣率\n";                  clearInputBuffer();                  break;              }              clearInputBuffer();              if (store.manageProductDiscount(currentUser, name, discount / 100))                  cout << "折扣修改成功。\n";              else                  cout << "折扣修改失败。\n";              break;          }          case 7:          { // 应用分类折扣              string category;              double discount;              std::vector<std::string> sellerCategories = store.getUniqueCategoriesForSeller(sellerUsername);              if (sellerCategories.empty())              {                  cout << "您当前没有任何商品分类可以应用折扣。" << endl;                  break;              }              cout << "您当前的商品分类有: ";              for (size\_t i = 0; i < sellerCategories.size(); ++i)              {                  cout << "\"" << sellerCategories[i] << "\"";                  if (i < sellerCategories.size() - 1)                  {                      cout << ", ";                  }              }              cout << endl;              cout << "输入要应用折扣的分类名称 (从以上列表选择): ";              getline(cin >> ws, category);              bool isValidCategory = false;              for (const auto &cat : sellerCategories)              {                  if (cat == category)                  {                      isValidCategory = true;                      break;                  }              }              if (!isValidCategory)              {                  cout << "输入的分类 \"" << category << "\" 不是您拥有的有效分类或不存在。" << endl;                  break;              }              cout << "输入折扣率 (0 到 100): ";              if (!(cin >> discount) || discount < 0 || discount > 100)              {                  cout << "无效折扣率\n";                  clearInputBuffer();                  break;              }              clearInputBuffer();              if (store.applyCategoryDiscount(currentUser, category, discount / 100))                  cout << "分类折扣应用成功。\n";              else                  cout << "分类折扣应用失败。\n";              break;          }          // case 8:          // {          //     double balance = currentUser->checkBalance();          //     cout << "当前余额: " << balance << endl;          //     break;          // }          case 0:              cout << "正在退出商城管理..." << endl;              break;          default:              cout << "无效选项。" << endl;          }      } while (choice != 0);      return 1; // 返回1表示正常退出商城，继续主循环  } |

#### Order.h

|  |
| --- |
| #ifndef ORDER\_H  #define ORDER\_H  #include <string>  #include <vector>  #include <ctime>  #include <iomanip>  // For std::setprecision  #include <iostream> // For display methods  // Forward declarations  class Product; // From store.h  class User;    // From user.h  class Store;   // From store.h  class OrderItem  {  public:      std::string productId;      std::string productName;      int quantity;      double priceAtPurchase;      std::string sellerUsername;      OrderItem(std::string pid, std::string pName, int qty, double price, std::string sUsername)          : productId(std::move(pid)), productName(std::move(pName)), quantity(qty),            priceAtPurchase(price), sellerUsername(std::move(sUsername)) {}      void display() const;      std::string toStringForSave() const;  };  class Order  {  private:      std::string orderId;      std::string customerUsername;      std::vector<OrderItem> items;      double totalAmount;      bool isProcessed; // Indicates if the order has been processed      time\_t orderTimestamp;      std::string status; // e.g., "PENDING\_CONFIRMATION", "COMPLETED", "CANCELLED\_INSUFFICIENT\_STOCK", "CANCELLED\_INSUFFICIENT\_FUNDS", "CANCELLED\_BY\_USER"      static std::string generateOrderId();  public:      Order(std::string custUsername);      // 新增默认构造函数      Order() : customerUsername(""),                totalAmount(0.0),                orderTimestamp(std::time(nullptr)), status("UNINITIALIZED"),                isProcessed(false)      {          orderId = "EMPTY-" + std::to\_string(orderTimestamp);      }      bool addItemFromProduct(const Product &product, int quantity); // Adds item and updates total      void calculateTotalAmount();                                   // Recalculates total based on items      // Getters      std::string getOrderId() const { return orderId; }      std::string getCustomerUsername() const { return customerUsername; }      const std::vector<OrderItem> &getItems() const { return items; }      double getTotalAmount() const { return totalAmount; }      std::string getStatus() const { return status; }      time\_t getTimestamp() const { return orderTimestamp; }      // Setters      void setStatus(const std::string &newStatus) { status = newStatus; }      void setProcessed(bool processed) { isProcessed = processed; }      bool getProcessed() const { return isProcessed; }      void displaySummary() const;      std::string toStringForSaveHeader() const;      // Persistence (Optional - for saving orders to files)      // bool saveToFile(const std::string& directoryPath) const;      // static Order loadFromFile(const std::string& filePath);  };  #endif // ORDER\_H |

#### Order.cpp

|  |
| --- |
| #include "order.h"  #include "../store/store.h"  #include "../user/user.h" // For Product and User definitions  #include <sstream>  #include <chrono>    // For generating unique ID  #include <fstream>   // For file operations (if saving orders)  #include <algorithm> // For std::remove\_if (if needed for item management)  // --- OrderItem Implementation ---  void OrderItem::display() const  {      std::cout << "    - 商品名称: " << productName << " (ID: " << productId << ")" << std::endl;      std::cout << "      购买数量: " << quantity << ", 商品单价: ¥" << priceAtPurchase                << ", 商品总价: ¥" << (quantity \* priceAtPurchase)                << ", 商家: " << sellerUsername << std::endl;  }  // --- Order Implementation ---  std::string Order::generateOrderId()  {      auto now = std::chrono::system\_clock::now();      auto now\_c = std::chrono::system\_clock::to\_time\_t(now); // Convert to time\_t for tm structure      std::tm now\_tm = \*std::localtime(&now\_c);               // Get local time structure      // Get milliseconds part      auto ms = std::chrono::duration\_cast<std::chrono::milliseconds>(now.time\_since\_epoch()) % 1000;      std::stringstream ss;      ss << "ORD-";      ss << std::put\_time(&now\_tm, "%Y%m%d-%H%M%S");                // Format: YYYYMMDD-HHMMSS      ss << "-" << std::setfill('0') << std::setw(3) << ms.count(); // Append milliseconds, zero-padded      return ss.str();  }  Order::Order(std::string custUsername)      : customerUsername(std::move(custUsername)), totalAmount(0.0), status("PENDING\_CREATION")  {      orderId = generateOrderId();      orderTimestamp = std::time(nullptr);  }  bool Order::addItemFromProduct(const Product &product, int quantity)  {      if (quantity <= 0)          return false;      // Note: Stock check should happen before calling this, or this method could also check      // For now, assumes stock check is done externally before adding to order object      items.emplace\_back(product.getName(), product.getName(), quantity, product.getPrice(), product.getSellerUsername());      calculateTotalAmount(); // Recalculate total every time an item is added      return true;  }  void Order::calculateTotalAmount()  {      totalAmount = 0;      for (const auto &item : items)      {          totalAmount += item.priceAtPurchase \* item.quantity;      }  }  void Order::displaySummary() const  {      std::cout << "\n--- 订单汇总 ---" << std::endl;      std::cout << "订单 ID: " << orderId << std::endl;      std::cout << "买家: " << customerUsername << std::endl;      std::cout << "日期: " << std::asctime(std::localtime(&orderTimestamp)); // ctime adds newline      std::cout << "状态: " << status << std::endl;      std::cout << "下单物品:" << std::endl;      std::cout << std::fixed << std::setprecision(2);      for (const auto &item : items)      {          item.display();      }      std::cout << "---------------------" << std::endl;      std::cout << "总花费: ¥" << totalAmount << std::endl;      std::cout << "---------------------" << std::endl;      std::cout << std::resetiosflags(std::ios\_base::fixed | std::ios\_base::floatfield);  }  // 转为CSV格式字符串  std::string OrderItem::toStringForSave() const  {      std::stringstream ss;      ss << std::fixed << std::setprecision(2);      // Simple CSV-like, ensure no commas in your string fields or use a different delimiter/quoting      ss << productId << "," << productName << "," << quantity << "," << priceAtPurchase << "," << sellerUsername;      return ss.str();  }  std::string Order::toStringForSaveHeader() const  {      std::stringstream ss;      ss << std::fixed << std::setprecision(2);      ss << "OrderID:" << orderId << std::endl;      ss << "CustomerUsername:" << customerUsername << std::endl;      ss << "Timestamp:" << orderTimestamp << std::endl; // Save as raw time\_t      ss << "Status:" << status << std::endl;      ss << "TotalAmount:" << totalAmount << std::endl;      ss << "ItemsCount:" << items.size() << std::endl; // Useful for loading      return ss.str();  }  // Optional: Implement saveToFile and loadFromFile if you want to persist orders  // bool Order::saveToFile(const std::string& directoryPath) const { ... }  // Order Order::loadFromFile(const std::string& filePath) { ... } |

#### Ordermanager.h

|  |
| --- |
| #ifndef ORDER\_MANAGER\_H  #define ORDER\_MANAGER\_H  #include "../order/order.h"  #include "../store/store.h"  #include "../user/user.h"  #include <deque>  #include <string>  #include <vector>  #include <thread>  #include <mutex>  #include <condition\_variable>  #include <atomic>  #include <memory> // 为shared\_ptr添加  class OrderManager  {  private:      std::deque<std::shared\_ptr<Order>> orderQueue;      std::string completedOrdersDirectory;      std::string pendingOrdersFile;      // 多线程支持      std::thread processingThread;      mutable std::mutex queueMutex;      std::condition\_variable queueCondVar;      std::atomic<bool> stopProcessing;      // Helper to save a single order to its own file      bool saveOrderToFile(const Order &order) const;      // 处理线程的主函数      void processingLoop(Store &store, std::vector<User \*> &allUsers);      // 内部处理订单的方法      void processNextOrderInternal(std::shared\_ptr<Order> currentOrder, Store &store, std::vector<User \*> &allUsers);  public:      OrderManager(const std::string &ordersDir);      ~OrderManager();      // 启动和停止处理线程      void startProcessingThread(Store &store, std::vector<User \*> &allUsers);      void stopProcessingThread();      // 提交订单到队列      std::shared\_ptr<Order> submitOrderRequest(const Order &orderRequest);      // 处理队列中的订单      void processNextOrder(Store &store, std::vector<User \*> &allUsers);      void processAllPendingOrders(Store &store, std::vector<User \*> &allUsers);      // 添加这个声明！      size\_t getPendingOrderCount() const;      // 显示待处理订单      void displayPendingOrders() const;  };  #endif // ORDER\_MANAGER\_H |

#### Ordermanager.cpp

|  |
| --- |
| #include "ordermanager.h"  #include "../store/store.h"  #include "../order/order.h"  #include "../user/user.h"  #include <iostream>  #include <fstream>  #include <filesystem>  #include <iomanip>  #include <chrono>  #include <thread>  // 声明外部变量  extern const std::string USER\_FILE;  extern bool startProcessing; // 用于控制订单处理线程的启动状态  using namespace std;  // 构造函数，初始化订单目录  OrderManager::OrderManager(const std::string &ordersDir)      : completedOrdersDirectory(ordersDir), stopProcessing(false)  {      // 确保订单目录存在      if (!std::filesystem::exists(completedOrdersDirectory))      {          try          {              std::filesystem::create\_directories(completedOrdersDirectory);              cout << "已创建订单目录：" << completedOrdersDirectory << endl;          }          catch (const std::filesystem::filesystem\_error &e)          {              cerr << "创建订单目录失败: " << e.what() << endl;          }      }      // 设置待处理订单文件路径      pendingOrdersFile = completedOrdersDirectory + "/pending\_orders.txt";  }  // 析构函数 - 停止处理线程  OrderManager::~OrderManager()  {      stopProcessingThread();  }  // 启动处理线程  void OrderManager::startProcessingThread(Store &store, std::vector<User \*> &allUsers)  {      // 如果线程已经在运行，先停止      stopProcessingThread();      // 重置停止标志      stopProcessing = false;      // 启动处理线程      processingThread = std::thread(&OrderManager::processingLoop, this, std::ref(store), std::ref(allUsers));      cout << "订单处理线程已启动" << endl;  }  // 停止处理线程  void OrderManager::stopProcessingThread()  {      if (processingThread.joinable())      {          {              std::lock\_guard<std::mutex> lock(queueMutex);              stopProcessing = true; // 设置停止标志          }          // 通知条件变量，唤醒等待的线程          queueCondVar.notify\_one();          // 等待线程结束          processingThread.join();          cout << "订单处理线程已停止" << endl;      }  }  // 处理线程的主循环函数  void OrderManager::processingLoop(Store &store, std::vector<User \*> &allUsers)  {      cout << "订单处理线程开始运行..." << endl;      while (!stopProcessing)      {          std::shared\_ptr<Order> orderToProcess;          bool hasOrder = false;          {              // 获取队列互斥锁              std::unique\_lock<std::mutex> lock(queueMutex);              // 等待条件变量：直到队列非空或收到停止信号              queueCondVar.wait(lock, [this]                                { return !orderQueue.empty() || stopProcessing; });              // 如果收到停止信号且队列为空，则退出              if (stopProcessing && orderQueue.empty())                  break;              // 如果队列非空，取出队首订单              if (!orderQueue.empty())              {                  orderToProcess = orderQueue.front();                  orderQueue.pop\_front();                  hasOrder = true;              }          }          // 如果有订单需要处理，则处理它          if (hasOrder)          {              cout << "\n线程正在处理订单 ID: " << orderToProcess->getOrderId()                   << "，客户: " << orderToProcess->getCustomerUsername() << endl;              // 处理订单              processNextOrderInternal(orderToProcess, store, allUsers);          }      }      cout << "订单处理线程已终止" << endl;  }  // 内部处理订单的方法（被处理线程调用）  void OrderManager::processNextOrderInternal(std::shared\_ptr<Order> currentOrder, Store &store, std::vector<User \*> &allUsers)  {      cout << "正在处理订单 ID: " << currentOrder->getOrderId()           << "，客户: " << currentOrder->getCustomerUsername() << endl;      currentOrder->setStatus("PROCESSING");      // 找到订单对应的客户      Customer \*customer = dynamic\_cast<Customer \*>(User::findUser(allUsers, currentOrder->getCustomerUsername()));      if (!customer)      {          cerr << "错误: 找不到订单对应的客户 " << currentOrder->getCustomerUsername() << endl;          currentOrder->setStatus("FAILED\_CUSTOMER\_NOT\_FOUND");          saveOrderToFile(\*currentOrder);   // 注意这里需要解引用          currentOrder->setProcessed(true); // 标记处理完成          return;      }      // 第一阶段：重新验证商品（库存可能在队列等待期间发生变化）      for (const auto &item : currentOrder->getItems())      {          Product \*product = store.findProductByName(item.productId);          if (!product)          {              cerr << "错误: 商品 \"" << item.productName << "\" 不存在或已下架。订单取消。" << endl;              currentOrder->setStatus("FAILED\_PRODUCT\_NOT\_FOUND");              saveOrderToFile(\*currentOrder);              currentOrder->setProcessed(true);              return;          }          if (product->getQuantity() < item.quantity)          {              cerr << "错误: 商品 \"" << item.productName << "\" 库存不足 (需要 "                   << item.quantity << ", 现有 " << product->getQuantity() << ")。订单取消。" << endl;              currentOrder->setStatus("FAILED\_INSUFFICIENT\_STOCK");              saveOrderToFile(\*currentOrder);              currentOrder->setProcessed(true);              return;          }      }      // 第二阶段：验证客户余额      if (customer->checkBalance() < currentOrder->getTotalAmount())      {          cerr << "错误: 客户 " << customer->getUsername() << " 余额不足。订单取消。" << endl;          currentOrder->setStatus("FAILED\_INSUFFICIENT\_FUNDS");          saveOrderToFile(\*currentOrder);          currentOrder->setProcessed(true);          return;      }      // 第三阶段：处理支付和更新库存      if (!customer->withdraw(currentOrder->getTotalAmount()))      {          cerr << "严重错误: 客户 " << customer->getUsername()               << " 扣款失败，尽管余额检查已通过。订单: " << currentOrder->getOrderId() << endl;          currentOrder->setStatus("FAILED\_PAYMENT\_ERROR");          saveOrderToFile(\*currentOrder);          currentOrder->setProcessed(true);          return;      }      bool allSellersPaid = true;      for (const auto &item : currentOrder->getItems())      {          Product \*product = store.findProductByName(item.productId);          // 更新商品库存          product->setQuantity(product->getQuantity() - item.quantity);          // 向卖家转账          User \*seller = User::findUser(allUsers, item.sellerUsername);          if (seller)          {              double itemAmount = item.priceAtPurchase \* item.quantity;              if (!seller->deposit(itemAmount))              {                  cerr << "警告: 向商家 " << item.sellerUsername                       << " 支付商品 " << item.productName                       << " 的款项 ¥" << fixed << setprecision(2) << itemAmount                       << " 失败。订单 ID: " << currentOrder->getOrderId() << endl;                  allSellersPaid = false;              }              else              {                  cout << "已向商家 " << item.sellerUsername                       << " 支付 ¥" << fixed << setprecision(2) << itemAmount << endl;              }          }          else if (!item.sellerUsername.empty())          {              cerr << "警告: 找不到商品 \"" << item.productName                   << "\" 的商家 \"" << item.sellerUsername                   << "\"。订单 ID: " << currentOrder->getOrderId() << endl;              allSellersPaid = false;          }      }      // 保存所有更改（用户和商店）      User::saveUsersToFile(allUsers, USER\_FILE);      store.saveAllProducts();      // 设置最终订单状态      if (allSellersPaid)      {          currentOrder->setStatus("COMPLETED");      }      else      {          currentOrder->setStatus("COMPLETED\_WITH\_PAYMENT\_ISSUES");      }      cout << "订单 ID: " << currentOrder->getOrderId()           << " 处理完成。最终状态: " << currentOrder->getStatus() << endl;      currentOrder->displaySummary();      saveOrderToFile(\*currentOrder);      // 标记订单已处理完成（关键步骤！）      currentOrder->setProcessed(true);  }  // 提交订单到处理队列 (线程安全版本)  std::shared\_ptr<Order> OrderManager::submitOrderRequest(const Order &orderRequest)  {      // 创建订单的共享指针副本      auto orderPtr = std::make\_shared<Order>(orderRequest);      // 设置订单状态为等待队列处理      orderPtr->setStatus("PENDING\_IN\_QUEUE");      orderPtr->setProcessed(false);      // 锁定队列      std::lock\_guard<std::mutex> lock(queueMutex);      // 将订单加入队列      orderQueue.push\_back(orderPtr);      // 通知处理线程有新订单      queueCondVar.notify\_one();      cout << "订单 ID: " << orderPtr->getOrderId()           << " 已提交到处理队列，当前队列长度: " << orderQueue.size() << endl;      // 返回订单共享指针      return orderPtr;  }  // 处理队列中的一个订单 (公共方法版，委托给内部方法)  void OrderManager::processNextOrder(Store &store, std::vector<User \*> &allUsers)  {      std::shared\_ptr<Order> orderToProcess;      bool hasOrder = false;      {          std::lock\_guard<std::mutex> lock(queueMutex);          if (!orderQueue.empty())          {              orderToProcess = orderQueue.front();              orderQueue.pop\_front();              hasOrder = true;          }      }      if (hasOrder)      {          processNextOrderInternal(orderToProcess, store, allUsers);      }  }  // 处理所有待处理订单 (线程安全版本)  void OrderManager::processAllPendingOrders(Store &store, std::vector<User \*> &allUsers)  {      std::vector<std::shared\_ptr<Order>> ordersToProcess;      // 首先，从队列中安全地取出所有订单      {          std::lock\_guard<std::mutex> lock(queueMutex);          while (!orderQueue.empty())          {              ordersToProcess.push\_back(orderQueue.front());              orderQueue.pop\_front();          }      }      // 如果没有订单，直接返回      if (ordersToProcess.empty())      {          cout << "当前没有待处理订单。" << endl;          return;      }      cout << "\n--- 开始处理 " << ordersToProcess.size() << " 个待处理订单 ---" << endl;      // 处理获取的订单      for (auto &order : ordersToProcess)      {          processNextOrderInternal(order, store, allUsers);      }      cout << "--- 成功处理 " << ordersToProcess.size() << " 个订单 ---" << endl;  }  // 获取待处理订单数量 (线程安全版本)  size\_t OrderManager::getPendingOrderCount() const  {      std::lock\_guard<std::mutex> lock(queueMutex);      return orderQueue.size();  }  // 显示待处理订单 (线程安全版本)  void OrderManager::displayPendingOrders() const  {      std::lock\_guard<std::mutex> lock(queueMutex);      if (orderQueue.empty())      {          cout << "当前队列中没有待处理订单。" << endl;          return;      }      cout << "\n--- 待处理订单队列 ---" << endl;      int i = 1;      for (const auto &order : orderQueue)      {          cout << i++ << ". 订单 ID: " << order->getOrderId()               << ", 客户: " << order->getCustomerUsername()               << ", 金额: ¥" << fixed << setprecision(2) << order->getTotalAmount()               << ", 状态: " << order->getStatus() << endl;      }      cout << "-----------------------------" << endl;  }  // 将订单保存到文件  bool OrderManager::saveOrderToFile(const Order &order) const  {      if (completedOrdersDirectory.empty())      {          cerr << "错误: 未设置订单目录。" << endl;          return false;      }      // 创建基于订单ID的文件名      std::string filePath = completedOrdersDirectory + "/" + order.getOrderId() + ".txt";      std::ofstream outFile(filePath);      if (!outFile)      {          cerr << "错误: 无法打开文件保存订单: " << filePath << endl;          return false;      }      // 保存订单头部信息      outFile << order.toStringForSaveHeader();      // 保存订单项      for (const auto &item : order.getItems())      {          outFile << item.toStringForSave() << endl;      }      outFile.close();      cout << "订单 " << order.getOrderId() << " 已保存到 " << filePath << endl;      return true;  } |

### imgUI

#### ui\_main.cpp

|  |
| --- |
| #include <iostream>  #include <vector>  #include <string>  #include "user/user.h"  #include "store/store.h"  #include "ordermanager/ordermanager.h"  #include "ui/ui.h"  #include <windows.h>  // 文件路径常量  extern const std::string USER\_FILE = "./user/users.txt";  extern const std::string STORE\_FILE = "./store";  extern const std::string CART\_FILE = "./user/carts";  extern const std::string ORDER\_DIR = "./order/orders"; // 订单目录  OrderManager g\_orderManager(ORDER\_DIR);                // 创建全局订单管理器对象  int main()  {  // 设置控制台输出编码为UTF-8  #ifdef \_WIN32      SetConsoleOutputCP(65001);      SetConsoleCP(65001);  #endif      std::vector<User \*> users;      User \*currentUser = nullptr;      // 创建 Store 对象      Store store(STORE\_FILE);      // 加载用户数据      users = User::loadUsersFromFile(USER\_FILE);      std::cout << "已加载 " << users.size() << " 个用户数据。" << std::endl;      // 加载商店数据      store.loadAllProducts(); // 启动订单处理线程      g\_orderManager.startProcessingThread(store, users);      std::cout << "订单处理线程已启动。" << std::endl;      // 创建并初始化UI      UI ui(800, 600);      ui.setOrderManager(&g\_orderManager);      if (!ui.init())      {          std::cerr << "UI初始化失败，程序退出" << std::endl;          return 1;      }      // 启动UI主循环      ui.mainLoop(users, store); // 停止订单处理线程      g\_orderManager.stopProcessingThread();      // 保存用户数据      User::saveUsersToFile(users, USER\_FILE);      std::cout << "用户数据已保存。" << std::endl;      // 程序结束，释放资源      for (User \*user : users)      {          delete user;      }      users.clear();      return 0;  } |

#### user.h

|  |
| --- |
| #ifndef USER\_H  #define USER\_H  #include <iostream>  #include <string>  #include <vector>  #include <fstream>  #include <limits>  #include <iomanip>          // For output formatting in cart display  #include "../order/order.h" // For OrderItem  // 前向声明  class Product;  class Store;  // 购物车中的商品项 (现在作为 Customer 类的内部结构或辅助结构)  struct CartItem  {      std::string productId; // 商品名称或唯一ID      std::string productName;      int quantity;      double priceAtAddition; // 添加到购物车时的价格      std::string sellerUsername;      CartItem(std::string pid = "", std::string pName = "", int qty = 0, double price = 0.0, std::string sName = "")          : productId(pid), productName(pName), quantity(qty), priceAtAddition(price), sellerUsername(sName) {}  };  class User  {  protected:      std::string username;      std::string password;      std::string userType;      double balance;      static void clearInputBuffer()      {          std::cin.clear();          std::cin.ignore(std::numeric\_limits<std::streamsize>::max(), '\n');      }  public:      User(std::string username, std::string password, double balance = 0.0,           bool is\_customer = false, bool is\_seller = false, bool is\_admin = false);      User();      virtual ~User() = default;      virtual std::string getUserType() const = 0;      void displayInfo(int index = -1) const;      bool changePassword(const std::string &oldPassword, const std::string &newPassword);      bool deposit(double amount);      bool withdraw(double amount);      double checkBalance() const;      static User \*registerUser(std::vector<User \*> &users);      static User \*userLogin(const std::vector<User \*> &users);      static bool saveUsersToFile(const std::vector<User \*> &users, const std::string &filename);      static std::vector<User \*> loadUsersFromFile(const std::string &filename);      static User \*findUser(const std::vector<User \*> &users, const std::string &username);      static void displayAllUsers(const std::vector<User \*> &users);      static bool isUsernameExists(const std::vector<User \*> &users, const std::string &username);      std::string getUsername() const;      std::string getPassword() const;      void setUsername(const std::string &newUsername);      void setPassword(const std::string &newPassword);      void setBalance(double newBalance);      static User \*login(const std::vector<User \*> &users, const std::string &username, const std::string &password);  };  // 消费者类  class Customer : public User  {  private:      std::string cartDirectoryPath; // 存储购物车文件的目录路径      // 购物车文件操作的辅助方法      std::string getCartFilePath() const;  public:      Customer(std::string username, std::string password, const std::string &cartDir, double balance = 0.0);      Customer(const std::string &cartDir); // 默认构造函数      std::string getUserType() const override { return "customer"; }      std::vector<CartItem> shoppingCartItems; // 直接在 Customer 中存储购物车项      void loadCartFromFile();      bool saveCartToFile() const;      // 购物车管理方法      bool addToCart(const Product &product, int quantity);      bool removeCartItem(const std::string &productName);      bool updateCartItemQuantity(const std::string &productName, int newQuantity, Store &store); // 需要 store 检查库存      void viewCart(Store &store, std::vector<User \*> &allUsers);                                 // 查看并管理购物车，结算也在这里处理      bool isCartEmpty() const;      void clearCartAndFile(); // 结算后清空购物车及文件  };  // 商家类  class Seller : public User  {  public:      Seller(std::string username, std::string password, double balance = 0.0);      Seller();      std::string getUserType() const override { return "seller"; }  };  // 管理员类  class Admin : public User  {  public:      Admin(std::string username, std::string password, double balance = 0.0);      Admin();      std::string getUserType() const override { return "admin"; }  };  #endif // USER\_H |

#### user.cpp

|  |
| --- |
| #include "user.h"  #include "../store/store.h" // 需要 Product 和 Store 的完整定义  #include "../order/order.h"  #include "../ordermanager/ordermanager.h"  #include <iostream>  #include <limits>  #include <string>  #include <fstream>  #include <sstream>  #include <filesystem> // 用于文件操作  #include <algorithm>  // for std::remove\_if  #include <iomanip>    // for std::fixed, std::setprecision  #include <thread>  using namespace std;  // 声明在 start.cpp 中定义的全局常量  extern const string CART\_FILE; // 用于 Customer 构造函数  extern const string USER\_FILE; // 用于保存用户信息  // User 构造函数实现  User::User(std::string u\_name, std::string pwd, double bal, bool is\_customer, bool is\_seller, bool is\_admin)      : username(u\_name), password(pwd), balance(bal)  {      // userType 在派生类中设置  }  User::User() : username(""), password(""), balance(0.0), userType("unknown") {}  // --- Customer 类购物车相关方法实现 ---  std::string Customer::getCartFilePath() const  {      if (this->username.empty())      { // 防止用户名为空时生成无效路径          // cerr << "警告: Customer 用户名为空，无法生成购物车路径。" << endl;          return ""; // 或者抛出异常，或者返回一个默认的无效路径      }      return this->cartDirectoryPath + "/" + this->username + "\_cart.txt";  }  void Customer::loadCartFromFile()  {      shoppingCartItems.clear();      std::string cartPath = getCartFilePath();      if (cartPath.empty())          return; // 用户名为空，无法加载      std::ifstream file(cartPath);      if (!file.is\_open())      {          return; // 文件不存在或无法打开是正常情况（购物车为空）      }      std::string line;      while (getline(file, line))      {          std::stringstream ss(line);          std::string productId, productName, sellerUsernameStr;          int quantity;          double priceAtAddition;          if (getline(ss, productId, ',') &&              getline(ss, productName, ',') &&              (ss >> quantity) &&              (ss.ignore(1) && ss >> priceAtAddition) &&              (ss.ignore(1) && getline(ss, sellerUsernameStr)))          {              shoppingCartItems.emplace\_back(productId, productName, quantity, priceAtAddition, sellerUsernameStr);          }          else          {              std::cerr << "警告: 购物车文件 " << cartPath << " 中存在格式错误的行: " << line << std::endl;          }      }      file.close();  }  bool Customer::saveCartToFile() const  {      std::string cartPath = getCartFilePath();      if (cartPath.empty())      {          std::cerr << "错误: Customer 用户名为空，无法保存购物车。" << std::endl;          return false;      }      std::ofstream file(cartPath); // 会覆盖旧文件      if (!file.is\_open())      {          std::cerr << "错误: 无法打开购物车文件进行写入: " << cartPath << std::endl;          return false;      }      for (const auto &item : shoppingCartItems)      {          file << item.productId << ","               << item.productName << ","               << item.quantity << ","               << item.priceAtAddition << ","               << item.sellerUsername << std::endl;      }      file.close();      return true;  }  // Customer 构造函数实现  Customer::Customer(std::string uname, std::string pwd, const std::string &cartDir, double bal)      : User(uname, pwd, bal, true, false, false), cartDirectoryPath(cartDir)  {      this->userType = "customer";      if (!this->username.empty())      { // 只有当用户名有效时才加载购物车          loadCartFromFile();      }  }  // 默认构造函数  Customer::Customer(const std::string &cartDir)      : User(), cartDirectoryPath(cartDir)  {      this->userType = "customer";      // 注意: 此时 username 为空，loadCartFromFile 不会执行。      // 如果后续通过 setUsername 设置了用户名，需要一种机制来加载对应的购物车。      // 或者，强制 Customer 总是在有用户名的情况下创建。  }  bool Customer::addToCart(const Product &product, int quantity)  {      if (this->username.empty())      {          cout << "错误：无法为匿名用户添加购物车。" << endl;          return false;      }      if (quantity <= 0)      {          cout << "添加数量必须大于0。" << endl;          return false;      }      bool found = false;      for (auto &item : shoppingCartItems)      {          if (item.productId == product.getName())          { // 假设商品名称是唯一ID              item.quantity += quantity;              // 可选：更新 priceAtAddition 为最新价格或保持不变              found = true;              break;          }      }      if (!found)      {          shoppingCartItems.emplace\_back(product.getName(), product.getName(), quantity, product.getPrice(), product.getSellerUsername());      }      cout << "\"" << product.getName() << "\" 已成功加入购物车。" << endl;      return saveCartToFile();  }  bool Customer::removeCartItem(const std::string &productName)  {      if (this->username.empty())          return false;      auto initial\_size = shoppingCartItems.size();      shoppingCartItems.erase(std::remove\_if(shoppingCartItems.begin(), shoppingCartItems.end(),                                             [&](const CartItem &item)                                             { return item.productName == productName; }),                              shoppingCartItems.end());      if (shoppingCartItems.size() < initial\_size)      {          cout << "商品 \"" << productName << "\" 已从购物车移除。" << endl;          return saveCartToFile();      }      else      {          cout << "未在购物车中找到商品: " << productName << endl;          return false;      }  }  bool Customer::updateCartItemQuantity(const std::string &productName, int newQuantity, Store &store)  {      if (this->username.empty())          return false;      if (newQuantity < 0)      {          cout << "数量不能为负。" << endl;          return false;      }      bool updated = false;      for (auto &item : shoppingCartItems)      {          if (item.productName == productName)          {              Product \*p\_info = store.findProductByName(productName); // 检查最新库存              if (p\_info && newQuantity > p\_info->getQuantity())              {                  cout << "库存不足！无法将购物车中 \"" << productName << "\" 的数量修改为 " << newQuantity                       << "。当前库存: " << p\_info->getQuantity() << endl;                  return false;              }              item.quantity = newQuantity;              updated = true;              break;          }      }      if (!updated)      {          cout << "未在购物车中找到商品: " << productName << endl;          return false;      }      // 移除数量为0的商品      shoppingCartItems.erase(std::remove\_if(shoppingCartItems.begin(), shoppingCartItems.end(),                                             [](const CartItem &item)                                             { return item.quantity <= 0; }),                              shoppingCartItems.end());      cout << "购物车商品 \"" << productName << "\" 数量已更新。" << endl;      return saveCartToFile();  }  void Customer::clearCartAndFile()  {      if (this->username.empty())          return;      shoppingCartItems.clear();      std::string cartPath = getCartFilePath();      if (!cartPath.empty() && std::filesystem::exists(cartPath))      {          try          {              std::filesystem::remove(cartPath);          }          catch (const std::filesystem::filesystem\_error &err)          {              std::cerr << "错误: 删除购物车文件失败: " << err.what() << std::endl;          }      }  }  bool Customer::isCartEmpty() const  {      return shoppingCartItems.empty();  }  void Customer::viewCart(Store &store, std::vector<User \*> &allUsers)  {      if (this->username.empty())      {          cout << "错误：匿名用户无法查看购物车。" << endl;          return;      }      // 确保购物车是最新的（如果其他地方可能修改文件但未更新内存）      // loadCartFromFile(); // 通常在每次操作后保存，所以这里可能不需要，除非有外部修改购物车的可能      if (shoppingCartItems.empty())      {          cout << "\n您的购物车是空的。" << endl;          return;      }      double totalCartPriceBasedOnCurrent = 0;      cout << "\n--- 我的购物车 ---" << endl;      cout << std::fixed << std::setprecision(2); // 设置输出格式      for (size\_t i = 0; i < shoppingCartItems.size(); ++i)      {          const auto &item = shoppingCartItems[i];          Product \*p\_info = store.findProductByName(item.productId);          double currentItemPrice = p\_info ? p\_info->getPrice() : item.priceAtAddition;          cout << i + 1 << ". 商品: " << item.productName               << " | 数量: " << item.quantity               << " | 当前单价: ¥" << currentItemPrice               << " | 小计: ¥" << (item.quantity \* currentItemPrice) << endl;          if (p\_info)          {              if (p\_info->getQuantity() < item.quantity)              {                  cout << "   注意: " << item.productName << " 当前库存(" << p\_info->getQuantity() << ")不足购物车数量(" << item.quantity << ")!" << endl;              }              if (p\_info->getPrice() != item.priceAtAddition)              {                  cout << "   提示: " << item.productName << " 加入时价格为 ¥" << item.priceAtAddition << ", 当前价格已变为 ¥" << p\_info->getPrice() << endl;              }          }          else          {              cout << "   警告: 商品 " << item.productName << " 可能已从商店下架!" << endl;          }          totalCartPriceBasedOnCurrent += item.quantity \* currentItemPrice;      }      cout << "--------------------" << endl;      cout << "购物车总计 (按当前价格): ¥" << totalCartPriceBasedOnCurrent << endl;      cout << "--------------------" << endl;      int cartChoice = -1;      do      {          cout << "\n购物车操作:" << endl;          cout << "1. 修改商品数量" << endl;          cout << "2. 移除商品" << endl;          cout << "3. 生成订单并结算" << endl;          cout << "0. 返回商城" << endl;          cout << "请选择: ";          if (!(cin >> cartChoice))          {              cout << "无效输入。" << endl;              User::clearInputBuffer();              cartChoice = -1;              continue;          }          User::clearInputBuffer();          switch (cartChoice)          {          case 1:          {              string nameToUpdate;              int newQty;              cout << "输入要修改数量的商品名称: ";              getline(cin >> ws, nameToUpdate);              cout << "输入新的数量 (输入0将移除该商品): ";              if (!(cin >> newQty))              {                  cout << "无效数量输入。" << endl;                  User::clearInputBuffer();                  break;              }              User::clearInputBuffer();              if (updateCartItemQuantity(nameToUpdate, newQty, store))              {                  // 成功后重新加载并显示购物车 (递归调用或重新加载数据)                  // 为了避免无限递归，这里直接返回，让外层循环重新调用 viewCart                  cout << "购物车已更新。" << endl;              }              return viewCart(store, allUsers); // 重新显示更新后的购物车          }          case 2:          {              string nameToRemove;              cout << "输入要移除的商品名称: ";              getline(cin >> ws, nameToRemove);              if (removeCartItem(nameToRemove))              {                  cout << "购物车已更新。" << endl;              }              return viewCart(store, allUsers); // 重新显示更新后的购物车          }          case 3: // Generate Order and Checkout          {              if (shoppingCartItems.empty())              {                  cout << "购物车是空的，无法生成订单。" << endl;                  break;              }              Order currentOrder(this->getUsername()); // Create a new order object              currentOrder.setStatus("PENDING\_VALIDATION");              // 第一阶段：验证商品并构建订单（预检查）              bool preCheckOk = true;              for (const auto &cart\_item : shoppingCartItems)              {                  Product \*product = store.findProductByName(cart\_item.productId);                  if (!product)                  {                      cout << "错误: 购物车商品 \"" << cart\_item.productName << "\" 已不存在。" << endl;                      currentOrder.setStatus("CANCELLED\_PRODUCT\_NOT\_FOUND");                      preCheckOk = false;                      break;                  }                  if (product->getQuantity() < cart\_item.quantity)                  {                      cout << "错误: 购物车商品 \"" << cart\_item.productName << "\" 库存不足 (需要 "                           << cart\_item.quantity << ", 现有 " << product->getQuantity() << ")。" << endl;                      currentOrder.setStatus("CANCELLED\_INSUFFICIENT\_STOCK");                      preCheckOk = false;                      break;                  }                  currentOrder.addItemFromProduct(\*product, cart\_item.quantity);              }              if (!preCheckOk)              {                  currentOrder.displaySummary(); // Display order with cancelled status                  cout << "订单无法继续，请调整购物车后重试。" << endl;                  break; // Back to cart menu              }              // 第二阶段：验证客户余额（只是初步检查，队列处理时还会再次检查）              if (this->checkBalance() < currentOrder.getTotalAmount())              {                  cout << "错误: 您的余额不足以支付此订单！" << endl;                  currentOrder.setStatus("CANCELLED\_INSUFFICIENT\_FUNDS");                  // currentOrder.displaySummary();                  cout << "您的余额:   ¥" << this->checkBalance() << endl;                  break; // Back to cart menu              }              // 第三阶段：显示订单并请求确认              currentOrder.setStatus("PENDING\_CONFIRMATION");              currentOrder.displaySummary();              cout << "您的当前余额: ¥" << std::fixed << std::setprecision(2) << this->checkBalance() << endl;              cout << "支付后余额将为: ¥" << (this->checkBalance() - currentOrder.getTotalAmount()) << endl;              cout << "--------------------" << endl;              char confirmChoice;              cout << "确认支付并完成订单吗? (y/n): ";              cin >> confirmChoice;              User::clearInputBuffer();              if (tolower(confirmChoice) != 'y')              {                  currentOrder.setStatus("CANCELLED\_BY\_USER");                  currentOrder.displaySummary();                  cout << "订单已取消。" << endl;                  break; // Back to cart menu              }              // Stage 4: Process payment and update inventory (Actual "lock" and transaction)              cout << "\n正在处理订单，请稍候..." << endl;              currentOrder.setStatus("PROCESSING");              // 访问OrderManager，提交订单 - 修改这部分              extern OrderManager g\_orderManager;              auto submittedOrder = g\_orderManager.submitOrderRequest(currentOrder);              cout << "订单已成功提交到处理队列，订单ID: " << submittedOrder->getOrderId() << endl;              cout << "正在等待订单处理，请稍候..." << endl;              // 等待订单处理完成              int waitCounter = 0;              const int MAX\_WAIT\_CYCLES = 100; // 最多等待10秒              while (!submittedOrder->getProcessed() && waitCounter < MAX\_WAIT\_CYCLES)              {                  // 短暂休眠，避免过度消耗CPU                  std::this\_thread::sleep\_for(std::chrono::milliseconds(100));                  waitCounter++;                  // 每秒显示等待状态                  if (waitCounter % 10 == 0)                  {                      cout << "已等待 " << (waitCounter / 10) << " 秒..." << endl;                  }              }              // 检查是否超时              if (waitCounter >= MAX\_WAIT\_CYCLES)              {                  cout << "等待订单处理超时！订单可能仍在处理中。" << endl;              }              else              {                  cout << "订单已处理完成! 状态: " << submittedOrder->getStatus() << endl;                  if (submittedOrder->getStatus() == "COMPLETED" ||                      submittedOrder->getStatus() == "COMPLETED\_WITH\_PAYMENT\_ISSUES")                  {                      clearCartAndFile(); // 成功提交后清空购物车                  }              }              cout << std::resetiosflags(std::ios\_base::fixed | std::ios\_base::floatfield);              return; // 退出viewCart，返回Customer\_page商店菜单              // if (!this->withdraw(currentOrder.getTotalAmount()))              // {              //     currentOrder.setStatus("FAILED\_PAYMENT\_WITHDRAWAL");              //     currentOrder.displaySummary();              //     cout << "错误: 支付失败。请检查余额或联系客服。" << endl;              //     // Potentially log this critical failure              //     break;              // }              // for (const auto &order\_item : currentOrder.getItems())              // {              //     Product \*product = store.findProductByName(order\_item.productId); // Re-fetch, though should be same              //     if (product)              //     { // Should always be true due to pre-checks              //         product->setQuantity(product->getQuantity() - order\_item.quantity);              //         User \*seller = User::findUser(allUsers, order\_item.sellerUsername);              //         if (seller)              //         {              //             if (!seller->deposit(order\_item.priceAtPurchase \* order\_item.quantity))              //             {              //                 cerr << "警告: 向商家 " << seller->getUsername() << " 转账失败 (商品: "              //                      << order\_item.productName << "). 订单号: " << currentOrder.getOrderId() << endl;              //                 // Log this for reconciliation              //             }              //         }              //         else if (!order\_item.sellerUsername.empty())              //         {              //             cerr << "警告: 未找到商品 \"" << order\_item.productName << "\" 的商家 \""              //                  << order\_item.sellerUsername << "\". 订单号: " << currentOrder.getOrderId() << endl;              //         }              //         else              //         {              //             cerr << "警告: 商品 \"" << order\_item.productName << "\" 没有商家信息. 订单号: " << currentOrder.getOrderId() << endl;              //         }              //     }              // }              // User::saveUsersToFile(allUsers, USER\_FILE);              // store.saveAllProducts(); // Or save only affected sellers              // currentOrder.setStatus("COMPLETED");              // cout << "\n订单处理完成！" << endl;              // currentOrder.displaySummary(); // Display final completed order              // // Optional: currentOrder.saveToFile(ORDER\_DIRECTORY\_BASE + "/" + this->getUsername());              // clearCartAndFile(); // Clear cart after successful order              // cout << std::resetiosflags(std::ios\_base::fixed | std::ios\_base::floatfield);              // return; // Exit viewCart, back to Customer\_page store menu          }          case 0:              // 返回商城，什么也不做，循环会结束              break;          default:              cout << "无效选项。" << endl;          }      } while (cartChoice != 0);      cout << std::resetiosflags(std::ios\_base::fixed | std::ios\_base::floatfield); // 重置输出格式  }  // Seller 构造函数实现  Seller::Seller(std::string uname, std::string pwd, double bal)      : User(uname, pwd, bal, false, true, false)  {      this->userType = "seller";  }  Seller::Seller() : User() { this->userType = "seller"; }  // Admin 构造函数实现  Admin::Admin(std::string uname, std::string pwd, double bal)      : User(uname, pwd, bal, false, false, true)  {      this->userType = "admin";  }  Admin::Admin() : User() { this->userType = "admin"; }  // --- 静态方法：用户注册实现 ---  User \*User::registerUser(std::vector<User \*> &users)  {      string newUsername, newPassword;      double initialBalance = 0.0;      int userTypeChoice;      cout << "请输入新用户名: ";      getline(cin >> ws, newUsername);      if (isUsernameExists(users, newUsername))      {          cout << "用户名已存在！" << endl;          return nullptr;      }      cout << "请输入密码: ";      getline(cin >> ws, newPassword);      cout << "选择用户类型 (1: 消费者, 2: 商家): ";      if (!(cin >> userTypeChoice) || (userTypeChoice != 1 && userTypeChoice != 2))      {          cout << "无效的用户类型选择。" << endl;          clearInputBuffer();          return nullptr;      }      clearInputBuffer();      User \*newUser = nullptr;      if (userTypeChoice == 1)      {                                                                                // Customer          newUser = new Customer(newUsername, newPassword, CART\_FILE, initialBalance); // 使用 CART\_FILE      }      else      { // Seller          newUser = new Seller(newUsername, newPassword, initialBalance);      }      if (newUser)      {          users.push\_back(newUser);          cout << (userTypeChoice == 1 ? "消费者" : "商家") << "用户 \"" << newUsername << "\" 注册成功！" << endl;      }      return newUser;  }  // 从文件加载用户信息  std::vector<User \*> User::loadUsersFromFile(const std::string &filename)  {      std::vector<User \*> loadedUsers;      std::ifstream file(filename);      if (!file.is\_open())      {          return loadedUsers;      }      std::string line;      while (getline(file, line))      {          std::stringstream ss(line);          std::string uname, pwd, type;          double bal;          if (              getline(ss, uname, ',') &&              getline(ss, pwd, ',') &&              getline(ss, type, ',') &&              (ss >> bal))          {              if (type == "customer")              {                  loadedUsers.push\_back(new Customer(uname, pwd, CART\_FILE, bal)); // 使用 CART\_FILE              }              else if (type == "seller")              {                  loadedUsers.push\_back(new Seller(uname, pwd, bal));              }              else if (type == "admin")              {                  loadedUsers.push\_back(new Admin(uname, pwd, bal));              }          }      }      file.close();      return loadedUsers;  }  // --- User 类其他成员函数实现 ---  // getUsername, getPassword, displayInfo, changePassword, deposit, withdraw, checkBalance,  // saveUsersToFile, findUser, isUsernameExists, login  // 这些方法的实现应该保持不变，除非它们需要与新的购物车逻辑交互（但看起来不需要）。  std::string User::getUsername() const { return username; }  std::string User::getPassword() const { return password; } // 密码不应这样直接返回，仅为示例  void User::setUsername(const std::string &newUsername) { username = newUsername; }  void User::setPassword(const std::string &newPassword) { password = newPassword; }  void User::setBalance(double newBalance)  {      if (newBalance >= 0)          balance = newBalance;  }  void User::displayInfo(int index) const  {      if (index != -1)      {          cout << "用户 " << index << ": ";      }      cout << "用户名: " << username << ", 类型: " << userType << ", 余额: " << balance << endl;  }  bool User::changePassword(const std::string &oldPassword, const std::string &newPassword)  {      if (this->password == oldPassword)      {          this->password = newPassword;          return true;      }      cout << "原密码错误！" << endl;      return false;  }  bool User::deposit(double amount)  {      if (amount <= 0)      {          cout << "充值金额必须为正！" << endl;          return false;      }      balance += amount;      return true;  }  bool User::withdraw(double amount)  {      if (amount <= 0)      {          cout << "消费金额必须为正！" << endl;          return false;      }      if (balance < amount)      {          cout << "余额不足！" << endl;          return false;      }      balance -= amount;      return true;  }  double User::checkBalance() const  {      return balance;  }  bool User::saveUsersToFile(const std::vector<User \*> &users, const std::string &filename)  {      ofstream file(filename);      if (!file.is\_open())      {          cerr << "错误: 无法打开用户文件进行写入: " << filename << endl;          return false;      }      for (const auto \*user : users)      {          if (user)          {              file << user->getUsername() << ","                   << user->getPassword() << ","                   << user->getUserType() << ","                   << user->checkBalance() << endl;          }      }      file.close();      return true;  }  User \*User::findUser(const std::vector<User \*> &users, const std::string &findUsername)  {      for (User \*user : users)      {          if (user->getUsername() == findUsername)          {              return user;          }      }      return nullptr;  }  bool User::isUsernameExists(const std::vector<User \*> &users, const std::string &checkUsername)  {      return findUser(users, checkUsername) != nullptr;  }  User \*User::userLogin(const std::vector<User \*> &users)  {      string loginUsername, loginPassword;      cout << "请输入用户名: ";      getline(cin >> ws, loginUsername);      cout << "请输入密码: ";      getline(cin >> ws, loginPassword);      User \*foundUser = findUser(users, loginUsername);      if (foundUser && foundUser->getPassword() == loginPassword)      {          cout << "登录成功！欢迎, " << loginUsername << "!" << endl;          // 如果是 Customer，并且之前用户名为空（例如通过默认构造后设置用户名），则加载其购物车          if (Customer \*cust = dynamic\_cast<Customer \*>(foundUser))          {              if (cust->shoppingCartItems.empty() && !cust->getUsername().empty())              {                             // 假设 shoppingCartItems 为空表示未加载                                            // cust->cartDirectoryPath 应该在构造时已设置                  cust->loadCartFromFile(); // 尝试加载购物车              }          }          return foundUser;      }      cout << "用户名或密码错误！" << endl;      return nullptr;  }  User \*User::login(const std::vector<User \*> &users, const std::string &username\_param, const std::string &password\_param)  {      User \*foundUser = findUser(users, username\_param);      if (foundUser && foundUser->getPassword() == password\_param)      {          return foundUser;      }      return nullptr;  }  void User::displayAllUsers(const std::vector<User \*> &users)  {      if (users.empty())      {          cout << "当前没有注册用户。" << endl;          return;      }      cout << "\n--- 所有用户信息 ---" << endl;      for (size\_t i = 0; i < users.size(); ++i)      {          users[i]->displayInfo(i + 1);      }      cout << "--------------------" << endl;  } |

#### ui.h

|  |
| --- |
| #ifndef UI\_H  #define UI\_H  #include <string>  #include <vector>  #include <functional>  #include <memory> // 为shared\_ptr添加  // 前向声明  class User;  class Store;  class Product;  class Customer;  class Order;  class OrderManager;  class OrderItem;  // UI系统主类  class UI  {  private:      // 窗口尺寸      int width = 800;      int height = 600;      // UI状态      bool shouldClose = false;      int currentPage = 0; // 0=主菜单, 1=用户菜单, 2=商城      int userType = 0;    // 0=游客, 1=消费者, 2=商家, 3=管理员      // 商品滚动位置      float scrollY = 0.0f;      // 登录输入缓冲区      char usernameBuffer[128] = {0};      char passwordBuffer[128] = {0};      // 用户信息和商店引用      User \*currentUser = nullptr;      std::vector<User \*> \*users = nullptr;      Store \*store = nullptr;      // 商品搜索      char searchBuffer[128] = {0};      std::vector<Product \*> searchResults;      // 购买商品      char buyProductName[128] = {0};      int buyQuantity = 1;      // 商品管理      char newProductName[128] = {0};      char newProductDesc[256] = {0};      float newProductPrice = 0.0f;      int newProductQuantity = 0;      char newProductAttr1[128] = {0}; // 作者/尺寸/过期日期      char newProductAttr2[128] = {0}; // ISBN/颜色      int newProductType = 0;          // 0=图书, 1=服装, 2=食品      // 充值金额      float depositAmount = 0.0f;      // 订单管理      OrderManager \*orderManager = nullptr;      std::vector<std::shared\_ptr<Order>> pendingOrders;      std::vector<std::shared\_ptr<Order>> completedOrders;      int selectedOrderIndex = -1; // 购物车视图      bool showCartWindow = false; // 订单视图      bool showOrderWindow = false;      // 确认对话框状态      bool showDirectPurchaseConfirm = false;      bool showCartCheckoutConfirm = false;      double confirmTotalAmount = 0.0;      // UI辅助函数      void renderMainMenu();      void renderUserMenu();      void renderStore();      void renderUserRegister();      void renderUserLogin();      void renderStoreCustomer();      void renderStoreSeller();      void renderProductList(const std::vector<Product \*> &products, const std::string &title);      void renderCartWindow();      void renderOrderManagement();      void renderOrderDetails(std::shared\_ptr<Order> order);      void processOrders();      // 加载用户订单      void loadUserOrders();  public:      UI(int width = 800, int height = 600);      ~UI();      // 初始化UI系统      bool init();      // 主循环      void mainLoop(std::vector<User \*> &users, Store &store);      // 是否应该关闭      bool isClosing() const { return shouldClose; }      // 设置当前用户      void setCurrentUser(User \*user);      // 设置订单管理器      void setOrderManager(OrderManager \*manager);      // 获取菜单返回值 (兼容原代码)      int getMenuResult();  };  #endif // UI\_H |

#### ui.cpp

|  |
| --- |
| #include "ui.h"  #include "../user/user.h"  #include "../store/store.h"  #include "../order/order.h"  #include "../ordermanager/ordermanager.h"  #include "../imgui/imgui.h"  #include "../imgui/backends/imgui\_impl\_glfw.h"  #include "../imgui/backends/imgui\_impl\_opengl3.h"  #include <GLFW/glfw3.h>  #include <iostream>  #include <algorithm>  #include <string>  #include <memory>  // 全局GLFW窗口指针  GLFWwindow \*window = nullptr;  // UI构造函数  UI::UI(int width, int height)      : width(width), height(height) {}  // UI析构函数  UI::~UI()  {      // 清理ImGui      ImGui\_ImplOpenGL3\_Shutdown();      ImGui\_ImplGlfw\_Shutdown();      ImGui::DestroyContext();      // 清理GLFW      if (window)      {          glfwDestroyWindow(window);          window = nullptr;      }      glfwTerminate();  }  // 设置订单管理器  void UI::setOrderManager(OrderManager \*manager)  {      orderManager = manager;  }  // 初始化UI  bool UI::init()  {      // 初始化GLFW      if (!glfwInit())      {          std::cerr << "GLFW初始化失败" << std::endl;          return false;      }      // 创建窗口      window = glfwCreateWindow(width, height, "电子商城系统", nullptr, nullptr);      if (!window)      {          std::cerr << "创建窗口失败" << std::endl;          glfwTerminate();          return false;      }      // 设置当前上下文      glfwMakeContextCurrent(window);      glfwSwapInterval(1); // 启用垂直同步      // 初始化ImGui      IMGUI\_CHECKVERSION();      ImGui::CreateContext();      ImGuiIO &io = ImGui::GetIO();      io.ConfigFlags |= ImGuiConfigFlags\_NavEnableKeyboard; // 启用键盘控制      io.Fonts->AddFontFromFileTTF("c:\\Windows\\Fonts\\simhei.ttf", 16.0f, NULL, io.Fonts->GetGlyphRangesChineseFull());      // 设置ImGui风格      ImGui::StyleColorsLight();      // 平台/渲染器后端      ImGui\_ImplGlfw\_InitForOpenGL(window, true);      ImGui\_ImplOpenGL3\_Init("#version 130");      return true;  }  // 设置当前用户  void UI::setCurrentUser(User \*user)  {      currentUser = user;      if (user)      { // 设置用户类型          std::string type = user->getUserType();          if (type == "customer")          {              userType = 1;              // 加载购物车              Customer \*customer = dynamic\_cast<Customer \*>(user);              if (customer)              {                  customer->loadCartFromFile();              }              // 用户登录时不自动显示订单窗口              showOrderWindow = false;          }          else if (type == "seller")              userType = 2;          else if (type == "admin")              userType = 3;          else              userType = 0;      }      else      {          userType = 0;      }  }  // 主循环  void UI::mainLoop(std::vector<User \*> &usersRef, Store &storeRef)  {      users = &usersRef;      store = &storeRef; // 主循环      while (!glfwWindowShouldClose(window) && !shouldClose)      {          glfwPollEvents();          // 开始新帧          ImGui\_ImplOpenGL3\_NewFrame();          ImGui\_ImplGlfw\_NewFrame();          ImGui::NewFrame();          // 全屏窗口          ImGui::SetNextWindowPos(ImVec2(0, 0));          ImGui::SetNextWindowSize(ImVec2(width, height));          ImGui::Begin("商城系统", nullptr,                       ImGuiWindowFlags\_NoTitleBar |                           ImGuiWindowFlags\_NoResize |                           ImGuiWindowFlags\_NoMove |                           ImGuiWindowFlags\_NoScrollbar);          // 处理订单队列          if (orderManager)          {              processOrders();          }          // 根据当前页面渲染不同的UI          switch (currentPage)          {          case 0: // 主菜单              renderMainMenu();              break;          case 1: // 用户菜单              renderUserMenu();              break;          case 2: // 商城              renderStore();              break;          case 3: // 注册页面              renderUserRegister();              break;          case 4: // 登录页面              renderUserLogin();              break;          } // 显示购物车窗口          if (showCartWindow && currentUser && userType == 1)          {              renderCartWindow();          }          // 显示订单管理窗口          if (showOrderWindow && currentUser && userType == 1 && orderManager)          {              renderOrderManagement();          }          ImGui::End();          // 渲染          ImGui::Render();          int displayW, displayH;          glfwGetFramebufferSize(window, &displayW, &displayH);          glViewport(0, 0, displayW, displayH);          glClearColor(0.45f, 0.55f, 0.60f, 1.00f);          glClear(GL\_COLOR\_BUFFER\_BIT);          ImGui\_ImplOpenGL3\_RenderDrawData(ImGui::GetDrawData());          glfwSwapBuffers(window);      }  }  // 渲染主菜单  void UI::renderMainMenu()  {      ImGui::SetWindowFontScale(1.5f);      ImGui::Text("电子商城系统");      ImGui::SetWindowFontScale(1.0f);      ImGui::Separator();      ImGui::Spacing();      ImGui::Spacing();      ImGui::SetCursorPosX((width - 200) / 2);      if (ImGui::Button("注册新用户", ImVec2(200, 50)))      {          currentPage = 3; // 切换到注册页面      }      ImGui::Spacing();      ImGui::SetCursorPosX((width - 200) / 2);      if (ImGui::Button("用户登录", ImVec2(200, 50)))      {          currentPage = 4; // 切换到登录页面          memset(usernameBuffer, 0, sizeof(usernameBuffer));          memset(passwordBuffer, 0, sizeof(passwordBuffer));      }      ImGui::Spacing();      ImGui::SetCursorPosX((width - 200) / 2);      if (ImGui::Button("进入商城 (游客)", ImVec2(200, 50)))      {          currentPage = 2; // 切换到商城      }      ImGui::Spacing();      ImGui::SetCursorPosX((width - 200) / 2);      if (ImGui::Button("退出程序", ImVec2(200, 50)))      {          shouldClose = true;      }  }  // 渲染用户菜单  void UI::renderUserMenu()  {      if (!currentUser)      {          currentPage = 0; // 回到主菜单          return;      }      std::string userTypeStr;      if (userType == 1)          userTypeStr = "消费者";      else if (userType == 2)          userTypeStr = "商家";      else if (userType == 3)          userTypeStr = "管理员";      else          userTypeStr = "未知";      ImGui::SetWindowFontScale(1.5f);      ImGui::Text("%s菜单 - %s", userTypeStr.c\_str(), currentUser->getUsername().c\_str());      ImGui::SetWindowFontScale(1.0f);      ImGui::Separator();      // 用户信息      ImGui::BeginChild("UserInfo", ImVec2(width \* 0.4f, height - 150), true);      ImGui::Text("用户名: %s", currentUser->getUsername().c\_str());      ImGui::Text("用户类型: %s", userTypeStr.c\_str());      ImGui::Text("账户余额: CNY%.2f", currentUser->checkBalance());      ImGui::EndChild();      // 操作区域      ImGui::SameLine();      ImGui::BeginChild("UserActions", ImVec2(0, height - 150), true);      // 修改密码      static char oldPass[128] = {0};      static char newPass[128] = {0};      ImGui::SetNextItemWidth(200);      ImGui::InputText("原密码", oldPass, IM\_ARRAYSIZE(oldPass), ImGuiInputTextFlags\_Password);      ImGui::SetNextItemWidth(200);      ImGui::InputText("新密码", newPass, IM\_ARRAYSIZE(newPass), ImGuiInputTextFlags\_Password);      if (ImGui::Button("修改密码", ImVec2(100, 30)))      {          if (currentUser->changePassword(oldPass, newPass))          {              User::saveUsersToFile(\*users, "user/users.txt");              ImGui::OpenPopup("密码修改成功");              memset(oldPass, 0, sizeof(oldPass));              memset(newPass, 0, sizeof(newPass));          }          else          {              ImGui::OpenPopup("密码修改失败");          }      }      // 弹窗      if (ImGui::BeginPopupModal("密码修改成功", NULL, ImGuiWindowFlags\_AlwaysAutoResize))      {          ImGui::Text("密码修改成功！");          if (ImGui::Button("确定", ImVec2(120, 0)))          {              ImGui::CloseCurrentPopup();          }          ImGui::EndPopup();      }      if (ImGui::BeginPopupModal("密码修改失败", NULL, ImGuiWindowFlags\_AlwaysAutoResize))      {          ImGui::Text("密码修改失败，请检查原密码是否正确。");          if (ImGui::Button("确定", ImVec2(120, 0)))          {              ImGui::CloseCurrentPopup();          }          ImGui::EndPopup();      }      ImGui::Separator();      // 消费者特有 - 充值      if (userType == 1)      {          ImGui::SetNextItemWidth(200);          ImGui::InputFloat("充值金额", &depositAmount, 10.0f, 100.0f, "%.2f");          if (ImGui::Button("充值", ImVec2(100, 30)))          {              if (depositAmount > 0)              {                  if (currentUser->deposit(depositAmount))                  {                      User::saveUsersToFile(\*users, "user/users.txt");                      ImGui::OpenPopup("充值成功");                      depositAmount = 0.0f;                  }              }              else              {                  ImGui::OpenPopup("充值失败");              }          }          if (ImGui::BeginPopupModal("充值成功", NULL, ImGuiWindowFlags\_AlwaysAutoResize))          {              ImGui::Text("充值成功！当前余额: CNY%.2f", currentUser->checkBalance());              if (ImGui::Button("确定", ImVec2(120, 0)))              {                  ImGui::CloseCurrentPopup();              }              ImGui::EndPopup();          }          if (ImGui::BeginPopupModal("充值失败", NULL, ImGuiWindowFlags\_AlwaysAutoResize))          {              ImGui::Text("充值失败，请确保充值金额大于0。");              if (ImGui::Button("确定", ImVec2(120, 0)))              {                  ImGui::CloseCurrentPopup();              }              ImGui::EndPopup();          }      }      ImGui::EndChild();      // 底部按钮      ImGui::Separator();      if (ImGui::Button("进入商城", ImVec2(120, 40)))      {          currentPage = 2; // 切换到商城          searchResults.clear();          memset(searchBuffer, 0, sizeof(searchBuffer));      }      ImGui::SameLine();      if (ImGui::Button("退出登录", ImVec2(120, 40)))      {          currentUser = nullptr;          userType = 0;          currentPage = 0; // 回到主菜单      }  }  // 渲染商城  void UI::renderStore()  {      ImGui::SetWindowFontScale(1.5f);      if (currentUser)      {          if (userType == 2)          {              ImGui::Text("商城管理 - %s", currentUser->getUsername().c\_str());          }          else          {              ImGui::Text("商城 - %s", currentUser->getUsername().c\_str());          }      }      else      {          ImGui::Text("商城 - 游客模式");      }      ImGui::SetWindowFontScale(1.0f);      ImGui::Separator();      // 根据用户类型显示不同的商城界面      if (userType == 2)      { // 商家          renderStoreSeller();      }      else      { // 消费者或游客          renderStoreCustomer();      }      // 底部按钮      ImGui::Separator();      if (ImGui::Button("返回", ImVec2(120, 40)))      {          if (currentUser)          {              currentPage = 1; // 返回用户菜单          }          else          {              currentPage = 0; // 返回主菜单          }      }  }  // 渲染注册页面  void UI::renderUserRegister()  {      ImGui::SetWindowFontScale(1.5f);      ImGui::Text("用户注册");      ImGui::SetWindowFontScale(1.0f);      ImGui::Separator();      static char regUsername[128] = {0};      static char regPassword[128] = {0};      static int regUserType = 0; // 0=消费者, 1=商家      ImGui::SetNextItemWidth(250);      ImGui::InputText("用户名", regUsername, IM\_ARRAYSIZE(regUsername));      ImGui::SetNextItemWidth(250);      ImGui::InputText("密码", regPassword, IM\_ARRAYSIZE(regPassword), ImGuiInputTextFlags\_Password);      ImGui::RadioButton("消费者", &regUserType, 0);      ImGui::SameLine();      ImGui::RadioButton("商家", &regUserType, 1);      if (ImGui::Button("注册", ImVec2(120, 40)))      {          if (strlen(regUsername) == 0)          {              ImGui::OpenPopup("注册失败");          }          else if (User::isUsernameExists(\*users, regUsername))          {              ImGui::OpenPopup("用户名已存在");          }          else          {              User \*newUser = nullptr;              if (regUserType == 0)              {                  extern const std::string CART\_FILE; // 引用外部声明的购物车目录常量                  newUser = new Customer(regUsername, regPassword, CART\_FILE);              }              else              {                  newUser = new Seller(regUsername, regPassword);              }              users->push\_back(newUser);              User::saveUsersToFile(\*users, "user/users.txt");              currentUser = newUser;              setCurrentUser(newUser);              currentPage = 1; // 跳转至用户菜单              // 清空输入              memset(regUsername, 0, sizeof(regUsername));              memset(regPassword, 0, sizeof(regPassword));          }      }      ImGui::SameLine();      if (ImGui::Button("返回", ImVec2(120, 40)))      {          currentPage = 0; // 回到主菜单      }      // 弹窗      if (ImGui::BeginPopupModal("注册失败", NULL, ImGuiWindowFlags\_AlwaysAutoResize))      {          ImGui::Text("注册失败，用户名不能为空！");          if (ImGui::Button("确定", ImVec2(120, 0)))          {              ImGui::CloseCurrentPopup();          }          ImGui::EndPopup();      }      if (ImGui::BeginPopupModal("用户名已存在", NULL, ImGuiWindowFlags\_AlwaysAutoResize))      {          ImGui::Text("该用户名已存在，请选择其他用户名！");          if (ImGui::Button("确定", ImVec2(120, 0)))          {              ImGui::CloseCurrentPopup();          }          ImGui::EndPopup();      }  }  // 渲染登录页面  void UI::renderUserLogin()  {      ImGui::SetWindowFontScale(1.5f);      ImGui::Text("用户登录");      ImGui::SetWindowFontScale(1.0f);      ImGui::Separator();      ImGui::SetNextItemWidth(250);      ImGui::InputText("用户名", usernameBuffer, IM\_ARRAYSIZE(usernameBuffer));      ImGui::SetNextItemWidth(250);      ImGui::InputText("密码", passwordBuffer, IM\_ARRAYSIZE(passwordBuffer), ImGuiInputTextFlags\_Password);      if (ImGui::Button("登录", ImVec2(120, 40)))      {          User \*user = User::login(\*users, usernameBuffer, passwordBuffer);          if (user)          {              currentUser = user;              setCurrentUser(user);              currentPage = 1; // 跳转至用户菜单          }          else          {              ImGui::OpenPopup("登录失败");          }      }      ImGui::SameLine();      if (ImGui::Button("返回", ImVec2(120, 40)))      {          currentPage = 0; // 回到主菜单      }      // 弹窗      if (ImGui::BeginPopupModal("登录失败", NULL, ImGuiWindowFlags\_AlwaysAutoResize))      {          ImGui::Text("登录失败，用户名或密码错误！");          if (ImGui::Button("确定", ImVec2(120, 0)))          {              ImGui::CloseCurrentPopup();          }          ImGui::EndPopup();      }  }  // 渲染消费者商城  void UI::renderStoreCustomer()  {      // 增加高度以容纳购物车和订单按钮      ImGui::BeginChild("StoreControls", ImVec2(width, 100));      ImGui::SetNextItemWidth(300);      if (ImGui::InputText("搜索商品", searchBuffer, IM\_ARRAYSIZE(searchBuffer), ImGuiInputTextFlags\_EnterReturnsTrue))      {          searchResults = store->searchProductsByName(searchBuffer);      }      ImGui::SameLine();      if (ImGui::Button("搜索", ImVec2(80, 0)))      {          searchResults = store->searchProductsByName(searchBuffer);      }      ImGui::SameLine();      if (ImGui::Button("显示所有商品", ImVec2(150, 0)))      {          searchResults.clear();          memset(searchBuffer, 0, sizeof(searchBuffer));      }      // 将购物车和订单按钮放在新的一行      if (currentUser && userType == 1)      {          ImGui::Spacing();          ImGui::Spacing(); // 增加一点垂直间距          // 居中放置按钮          float buttonsWidth = 150.0f + 20.0f + 150.0f; // 两个按钮加上它们之间的间距          ImGui::SetCursorPosX((width - buttonsWidth) \* 0.5f);          if (ImGui::Button("查看购物车", ImVec2(150, 30))) // 增加按钮高度使其更明显          {              showCartWindow = true;          }          ImGui::SameLine(0, 20.0f);                      // 设置按钮之间的间距为20          if (ImGui::Button("查看订单", ImVec2(150, 30))) // 和购物车按钮保持一致的大小          {              // 加载订单数据              if (orderManager)              {                  // 加载用户订单                  loadUserOrders();                  // 设置显示订单窗口标志                  showOrderWindow = true;              }          }      }      ImGui::EndChild();      ImGui::Separator();      // 商品列表      if (!searchResults.empty())      {          renderProductList(searchResults, "搜索结果");      }      else      {          renderProductList(store->getProducts(), "所有商品");      } // 购买商品区域      if (currentUser && userType == 1)      {          ImGui::Separator();          ImGui::Text("购买商品");          ImGui::SetNextItemWidth(250);          ImGui::InputText("商品名称", buyProductName, IM\_ARRAYSIZE(buyProductName));          ImGui::SetNextItemWidth(100);          ImGui::InputInt("购买数量", &buyQuantity);          if (buyQuantity < 1)              buyQuantity = 1;          if (ImGui::Button("直接购买", ImVec2(100, 30)))          {              if (strlen(buyProductName) == 0)              {                  ImGui::OpenPopup("购买失败");              }              else              {                  Product \*product = store->findProductByName(buyProductName);                  if (!product)                  {                      ImGui::OpenPopup("商品不存在");                  }                  else if (product->getQuantity() < buyQuantity)                  {                      ImGui::OpenPopup("库存不足");                  }                  else if (currentUser->checkBalance() < product->getPrice() \* buyQuantity)                  {                      ImGui::OpenPopup("余额不足");                  }                  else                  {                      // 设置确认对话框信息并显示                      confirmTotalAmount = product->getPrice() \* buyQuantity;                      showDirectPurchaseConfirm = true;                  }              }          }          ImGui::SameLine();          if (ImGui::Button("加入购物车", ImVec2(120, 30)))          {              if (strlen(buyProductName) == 0)              {                  ImGui::OpenPopup("购买失败");              }              else              {                  Product \*product = store->findProductByName(buyProductName);                  if (!product)                  {                      ImGui::OpenPopup("商品不存在");                  }                  else if (product->getQuantity() < buyQuantity)                  {                      ImGui::OpenPopup("库存不足");                  }                  else                  {                      // 转换为Customer\*                      Customer \*customer = dynamic\_cast<Customer \*>(currentUser);                      if (customer)                      {                          // 添加到购物车                          if (customer->addToCart(\*product, buyQuantity))                          {                              ImGui::OpenPopup("加入购物车成功");                              memset(buyProductName, 0, sizeof(buyProductName));                              buyQuantity = 1;                          }                          else                          {                              ImGui::OpenPopup("加入购物车失败");                          }                      }                  }              }          }          // 弹窗          if (ImGui::BeginPopupModal("购买失败", NULL, ImGuiWindowFlags\_AlwaysAutoResize))          {              ImGui::Text("请输入要购买的商品名称！");              if (ImGui::Button("确定", ImVec2(120, 0)))              {                  ImGui::CloseCurrentPopup();              }              ImGui::EndPopup();          }          if (ImGui::BeginPopupModal("商品不存在", NULL, ImGuiWindowFlags\_AlwaysAutoResize))          {              ImGui::Text("找不到商品 \"%s\"！", buyProductName);              if (ImGui::Button("确定", ImVec2(120, 0)))              {                  ImGui::CloseCurrentPopup();              }              ImGui::EndPopup();          }          if (ImGui::BeginPopupModal("库存不足", NULL, ImGuiWindowFlags\_AlwaysAutoResize))          {              Product \*product = store->findProductByName(buyProductName);              if (product)              {                  ImGui::Text("库存不足！当前库存: %d", product->getQuantity());              }              if (ImGui::Button("确定", ImVec2(120, 0)))              {                  ImGui::CloseCurrentPopup();              }              ImGui::EndPopup();          }          if (ImGui::BeginPopupModal("余额不足", NULL, ImGuiWindowFlags\_AlwaysAutoResize))          {              Product \*product = store->findProductByName(buyProductName);              if (product)              {                  double totalCost = product->getPrice() \* buyQuantity;                  ImGui::Text("余额不足！需要: CNY%.2f, 当前余额: CNY%.2f",                              totalCost, currentUser->checkBalance());              }              if (ImGui::Button("确定", ImVec2(120, 0)))              {                  ImGui::CloseCurrentPopup();              }              ImGui::EndPopup();          }          if (ImGui::BeginPopupModal("订单提交成功", NULL, ImGuiWindowFlags\_AlwaysAutoResize))          {              ImVec2 center = ImVec2(width \* 0.5f, height \* 0.5f);              ImGui::Text("购买订单提交成功，请等待处理。");              if (ImGui::Button("确定", ImVec2(120, 0)))              {                  ImGui::CloseCurrentPopup();                  showCartWindow = false;              }              ImGui::EndPopup();          }          if (ImGui::BeginPopupModal("加入购物车成功", NULL, ImGuiWindowFlags\_AlwaysAutoResize))          {              ImGui::Text("成功加入购物车！");              if (ImGui::Button("查看购物车", ImVec2(120, 0)))              {                  showCartWindow = true;                  ImGui::CloseCurrentPopup();              }              ImGui::SameLine();              if (ImGui::Button("继续购物", ImVec2(120, 0)))              {                  ImGui::CloseCurrentPopup();              }              ImGui::EndPopup();          }          if (ImGui::BeginPopupModal("加入购物车失败", NULL, ImGuiWindowFlags\_AlwaysAutoResize))          {              ImGui::Text("加入购物车失败，请稍后重试。");              if (ImGui::Button("确定", ImVec2(120, 0)))              {                  ImGui::CloseCurrentPopup();              }              ImGui::EndPopup();          }      }      // 直接购买确认对话框      if (showDirectPurchaseConfirm)      {          // 设置对话框居中显示          ImVec2 center = ImVec2(width \* 0.5f, height \* 0.5f);          ImGui::SetNextWindowPos(center, ImGuiCond\_Always, ImVec2(0.5f, 0.5f));          if (ImGui::BeginPopupModal("确认直接购买", NULL, ImGuiWindowFlags\_AlwaysAutoResize))          {              Product \*product = store->findProductByName(buyProductName);              if (product)              {                  ImGui::Text("订单详情:");                  ImGui::Separator();                  ImGui::Text("商品名称: %s", buyProductName);                  ImGui::Text("购买数量: %d", buyQuantity);                  ImGui::Text("单价: CNY %.2f", product->getPrice());                  ImGui::Text("总金额: CNY %.2f", confirmTotalAmount);                  ImGui::Separator();                  ImGui::Text("您的余额: CNY %.2f", currentUser->checkBalance());                  ImGui::Text("支付后余额: CNY %.2f", currentUser->checkBalance() - confirmTotalAmount);                  ImGui::Separator();                  if (ImGui::Button("确认购买", ImVec2(120, 30)))                  {                      // 执行购买逻辑                      Order order(currentUser->getUsername());                      order.addItemFromProduct(\*product, buyQuantity);                      if (orderManager)                      {                          std::shared\_ptr<Order> orderPtr = orderManager->submitOrderRequest(order);                          if (orderPtr)                          {                              ImGui::OpenPopup("订单提交成功");                              memset(buyProductName, 0, sizeof(buyProductName));                              buyQuantity = 1;                          }                      }                      showDirectPurchaseConfirm = false;                      ImGui::CloseCurrentPopup();                  }                  ImGui::SameLine();                  if (ImGui::Button("取消", ImVec2(120, 30)))                  {                      showDirectPurchaseConfirm = false;                      ImGui::CloseCurrentPopup();                  }              }              else              {                  ImGui::Text("商品信息获取失败");                  if (ImGui::Button("确定", ImVec2(120, 30)))                  {                      showDirectPurchaseConfirm = false;                      ImGui::CloseCurrentPopup();                  }              }              ImGui::EndPopup();          }          else          {              // 如果对话框没有打开，则打开它              ImGui::OpenPopup("确认直接购买");          }      }  }  // 渲染商家商城  void UI::renderStoreSeller()  {      if (!currentUser || userType != 2)          return;      std::string sellerName = currentUser->getUsername();      // 显示选项卡      static int currentTab = 0;      ImGui::BeginTabBar("SellerTabs");      if (ImGui::BeginTabItem("我的商品"))      {          ImGui::BeginChild("MyProducts", ImVec2(0, height - 200));          renderProductList(store->getSellerProducts(sellerName), "我的商品");          ImGui::EndChild();          ImGui::EndTabItem();      }      if (ImGui::BeginTabItem("添加商品"))      {          ImGui::BeginChild("AddProduct", ImVec2(0, height - 200));          ImGui::SetNextItemWidth(250);          ImGui::InputText("商品名称", newProductName, IM\_ARRAYSIZE(newProductName));          ImGui::SetNextItemWidth(350);          ImGui::InputTextMultiline("商品描述", newProductDesc, IM\_ARRAYSIZE(newProductDesc));          ImGui::SetNextItemWidth(150);          ImGui::InputFloat("商品价格", &newProductPrice, 1.0f, 10.0f, "CNY%.2f");          if (newProductPrice < 0)              newProductPrice = 0;          ImGui::SetNextItemWidth(150);          ImGui::InputInt("库存数量", &newProductQuantity);          if (newProductQuantity < 0)              newProductQuantity = 0;          const char \*productTypes[] = {"图书", "服装", "食品", "其他"};          ImGui::SetNextItemWidth(150);          ImGui::Combo("商品类型", &newProductType, productTypes, IM\_ARRAYSIZE(productTypes));          // 根据不同的商品类型显示不同的输入字段          if (newProductType == 0)          { // 图书              ImGui::SetNextItemWidth(250);              ImGui::InputText("作者", newProductAttr1, IM\_ARRAYSIZE(newProductAttr1));              ImGui::SetNextItemWidth(250);              ImGui::InputText("ISBN", newProductAttr2, IM\_ARRAYSIZE(newProductAttr2));          }          else if (newProductType == 1)          { // 服装              ImGui::SetNextItemWidth(150);              ImGui::InputText("尺寸", newProductAttr1, IM\_ARRAYSIZE(newProductAttr1));              ImGui::SetNextItemWidth(150);              ImGui::InputText("颜色", newProductAttr2, IM\_ARRAYSIZE(newProductAttr2));          }          else if (newProductType == 2)          { // 食品              ImGui::SetNextItemWidth(150);              ImGui::InputText("过期日期", newProductAttr1, IM\_ARRAYSIZE(newProductAttr1));          }          else          {              ImGui::SetNextItemWidth(250);              ImGui::InputText("分类名称", newProductAttr1, IM\_ARRAYSIZE(newProductAttr1));          }          if (ImGui::Button("添加商品", ImVec2(120, 30)))          {              if (strlen(newProductName) == 0)              {                  ImGui::OpenPopup("商品添加失败");              }              else              {                  bool success = false;                  if (newProductType == 0)                  { // 图书                      success = store->createBook(                          currentUser, newProductName, newProductDesc,                          newProductPrice, newProductQuantity,                          newProductAttr1, newProductAttr2);                  }                  else if (newProductType == 1)                  { // 服装                      success = store->createClothing(                          currentUser, newProductName, newProductDesc,                          newProductPrice, newProductQuantity,                          newProductAttr1, newProductAttr2);                  }                  else if (newProductType == 2)                  { // 食品                      success = store->createFood(                          currentUser, newProductName, newProductDesc,                          newProductPrice, newProductQuantity,                          newProductAttr1);                  }                  else                  { // 其他                      success = store->createGenericProduct(                          currentUser, newProductName, newProductDesc,                          newProductPrice, newProductQuantity,                          newProductAttr1);                  }                  if (success)                  {                      ImGui::OpenPopup("商品添加成功");                      memset(newProductName, 0, sizeof(newProductName));                      memset(newProductDesc, 0, sizeof(newProductDesc));                      memset(newProductAttr1, 0, sizeof(newProductAttr1));                      memset(newProductAttr2, 0, sizeof(newProductAttr2));                      newProductPrice = 0.0f;                      newProductQuantity = 0;                  }                  else                  {                      ImGui::OpenPopup("商品添加失败");                  }              }          }          if (ImGui::BeginPopupModal("商品添加成功", NULL, ImGuiWindowFlags\_AlwaysAutoResize))          {              ImGui::Text("商品添加成功！");              if (ImGui::Button("确定", ImVec2(120, 0)))              {                  ImGui::CloseCurrentPopup();              }              ImGui::EndPopup();          }          ImGui::EndChild();          ImGui::EndTabItem();      }      if (ImGui::BeginTabItem("管理商品"))      {          ImGui::BeginChild("ManageProducts", ImVec2(0, height - 200));          ImGui::Text("修改商品价格");          static char priceProductName[128] = {0};          static float newPrice = 0.0f;          ImGui::SetNextItemWidth(250);          ImGui::InputText("商品名称##price", priceProductName, IM\_ARRAYSIZE(priceProductName));          ImGui::SetNextItemWidth(150);          ImGui::InputFloat("新价格", &newPrice, 1.0f, 10.0f, "CNY%.2f");          if (newPrice < 0)              newPrice = 0;          if (ImGui::Button("修改价格", ImVec2(120, 30)))          {              if (strlen(priceProductName) == 0)              {                  ImGui::OpenPopup("修改失败");              }              else              {                  if (store->manageProductPrice(currentUser, priceProductName, newPrice))                  {                      ImGui::OpenPopup("价格修改成功");                      memset(priceProductName, 0, sizeof(priceProductName));                      newPrice = 0.0f;                  }                  else                  {                      ImGui::OpenPopup("价格修改失败");                  }              }          }          if (ImGui::BeginPopupModal("修改失败", NULL, ImGuiWindowFlags\_AlwaysAutoResize))          {              ImGui::Text("请输入商品名称！");              if (ImGui::Button("确定", ImVec2(120, 0)))              {                  ImGui::CloseCurrentPopup();              }              ImGui::EndPopup();          }          if (ImGui::BeginPopupModal("价格修改成功", NULL, ImGuiWindowFlags\_AlwaysAutoResize))          {              ImGui::Text("商品价格修改成功！");              if (ImGui::Button("确定", ImVec2(120, 0)))              {                  ImGui::CloseCurrentPopup();              }              ImGui::EndPopup();          }          if (ImGui::BeginPopupModal("价格修改失败", NULL, ImGuiWindowFlags\_AlwaysAutoResize))          {              ImGui::Text("价格修改失败！请确认商品名称正确且为您所有。");              if (ImGui::Button("确定", ImVec2(120, 0)))              {                  ImGui::CloseCurrentPopup();              }              ImGui::EndPopup();          }          ImGui::Separator();          ImGui::Text("修改商品库存");          static char qtyProductName[128] = {0};          static int newQty = 0;          ImGui::SetNextItemWidth(250);          ImGui::InputText("商品名称##qty", qtyProductName, IM\_ARRAYSIZE(qtyProductName));          ImGui::SetNextItemWidth(150);          ImGui::InputInt("新库存", &newQty);          if (newQty < 0)              newQty = 0;          if (ImGui::Button("修改库存", ImVec2(120, 30)))          {              if (strlen(qtyProductName) == 0)              {                  ImGui::OpenPopup("修改失败");              }              else              {                  if (store->manageProductQuantity(currentUser, qtyProductName, newQty))                  {                      ImGui::OpenPopup("库存修改成功");                      memset(qtyProductName, 0, sizeof(qtyProductName));                      newQty = 0;                  }                  else                  {                      ImGui::OpenPopup("库存修改失败");                  }              }          }          if (ImGui::BeginPopupModal("库存修改成功", NULL, ImGuiWindowFlags\_AlwaysAutoResize))          {              ImGui::Text("商品库存修改成功！");              if (ImGui::Button("确定", ImVec2(120, 0)))              {                  ImGui::CloseCurrentPopup();              }              ImGui::EndPopup();          }          if (ImGui::BeginPopupModal("库存修改失败", NULL, ImGuiWindowFlags\_AlwaysAutoResize))          {              ImGui::Text("库存修改失败！请确认商品名称正确且为您所有。");              if (ImGui::Button("确定", ImVec2(120, 0)))              {                  ImGui::CloseCurrentPopup();              }              ImGui::EndPopup();          }          ImGui::Separator();          ImGui::Text("修改商品折扣");          static char discountProductName[128] = {0};          static float newDiscount = 0.0f;          ImGui::SetNextItemWidth(250);          ImGui::InputText("商品名称##discount", discountProductName, IM\_ARRAYSIZE(discountProductName));          ImGui::SetNextItemWidth(150);          ImGui::SliderFloat("折扣 (%)", &newDiscount, 0.0f, 100.0f, "%.1f%%");          if (ImGui::Button("应用折扣", ImVec2(120, 30)))          {              if (strlen(discountProductName) == 0)              {                  ImGui::OpenPopup("修改失败");              }              else              {                  if (store->manageProductDiscount(currentUser, discountProductName, newDiscount / 100.0f))                  {                      ImGui::OpenPopup("折扣修改成功");                      memset(discountProductName, 0, sizeof(discountProductName));                      newDiscount = 0.0f;                  }                  else                  {                      ImGui::OpenPopup("折扣修改失败");                  }              }          }          if (ImGui::BeginPopupModal("折扣修改成功", NULL, ImGuiWindowFlags\_AlwaysAutoResize))          {              ImGui::Text("商品折扣修改成功！");              if (ImGui::Button("确定", ImVec2(120, 0)))              {                  ImGui::CloseCurrentPopup();              }              ImGui::EndPopup();          }          if (ImGui::BeginPopupModal("折扣修改失败", NULL, ImGuiWindowFlags\_AlwaysAutoResize))          {              ImGui::Text("折扣修改失败！请确认商品名称正确且为您所有。");              if (ImGui::Button("确定", ImVec2(120, 0)))              {                  ImGui::CloseCurrentPopup();              }              ImGui::EndPopup();          }          ImGui::EndChild();          ImGui::EndTabItem();      }      ImGui::EndTabBar();  }  // 渲染商品列表  void UI::renderProductList(const std::vector<Product \*> &products, const std::string &title)  {      ImGui::Text("%s (%d 件商品)", title.c\_str(), products.size());      if (products.empty())      {          ImGui::TextColored(ImVec4(0.7f, 0.7f, 0.7f, 1.0f), "没有商品");          return;      }      // 商品表格      if (ImGui::BeginTable("ProductTable", 5, ImGuiTableFlags\_Borders | ImGuiTableFlags\_RowBg))      {          ImGui::TableSetupColumn("名称");          ImGui::TableSetupColumn("描述");          ImGui::TableSetupColumn("价格");          ImGui::TableSetupColumn("库存");          ImGui::TableSetupColumn("商家");          ImGui::TableHeadersRow();          for (const auto &product : products)          {              ImGui::TableNextRow();              ImGui::TableSetColumnIndex(0);              ImGui::Text("%s", product->getName().c\_str());              ImGui::TableSetColumnIndex(1);              ImGui::TextWrapped("%s", product->getDescription().c\_str());              ImGui::TableSetColumnIndex(2);              if (product->getDiscountRate() > 0)              {                  ImGui::Text("CNY%.2f (%.0f%%折)", product->getPrice(),                              (1.0 - product->getDiscountRate()) \* 100);              }              else              {                  ImGui::Text("CNY%.2f", product->getPrice());              }              ImGui::TableSetColumnIndex(3);              ImGui::Text("%d", product->getQuantity());              ImGui::TableSetColumnIndex(4);              ImGui::Text("%s", product->getSellerUsername().c\_str());          }          ImGui::EndTable();      }  }  // 获取菜单结果 (兼容原代码)  int UI::getMenuResult()  {      if (shouldClose)          return 0; // 退出程序      if (currentPage == 2)          return 3; // 进入商城      if (!currentUser)          return 1; // 继续主循环      return 1;     // 默认继续主循环  }  // 渲染购物车窗口  void UI::renderCartWindow()  {      Customer \*customer = dynamic\_cast<Customer \*>(currentUser);      if (!customer)          return;      ImGui::SetNextWindowPos(ImVec2(width / 2 - 300, height / 2 - 200));      ImGui::SetNextWindowSize(ImVec2(600, 400));      if (ImGui::Begin("购物车", &showCartWindow, ImGuiWindowFlags\_NoCollapse))      {          if (customer->shoppingCartItems.empty())          {              ImGui::TextColored(ImVec4(0.7f, 0.7f, 0.7f, 1.0f), "购物车为空");          }          else          {              // 购物车表格              if (ImGui::BeginTable("CartTable", 6, ImGuiTableFlags\_Borders | ImGuiTableFlags\_RowBg))              {                  ImGui::TableSetupColumn("商品名称");                  ImGui::TableSetupColumn("数量");                  ImGui::TableSetupColumn("单价");                  ImGui::TableSetupColumn("小计");                  ImGui::TableSetupColumn("商家");                  ImGui::TableSetupColumn("操作");                  ImGui::TableHeadersRow();                  double totalAmount = 0.0;                  std::vector<int> itemsToRemove;                  int itemIndex = 0;                  // 显示购物车中的商品                  for (auto &item : customer->shoppingCartItems)                  {                      ImGui::TableNextRow();                      ImGui::TableSetColumnIndex(0);                      ImGui::Text("%s", item.productName.c\_str());                      ImGui::TableSetColumnIndex(1);                      ImGui::PushID(itemIndex);                      int quantity = item.quantity;                      if (ImGui::InputInt("##qty", &quantity, 1, 5))                      {                          if (quantity > 0)                          {                              // 检查库存                              Product \*product = store->findProductByName(item.productName, item.sellerUsername);                              if (product && product->getQuantity() >= quantity)                              {                                  item.quantity = quantity;                                  customer->saveCartToFile();                              }                          }                      }                      ImGui::PopID();                      ImGui::TableSetColumnIndex(2);                      ImGui::Text("CNY %.2f", item.priceAtAddition);                      ImGui::TableSetColumnIndex(3);                      double subtotal = item.priceAtAddition \* item.quantity;                      totalAmount += subtotal;                      ImGui::Text("CNY %.2f", subtotal);                      ImGui::TableSetColumnIndex(4);                      ImGui::Text("%s", item.sellerUsername.c\_str());                      ImGui::TableSetColumnIndex(5);                      ImGui::PushID(1000 + itemIndex);                      if (ImGui::Button("删除"))                      {                          itemsToRemove.push\_back(itemIndex);                      }                      ImGui::PopID();                      itemIndex++;                  }                  ImGui::EndTable();                  // 处理要删除的商品                  for (int i = itemsToRemove.size() - 1; i >= 0; i--)                  {                      int idx = itemsToRemove[i];                      if (idx >= 0 && idx < customer->shoppingCartItems.size())                      {                          customer->removeCartItem(customer->shoppingCartItems[idx].productName);                      }                  }                  // 显示总计                  ImGui::Separator();                  ImGui::Text("总计: CNY %.2f", totalAmount); // 结算按钮                  if (ImGui::Button("结算", ImVec2(120, 30)))                  {                      if (totalAmount > 0)                      {                          // 检查用户余额                          if (customer->checkBalance() < totalAmount)                          {                              ImGui::OpenPopup("余额不足");                          }                          else                          {                              // 验证所有商品库存                              bool allItemsAvailable = true;                              for (auto &item : customer->shoppingCartItems)                              {                                  Product \*product = store->findProductByName(item.productName, item.sellerUsername);                                  if (!product || product->getQuantity() < item.quantity)                                  {                                      allItemsAvailable = false;                                      break;                                  }                              }                              if (!allItemsAvailable)                              {                                  ImGui::OpenPopup("库存不足");                              }                              else                              {                                  // 设置确认对话框信息并显示                                  confirmTotalAmount = totalAmount;                                  showCartCheckoutConfirm = true;                              }                          }                      }                  }                  // 弹窗                  if (ImGui::BeginPopupModal("余额不足", NULL, ImGuiWindowFlags\_AlwaysAutoResize))                  {                      ImGui::Text("您的余额不足，需要充值。当前余额: CNY%.2f, 订单金额: CNY%.2f",                                  customer->checkBalance(), totalAmount);                      if (ImGui::Button("确定", ImVec2(120, 0)))                      {                          ImGui::CloseCurrentPopup();                      }                      ImGui::EndPopup();                  }                  if (ImGui::BeginPopupModal("库存不足", NULL, ImGuiWindowFlags\_AlwaysAutoResize))                  {                      ImGui::Text("部分商品库存不足，请调整购买数量。");                      if (ImGui::Button("确定", ImVec2(120, 0)))                      {                          ImGui::CloseCurrentPopup();                      }                      ImGui::EndPopup();                  }                  if (ImGui::BeginPopupModal("订单提交成功", NULL, ImGuiWindowFlags\_AlwaysAutoResize))                  {                      ImGui::Text("订单提交成功，请等待处理。");                      if (ImGui::Button("确定", ImVec2(120, 0)))                      {                          ImGui::CloseCurrentPopup();                          showCartWindow = false;                      }                      ImGui::EndPopup();                  }              }              ImGui::Separator();              // 继续购物按钮              if (ImGui::Button("继续购物", ImVec2(120, 30)))              {                  showCartWindow = false;              }              ImGui::SameLine();              // 清空购物车按钮              if (ImGui::Button("清空购物车", ImVec2(120, 30)))              {                  if (!customer->shoppingCartItems.empty())                  {                      ImGui::OpenPopup("确认清空");                  }              }              if (ImGui::BeginPopupModal("确认清空", NULL, ImGuiWindowFlags\_AlwaysAutoResize))              {                  ImGui::Text("确定要清空购物车吗？");                  if (ImGui::Button("确定", ImVec2(120, 0)))                  {                      customer->clearCartAndFile();                      ImGui::CloseCurrentPopup();                  }                  ImGui::SameLine();                  if (ImGui::Button("取消", ImVec2(120, 0)))                  {                      ImGui::CloseCurrentPopup();                  }                  ImGui::EndPopup();              }          }          ImGui::End();      }      // 购物车结账确认对话框      if (showCartCheckoutConfirm && customer)      {          // 设置对话框居中显示          ImVec2 center = ImVec2(width \* 0.5f, height \* 0.5f);          ImGui::SetNextWindowPos(center, ImGuiCond\_Always, ImVec2(0.5f, 0.5f));          if (ImGui::BeginPopupModal("确认购物车结账", NULL, ImGuiWindowFlags\_AlwaysAutoResize))          {              ImGui::Text("订单详情:");              ImGui::Separator();              // 显示购物车商品明细              if (ImGui::BeginTable("CheckoutTable", 4, ImGuiTableFlags\_Borders | ImGuiTableFlags\_RowBg))              {                  ImGui::TableSetupColumn("商品名称", ImGuiTableColumnFlags\_WidthFixed, 150);                  ImGui::TableSetupColumn("数量", ImGuiTableColumnFlags\_WidthFixed, 60);                  ImGui::TableSetupColumn("单价", ImGuiTableColumnFlags\_WidthFixed, 80);                  ImGui::TableSetupColumn("小计", ImGuiTableColumnFlags\_WidthFixed, 80);                  ImGui::TableHeadersRow();                  for (const auto &item : customer->shoppingCartItems)                  {                      ImGui::TableNextRow();                      ImGui::TableSetColumnIndex(0);                      ImGui::TextUnformatted(item.productName.c\_str());                      ImGui::TableSetColumnIndex(1);                      ImGui::Text("%d", item.quantity);                      ImGui::TableSetColumnIndex(2);                      ImGui::Text("CNY %.2f", item.priceAtAddition);                      ImGui::TableSetColumnIndex(3);                      ImGui::Text("CNY %.2f", item.priceAtAddition \* item.quantity);                  }                  ImGui::EndTable();              }              ImGui::Separator();              ImGui::Text("总金额: CNY %.2f", confirmTotalAmount);              ImGui::Text("您的余额: CNY %.2f", customer->checkBalance());              ImGui::Text("支付后余额: CNY %.2f", customer->checkBalance() - confirmTotalAmount);              ImGui::Separator();              if (ImGui::Button("确认结账", ImVec2(120, 30)))              {                  // 执行结账逻辑                  Order order(customer->getUsername());                  // 添加订单项                  for (auto &item : customer->shoppingCartItems)                  {                      Product \*product = store->findProductByName(item.productName, item.sellerUsername);                      if (product)                      {                          order.addItemFromProduct(\*product, item.quantity);                      }                  }                  // 提交订单                  if (orderManager)                  {                      std::shared\_ptr<Order> orderPtr = orderManager->submitOrderRequest(order);                      if (orderPtr)                      {                          // 清空购物车                          customer->clearCartAndFile();                          // 弹出成功消息                          ImGui::OpenPopup("订单提交成功");                      }                  }                  showCartCheckoutConfirm = false;                  ImGui::CloseCurrentPopup();              }              ImGui::SameLine();              if (ImGui::Button("取消", ImVec2(120, 30)))              {                  showCartCheckoutConfirm = false;                  ImGui::CloseCurrentPopup();              }              ImGui::EndPopup();          }          else          {              // 如果对话框没有打开，则打开它              ImGui::OpenPopup("确认购物车结账");          }      }  }  // 渲染订单管理窗口  void UI::renderOrderManagement()  {      if (!currentUser || userType != 1 || !orderManager)          return;      Customer \*customer = dynamic\_cast<Customer \*>(currentUser);      if (!customer)          return;      // 窗口定位居中，方便用户查看      ImGui::SetNextWindowPos(ImVec2(width / 2 - 300, height / 2 - 200), ImGuiCond\_Once);      ImGui::SetNextWindowSize(ImVec2(600, 400), ImGuiCond\_Once);      if (ImGui::Begin("我的订单", &showOrderWindow, ImGuiWindowFlags\_NoCollapse))      {          static bool showPending = true;          static bool showCompleted = true;          ImGui::Checkbox("显示待处理订单", &showPending);          ImGui::SameLine();          ImGui::Checkbox("显示已完成订单", &showCompleted);          // 订单表格高度          float tableHeight = 250;          // 如果点击刷新按钮，则重新加载订单数据          if (ImGui::Button("刷新订单列表", ImVec2(150, 30)))          {              // 重新加载用户订单              loadUserOrders();          }          ImGui::Separator();          // 创建一个子窗口来显示订单列表          ImGui::BeginChild("OrdersList", ImVec2(0, tableHeight), true); // 显示订单列表          if (showPending && !pendingOrders.empty())          {              if (ImGui::CollapsingHeader("待处理订单", ImGuiTreeNodeFlags\_DefaultOpen))              {                  for (int i = 0; i < pendingOrders.size(); i++)                  {                      std::string displayName = "订单 " + pendingOrders[i]->getOrderId().substr(8) + "...";                      if (ImGui::Selectable(displayName.c\_str(), selectedOrderIndex == i))                      {                          selectedOrderIndex = i;                          // 确保弹窗在根级别打开                          ImGui::OpenPopup("##OrderDetails");                      }                  }              }          }          if (showCompleted && !completedOrders.empty())          {              if (ImGui::CollapsingHeader("已完成订单", ImGuiTreeNodeFlags\_DefaultOpen))              {                  for (int i = 0; i < completedOrders.size(); i++)                  {                      int index = i + pendingOrders.size(); // 调整索引                      std::string displayName = "订单 " + completedOrders[i]->getOrderId().substr(9) + "...";                      if (ImGui::Selectable(displayName.c\_str(), selectedOrderIndex == index))                      {                          selectedOrderIndex = index;                          // 确保弹窗在根级别打开                          ImGui::OpenPopup("##OrderDetails");                      }                  }              }          }          // 如果没有符合条件的订单，显示提示信息          if ((pendingOrders.empty() && completedOrders.empty()) ||              (!showPending && !showCompleted) ||              (pendingOrders.empty() && showPending && !showCompleted) ||              (completedOrders.empty() && !showPending && showCompleted))          {              ImGui::TextColored(ImVec4(0.7f, 0.7f, 0.7f, 1.0f), "没有符合条件的订单");              ImGui::TextColored(ImVec4(0.7f, 0.7f, 0.7f, 1.0f), "请点击\"刷新订单列表\"按钮加载订单");          }          if (ImGui::BeginPopupModal("##OrderDetails", NULL, ImGuiWindowFlags\_AlwaysAutoResize))          {              std::shared\_ptr<Order> order;              if (selectedOrderIndex >= 0)              {                  if (selectedOrderIndex < pendingOrders.size())                  {                      order = pendingOrders[selectedOrderIndex];                  }                  else                  {                      int completedIndex = selectedOrderIndex - pendingOrders.size();                      if (completedIndex < completedOrders.size())                      {                          order = completedOrders[completedIndex];                      }                  }              }              if (order)              {                  renderOrderDetails(order);              }              else              {                  ImGui::Text("订单详情加载失败");                  if (ImGui::Button("关闭", ImVec2(120, 30)))                  {                      ImGui::CloseCurrentPopup();                  }              }              ImGui::EndPopup();          }          ImGui::EndChild(); // 显示订单详情弹窗          ImGui::Separator();          // 底部按钮 - 关闭窗口          if (ImGui::Button("关闭窗口", ImVec2(120, 30)))          {              showOrderWindow = false;          }          ImGui::End();      }  }  // 渲染订单详情  void UI::renderOrderDetails(std::shared\_ptr<Order> order)  {      if (!order)          return;      // 订单基本信息      ImGui::Text("订单编号: %s", order->getOrderId().c\_str());      ImGui::Text("下单用户: %s", order->getCustomerUsername().c\_str());      ImGui::Text("订单状态: %s", order->getStatus().c\_str());      ImGui::Text("订单总额: CNY %.2f", order->getTotalAmount());      char timeBuffer[64];      time\_t timestamp = order->getTimestamp();      struct tm \*timeinfo = localtime(&timestamp);      strftime(timeBuffer, sizeof(timeBuffer), "%Y-%m-%d %H:%M:%S", timeinfo);      ImGui::Text("下单时间: %s", timeBuffer);      ImGui::Separator(); // 订单商品列表      ImGui::Text("订单商品:");      if (ImGui::BeginTable("OrderItemTable", 4, ImGuiTableFlags\_Borders | ImGuiTableFlags\_RowBg))      {          ImGui::TableSetupColumn("商品名称");          ImGui::TableSetupColumn("数量");          ImGui::TableSetupColumn("单价");          ImGui::TableSetupColumn("商家");          ImGui::TableHeadersRow();          const std::vector<OrderItem> &items = order->getItems();          if (items.empty())          {              // 如果没有商品，显示一条信息              ImGui::TableNextRow();              ImGui::TableSetColumnIndex(0);              ImGui::TextColored(ImVec4(0.7f, 0.7f, 0.7f, 1.0f), "（无商品数据）");          }          else          {              // 显示所有商品              for (const auto &item : items)              {                  ImGui::TableNextRow();                  ImGui::TableSetColumnIndex(0);                  ImGui::TextUnformatted(item.productName.c\_str()); // 使用TextUnformatted避免格式化问题                  ImGui::TableSetColumnIndex(1);                  ImGui::Text("%d", item.quantity);                  ImGui::TableSetColumnIndex(2);                  ImGui::Text("CNY %.2f", item.priceAtPurchase);                  ImGui::TableSetColumnIndex(3);                  ImGui::TextUnformatted(item.sellerUsername.c\_str());              }          }          ImGui::EndTable();      }      ImGui::Separator();      // 订单操作按钮      if (order->getStatus() == "PENDING\_CONFIRMATION" || order->getStatus() == "PENDING\_IN\_QUEUE")      {          if (ImGui::Button("取消订单", ImVec2(120, 30)))          {              // 添加取消订单的代码              // TODO: 实现取消订单功能              ImGui::OpenPopup("订单取消确认");          }          if (ImGui::BeginPopupModal("订单取消确认", NULL, ImGuiWindowFlags\_AlwaysAutoResize))          {              ImGui::Text("确定要取消此订单吗？操作无法撤销。");              if (ImGui::Button("确定", ImVec2(120, 0)))              {                  // 实际取消订单                  // TODO: 实现实际取消订单功能                  ImGui::CloseCurrentPopup();                  ImGui::CloseCurrentPopup(); // 关闭订单详情弹窗              }              ImGui::SameLine();              if (ImGui::Button("取消", ImVec2(120, 0)))              {                  ImGui::CloseCurrentPopup();              }              ImGui::EndPopup();          }          ImGui::SameLine();      }      if (ImGui::Button("关闭", ImVec2(120, 30)))      {          // 关闭订单详情窗口          selectedOrderIndex = -1;          ImGui::CloseCurrentPopup();      }  }  // 处理订单 未完成  void UI::processOrders()  {      if (!orderManager || !currentUser)          return;      // 每隔一定时间检查一次订单状态      static float lastCheckTime = 0.0f;      static float checkInterval = 1.0f; // 1秒检查一次      ImGuiIO &io = ImGui::GetIO();      float currentTime = io.DeltaTime;      lastCheckTime += currentTime;      if (lastCheckTime >= checkInterval)      {          // 重置计时器          lastCheckTime = 0.0f;          // 获取待处理订单数          size\_t pendingCount = orderManager->getPendingOrderCount();          // 如果有待处理订单，显示提示          if (pendingCount > 0 && currentUser)          {              if (userType == 1) // 消费者              {                  // 更新消费者的订单列表                  // 这里可以添加代码获取消费者的订单                  // 显示订单更新通知                  // 这里可以添加代码显示订单状态更新的通知              }              else if (userType == 2) // 商家              {                  // 商家可以看到自己需要处理的订单                  // 这里可以添加代码获取商家的订单              }          }      }  }  // 加载用户订单  void UI::loadUserOrders()  {      if (!orderManager || !currentUser || userType != 1)          return;      Customer \*customer = dynamic\_cast<Customer \*>(currentUser);      if (!customer)          return;      // 调用订单管理器获取该用户的所有订单      orderManager->getUserOrders(customer->getUsername(), pendingOrders, completedOrders);      // 日志输出加载的订单数量      std::cout << "已加载用户 " << customer->getUsername()                << " 的订单: 待处理 " << pendingOrders.size()                << " 个, 已完成 " << completedOrders.size() << " 个" << std::endl;  } |

#### store.h

|  |
| --- |
| #ifndef STORE\_H  #define STORE\_H  #include <string>  #include <vector>  #include <iostream>  #include <fstream>  #include <sstream>  #include <iomanip>  #include <limits>  #include <map>  #include <filesystem>  // 前向声明 User  class User;  // --- Product 基类和派生类 ---  class Product  {  protected:      std::string name;      std::string description;      double originalPrice;      int quantity;      double discountRate;      std::string sellerUsername; // 添加商品所属商家  public:      Product(std::string name, std::string desc, double price, int qty, std::string seller = "")          : name(name), description(desc), originalPrice(price), quantity(qty),            discountRate(0.0), sellerUsername(seller) {}      virtual ~Product() = default;      virtual double getPrice() const { return originalPrice \* (1.0 - discountRate); }      virtual void display() const;      virtual std::string getType() const = 0;      virtual std::string getUserCategory() const { return ""; } // 默认返回空字符串      virtual void save(std::ofstream &ofs) const;      // Getters and Setters      std::string getName() const { return name; }      std::string getDescription() const { return description; }      double getOriginalPrice() const { return originalPrice; }      int getQuantity() const { return quantity; }      double getDiscountRate() const { return discountRate; }      std::string getSellerUsername() const { return sellerUsername; } // 获取商品所属商家      void setName(const std::string &newName) { name = newName; }      void setDescription(const std::string &newDesc) { description = newDesc; }      void setOriginalPrice(double newPrice)      {          if (newPrice >= 0)              originalPrice = newPrice;      }      void setQuantity(int newQuantity)      {          if (newQuantity >= 0)              quantity = newQuantity;      }      void setDiscountRate(double newRate)      {          if (newRate >= 0.0 && newRate <= 1.0)              discountRate = newRate;      }      void setSellerUsername(const std::string &seller) { sellerUsername = seller; } // 设置商品所属商家  };  class Book : public Product  {  private:      std::string author;      std::string isbn;  public:      Book(std::string name, std::string desc, double price, int qty,           std::string auth, std::string isbn\_num, std::string seller = "")          : Product(name, desc, price, qty, seller), author(auth), isbn(isbn\_num) {}      std::string getType() const override { return "Book"; }      std::string getUserCategory() const override { return "Book"; } // 固定类型为 "Book"      void display() const override;      void save(std::ofstream &ofs) const override;      // 新增的 getters      std::string getAuthor() const { return author; }      std::string getIsbn() const { return isbn; }  };  class Clothing : public Product  {  private:      std::string size;      std::string color;  public:      Clothing(std::string name, std::string desc, double price, int qty,               std::string sz, std::string clr, std::string seller = "")          : Product(name, desc, price, qty, seller), size(sz), color(clr) {}      std::string getType() const override { return "Clothing"; }      std::string getUserCategory() const override { return "Clothing"; } // 固定类型为 "Clothing"      void display() const override;      void save(std::ofstream &ofs) const override;      // 新增的 getters      std::string getSize() const { return size; }      std::string getColor() const { return color; }  };  class Food : public Product  {  private:      std::string expirationDate;  public:      Food(std::string name, std::string desc, double price, int qty,           std::string expDate, std::string seller = "")          : Product(name, desc, price, qty, seller), expirationDate(expDate) {}      std::string getType() const override { return "Food"; }      std::string getUserCategory() const override { return "Food"; } // 固定类型为 "Food"      void display() const override;      void save(std::ofstream &ofs) const override;      // 新增的 getters      std::string getExpirationDate() const { return expirationDate; }  };  // 自定义商品类  class GenericProduct : public Product  {  private:      std::string categoryTag; // 用户自定义的分类标签  public:      GenericProduct(std::string name, std::string desc, double price, int qty,                     std::string tag, std::string seller = "")          : Product(name, desc, price, qty, seller), categoryTag(tag) {}      std::string getType() const override { return "Generic"; }           // 固定类型为 "Generic"      std::string getUserCategory() const override { return categoryTag; } // 返回用户自定义标签      void display() const override;      void save(std::ofstream &ofs) const override;      // Getter for categoryTag      std::string getCategoryTag() const { return categoryTag; }      void setCategoryTag(const std::string &tag) { categoryTag = tag; }  };  // --- Store Class ---  class Store  {  private:      std::vector<Product \*> allProducts;                           // 存储所有商家的商品      std::map<std::string, std::vector<Product \*>> sellerProducts; // 每个商家的商品映射      std::string storeDirectory;                                   // 商品文件所在目录      // 辅助方法      std::string getSellerFilename(const std::string &username) const;      bool saveProductsForSeller(const std::string &sellerUsername);      bool ensureDirectoryExists(const std::string &path) const;  public:      // 构造函数接收目录名      Store(const std::string &directory);      ~Store();      Product \*findProductByName(const std::string &name, const std::string &sellerUsername = "");      // 文件操作      bool loadAllProducts();      bool loadSellerProducts(const std::string &sellerUsername);      bool saveAllProducts();      // 显示功能      void displayAllProducts() const;                                     // 显示所有商品      void displaySellerProducts(const std::string &sellerUsername) const; // 显示指定商家的商品      // 搜索功能      std::vector<Product \*> searchProductsByName(const std::string &searchTerm, const std::string &sellerUsername = "") const;      // 商家商品管理功能      bool createBook(User \*currentUser, const std::string &name, const std::string &desc,                      double price, int qty, const std::string &author, const std::string &isbn);      bool createClothing(User \*currentUser, const std::string &name, const std::string &desc,                          double price, int qty, const std::string &size, const std::string &color);      bool createFood(User \*currentUser, const std::string &name, const std::string &desc,                      double price, int qty, const std::string &expDate);      bool createGenericProduct(User \*currentUser, const std::string &name, const std::string &desc,                                double price, int qty, const std::string &categoryTag);      bool manageProductPrice(User \*currentUser, const std::string &productName, double newPrice);      bool manageProductQuantity(User \*currentUser, const std::string &productName, int newQuantity);      bool manageProductDiscount(User \*currentUser, const std::string &productName, double newDiscount);      bool applyCategoryDiscount(User \*currentUser, const std::string &category, double discount);      // 获取商品      const std::vector<Product \*> &getProducts() const { return allProducts; }      std::vector<Product \*> getSellerProducts(const std::string &sellerUsername) const;      // 获取商家商品的唯一分类      std::vector<std::string> getUniqueCategoriesForSeller(const std::string &sellerUsername) const;  };  #endif // STORE\_H |

#### store.cpp

|  |
| --- |
| #include "store.h"  #include "../user/user.h"  #include <iostream>  #include <vector>  #include <string>  #include <fstream>  #include <sstream>  #include <iomanip>  #include <algorithm>  #include <filesystem>  #include <set>  #include <direct.h> // Windows 系统特定的目录操作  namespace fs = std::filesystem;  using namespace std;  // --- Product 方法实现 ---  void Product::display() const  {      std::cout << "  商户名称：" << sellerUsername << std::endl;      std::cout << "  商品名称: " << name << std::endl;      std::cout << "  描述: " << description << std::endl;      if (!getUserCategory().empty())      {          std::cout << "  分类标签: " << getUserCategory() << std::endl;      }      if (discountRate > 0)      {          std::cout << "  原价: ¥" << std::fixed << std::setprecision(2) << originalPrice << std::endl;          std::cout << "  折扣: " << (discountRate \* 100) << "%" << std::endl;          std::cout << "  现价: ¥" << std::fixed << std::setprecision(2) << getPrice() << std::endl;      }      else      {          std::cout << "  价格: ¥" << std::fixed << std::setprecision(2) << getPrice() << std::endl;      }      std::cout << "  库存: " << quantity << " 件" << std::endl;      if (!sellerUsername.empty())      {          std::cout << "  商家: " << sellerUsername << std::endl;      }  }  void Product::save(std::ofstream &ofs) const  {      ofs << getType() << "," << name << "," << description << ","          << originalPrice << "," << quantity << "," << discountRate << "," << sellerUsername;  }  void Book::display() const // 复用父类  {      Product::display();      std::cout << "  作者: " << author << std::endl;      std::cout << "  ISBN: " << isbn << std::endl;  }  void Book::save(std::ofstream &ofs) const  {      Product::save(ofs);      ofs << "," << author << "," << isbn;  }  void Clothing::display() const  {      Product::display();      std::cout << "  尺寸: " << size << std::endl;      std::cout << "  颜色: " << color << std::endl;  }  void Clothing::save(std::ofstream &ofs) const  {      Product::save(ofs);      ofs << "," << size << "," << color;  }  void Food::display() const  {      Product::display();      std::cout << "  保质期限: " << expirationDate << std::endl;  }  void Food::save(std::ofstream &ofs) const  {      Product::save(ofs);      ofs << "," << expirationDate;  }  void GenericProduct::display() const  {      Product::display();  }  void GenericProduct::save(std::ofstream &ofs) const  {      Product::save(ofs);      ofs << "," << categoryTag;  }  // --- Store 类实现 ---  // 构造函数  Store::Store(const string &directory) : storeDirectory(directory)  {      // 确保商家目录存在      string sellersDir = storeDirectory + "/sellers";      ensureDirectoryExists(sellersDir);      // 加载所有商品数据      if (!loadAllProducts())      {          cerr << "警告: 无法加载商店数据。商店可能为空或文件丢失/损坏。" << endl;      }      else      {          cout << "已从商店加载 " << allProducts.size() << " 件商品。" << endl;      }  }  // 析构函数  Store::~Store()  {      for (Product \*p : allProducts)      {          delete p;      }      allProducts.clear();      sellerProducts.clear(); // 清除映射  }  // 确保目录存在  bool Store::ensureDirectoryExists(const string &path) const  {      try      {          if (!fs::exists(path))          {              fs::create\_directories(path);              cout << "创建目录: " << path << endl;          }          return true;      }      catch (const fs::filesystem\_error &e)      {          cerr << "创建目录失败 " << path << ": " << e.what() << endl;          // 尝试使用传统方法创建目录          if (\_mkdir(path.c\_str()) == 0)          {              cout << "使用传统方法创建目录: " << path << endl;              return true;          }          return false;      }      catch (...)      {          cerr << "创建目录时发生未知错误: " << path << endl;          return false;      }  }  // 获取商家文件名  string Store::getSellerFilename(const string &username) const  {      return storeDirectory + "/sellers/" + username + ".txt";  }  // 加载所有商品  bool Store::loadAllProducts()  {      // 清空现有商品      for (Product \*p : allProducts)      {          delete p;      }      allProducts.clear();      sellerProducts.clear();      // 获取sellers目录下的所有文件      string sellersDir = storeDirectory + "/sellers";      try      {          // 如果目录不存在，创建目录并返回成功（表示商店为空）          if (!fs::exists(sellersDir))          {              ensureDirectoryExists(sellersDir);              return true;          }          // 遍历目录中的所有文件          for (const auto &entry : fs::directory\_iterator(sellersDir))          {              if (entry.is\_regular\_file())              {                  string filename = entry.path().string();                  string sellerUsername = entry.path().stem().string(); // 获取不带扩展名的文件名                  loadSellerProducts(sellerUsername);              }          }          return true;      }      catch (const fs::filesystem\_error &e)      {          cerr << "读取商家目录失败: " << e.what() << endl;          return false;      }      catch (...)      {          cerr << "读取商家数据时发生未知错误" << endl;          return false;      }  }  // 加载指定商家的商品  bool Store::loadSellerProducts(const string &sellerUsername)  {      string filename = getSellerFilename(sellerUsername);      ifstream file(filename);      if (!file.is\_open())      {          // 对于新商家，文件不存在是正常的          return true;      }      vector<Product \*> sellerProductsList;      string line;      while (getline(file, line))      {          stringstream ss(line);          string segment;          vector<string> seglist;          while (getline(ss, segment, ','))          {              seglist.push\_back(segment);          }          // 格式检查: 至少需要7个字段 (类型,名称,描述,价格,数量,折扣,商家)          if (seglist.size() < 7)          {              cerr << "无效的产品数据行: " << line << endl;              continue;          }          try          {              string type = seglist[0];              string name = seglist[1];              string desc = seglist[2];              double price = stod(seglist[3]);              int qty = stoi(seglist[4]);              double discount = stod(seglist[5]);              string seller = seglist[6]; // 第7个字段是商家名              // 确认商家名称匹配              if (seller != sellerUsername)              {                  cerr << "警告: 文件 " << filename << " 中的商品所属商家 \""                       << seller << "\" 与预期的 \"" << sellerUsername << "\" 不符" << endl;                  seller = sellerUsername; // 强制使用文件名指示的商家              }              Product \*newProduct = nullptr;              if (type == "Book" && seglist.size() >= 9)              {                  // Book需要额外的author和isbn字段                  newProduct = new Book(name, desc, price, qty, seglist[7], seglist[8], seller);              }              else if (type == "Clothing" && seglist.size() >= 9)              {                  // Clothing需要额外的size和color字段                  newProduct = new Clothing(name, desc, price, qty, seglist[7], seglist[8], seller);              }              else if (type == "Food" && seglist.size() >= 8)              {                  // Food需要额外的expirationDate字段                  newProduct = new Food(name, desc, price, qty, seglist[7], seller);              }              else if (type == "Generic" && seglist.size() >= 8)              {                  // GenericProduct需要额外的categoryTag字段                  newProduct = new GenericProduct(name, desc, price, qty, seglist[7], seller);              }              else              {                  cerr << "未知商品类型或缺少必要字段: " << line << endl;                  continue;              }              if (newProduct)              {                  newProduct->setDiscountRate(discount);                  allProducts.push\_back(newProduct);                  sellerProductsList.push\_back(newProduct);              }          }          catch (const exception &e)          {              cerr << "解析商品数据时出错: " << e.what() << " 行: " << line << endl;          }      }      file.close();      // 更新商家商品映射      sellerProducts[sellerUsername] = sellerProductsList;      cout << "已加载商家 \"" << sellerUsername << "\" 的 "           << sellerProductsList.size() << " 件商品" << endl;      return true;  }  // 保存所有商品  bool Store::saveAllProducts()  {      // 对于每个商家，保存其商品      set<string> processedSellers; // 用于跟踪已处理的商家      for (const auto &product : allProducts)      {          string seller = product->getSellerUsername();          if (!seller.empty() && processedSellers.find(seller) == processedSellers.end())          {              saveProductsForSeller(seller);              processedSellers.insert(seller);          }      }      return true;  }  // 保存指定商家的商品  bool Store::saveProductsForSeller(const string &sellerUsername)  {      if (sellerUsername.empty())      {          cerr << "错误: 尝试保存无效商家的商品" << endl;          return false;      }      vector<Product \*> products;      // 获取该商家的所有商品      for (Product \*p : allProducts)      {          if (p->getSellerUsername() == sellerUsername)          {              products.push\_back(p);          }      }      string filename = getSellerFilename(sellerUsername);      ofstream file(filename);      if (!file.is\_open())      {          cerr << "错误: 无法打开文件保存商品: " << filename << endl;          return false;      }      for (const Product \*p : products)      {          p->save(file);          file << endl;      }      file.close();      // cout << "商家 \"" << sellerUsername << "\" 的 " << products.size()      //      << " 件商品已保存至 " << filename << endl;      return true;  }  // 查找商品  Product \*Store::findProductByName(const string &name, const string &sellerUsername)  {      if (!sellerUsername.empty())      {          // 只在指定商家的商品中查找          auto it = sellerProducts.find(sellerUsername);          if (it != sellerProducts.end())          {              for (Product \*p : it->second)              {                  if (p->getName() == name)                  {                      return p;                  }              }          }          return nullptr;      }      else      {          // 在所有商品中查找          for (Product \*p : allProducts)          {              if (p->getName() == name)              {                  return p;              }          }          return nullptr;      }  }  // 显示所有商品  void Store::displayAllProducts() const  {      if (allProducts.empty())      {          cout << "\n--- 商品列表为空 ---" << endl;          return;      }      cout << "\n--- 所有商品列表 ---" << endl;      int count = 0;      for (const Product \*p : allProducts)      {          cout << "--------------------------" << endl;          p->display();          count++;      }      cout << "--------------------------" << endl;      cout << "共显示 " << count << " 件商品" << endl;  }  // 显示指定商家的商品  void Store::displaySellerProducts(const string &sellerUsername) const  {      auto it = sellerProducts.find(sellerUsername);      if (it == sellerProducts.end() || it->second.empty())      {          cout << "\n--- 商家 \"" << sellerUsername << "\" 没有商品 ---" << endl;          return;      }      cout << "\n--- 商家 \"" << sellerUsername << "\" 的商品列表 ---" << endl;      int count = 0;      for (const Product \*p : it->second)      {          cout << "--------------------------" << endl;          p->display();          count++;      }      cout << "--------------------------" << endl;      cout << "共显示 " << count << " 件商品" << endl;  }  // 按名称搜索商品  vector<Product \*> Store::searchProductsByName(const string &searchTerm, const string &sellerUsername) const  {      vector<Product \*> results;      const vector<Product \*> &productsToSearch = sellerUsername.empty() ? allProducts : (sellerProducts.count(sellerUsername) ? sellerProducts.at(sellerUsername) : vector<Product \*>());      for (Product \*p : productsToSearch)      {          // 简单的包含搜索 (不区分大小写)          string productName = p->getName();          string searchTermLower = searchTerm;          // 将两个字符串转为小写进行比较          transform(productName.begin(), productName.end(), productName.begin(), ::tolower);          transform(searchTermLower.begin(), searchTermLower.end(), searchTermLower.begin(), ::tolower);          if (productName.find(searchTermLower) != string::npos)          {              results.push\_back(p);          }      }      return results;  }  // 获取商家的商品  vector<Product \*> Store::getSellerProducts(const string &sellerUsername) const  {      auto it = sellerProducts.find(sellerUsername);      return (it != sellerProducts.end()) ? it->second : vector<Product \*>();  }  // 创建图书  bool Store::createBook(User \*currentUser, const string &name, const string &desc,                         double price, int qty, const string &author, const string &isbn)  {      if (!currentUser)      {          cerr << "错误: 用户未登录" << endl;          return false;      }      if (currentUser->getUserType() != "seller")      {          cerr << "错误: 只有商家可以添加商品" << endl;          return false;      }      string sellerUsername = currentUser->getUsername();      // 检查该商家是否已有同名商品      if (findProductByName(name, sellerUsername) != nullptr)      {          cerr << "错误: 您已有名为 \"" << name << "\" 的商品" << endl;          return false;      }      if (price < 0 || qty < 0)      {          cerr << "错误: 价格和数量不能为负数" << endl;          return false;      }      try      {          Book \*newBook = new Book(name, desc, price, qty, author, isbn, sellerUsername);          allProducts.push\_back(newBook);          // 更新商家商品映射          sellerProducts[sellerUsername].push\_back(newBook);          // 保存商家的商品          if (saveProductsForSeller(sellerUsername))          {              cout << "商品 \"" << name << "\" 添加成功！" << endl;              return true;          }          else          {              cout << "商品 \"" << name << "\" 添加失败！" << endl;              return false;          }      }      catch (const bad\_alloc &e)      {          cerr << "错误: 无法分配内存: " << e.what() << endl;          return false;      }      catch (const exception &e)      {          cerr << "错误: " << e.what() << endl;          return false;      }      catch (...)      {          cerr << "错误: 添加商品时发生未知错误" << endl;          return false;      }  }  // 创建服装  bool Store::createClothing(User \*currentUser, const string &name, const string &desc,                             double price, int qty, const string &size, const string &color)  {      if (!currentUser)      {          cerr << "错误: 用户未登录" << endl;          return false;      }      if (currentUser->getUserType() != "seller")      {          cerr << "错误: 只有商家可以添加商品" << endl;          return false;      }      string sellerUsername = currentUser->getUsername();      // 检查该商家是否已有同名商品      if (findProductByName(name, sellerUsername) != nullptr)      {          cerr << "错误: 您已有名为 \"" << name << "\" 的商品" << endl;          return false;      }      if (price < 0 || qty < 0)      {          cerr << "错误: 价格和数量不能为负数" << endl;          return false;      }      try      {          Clothing \*newClothing = new Clothing(name, desc, price, qty, size, color, sellerUsername);          allProducts.push\_back(newClothing);          // 更新商家商品映射          sellerProducts[sellerUsername].push\_back(newClothing);          // 保存商家的商品          if (saveProductsForSeller(sellerUsername))          {              cout << "商品 \"" << name << "\" 添加成功！" << endl;              return true;          }          else          {              cout << "商品 \"" << name << "\" 添加失败！" << endl;              return false;          }      }      catch (const bad\_alloc &e)      {          cerr << "错误: 无法分配内存: " << e.what() << endl;          return false;      }      catch (const exception &e)      {          cerr << "错误: " << e.what() << endl;          return false;      }      catch (...)      {          cerr << "错误: 添加商品时发生未知错误" << endl;          return false;      }  }  // 创建食品  bool Store::createFood(User \*currentUser, const string &name, const string &desc,                         double price, int qty, const string &expDate)  {      if (!currentUser)      {          cerr << "错误: 用户未登录" << endl;          return false;      }      if (currentUser->getUserType() != "seller")      {          cerr << "错误: 只有商家可以添加商品" << endl;          return false;      }      string sellerUsername = currentUser->getUsername();      // 检查该商家是否已有同名商品      if (findProductByName(name, sellerUsername) != nullptr)      {          cerr << "错误: 您已有名为 \"" << name << "\" 的商品" << endl;          return false;      }      if (price < 0 || qty < 0)      {          cerr << "错误: 价格和数量不能为负数" << endl;          return false;      }      try      {          Food \*newFood = new Food(name, desc, price, qty, expDate, sellerUsername);          allProducts.push\_back(newFood);          // 更新商家商品映射          sellerProducts[sellerUsername].push\_back(newFood);          // 保存商家的商品          if (saveProductsForSeller(sellerUsername))          {              cout << "商品 \"" << name << "\" 添加成功！" << endl;              return true;          }          else          {              cout << "商品 \"" << name << "\" 添加失败！" << endl;              return false;          }      }      catch (const bad\_alloc &e)      {          cerr << "错误: 无法分配内存: " << e.what() << endl;          return false;      }      catch (const exception &e)      {          cerr << "错误: " << e.what() << endl;          return false;      }      catch (...)      {          cerr << "错误: 添加商品时发生未知错误" << endl;          return false;      }  }  bool Store::createGenericProduct(User \*currentUser, const string &name, const string &desc,                                   double price, int qty, const string &categoryTag)  {      if (!currentUser)      {          cerr << "错误: 用户未登录" << endl;          return false;      }      if (currentUser->getUserType() != "seller")      {          cerr << "错误: 只有商家可以添加商品" << endl;          return false;      }      string sellerUsername = currentUser->getUsername();      if (findProductByName(name, sellerUsername) != nullptr)      {          cerr << "错误: 您已有名为 \"" << name << "\" 的商品" << endl;          return false;      }      if (price < 0 || qty < 0)      {          cerr << "错误: 价格和数量不能为负数" << endl;          return false;      }      if (categoryTag.empty())      {          cerr << "错误: 通用商品必须指定分类标签" << endl;          return false;      }      try      {          GenericProduct \*newGenericProduct = new GenericProduct(name, desc, price, qty, categoryTag, sellerUsername);          allProducts.push\_back(newGenericProduct);          sellerProducts[sellerUsername].push\_back(newGenericProduct);          if (saveProductsForSeller(sellerUsername))          {              cout << "通用商品 \"" << name << "\" (分类: " << categoryTag << ") 添加成功！" << endl;              return true;          }          else          {              // 如果保存失败，应该从内存中移除，以保持一致性              allProducts.pop\_back();              sellerProducts[sellerUsername].pop\_back();              delete newGenericProduct;              cout << "通用商品 \"" << name << "\" 添加失败（保存错误）！" << endl;              return false;          }      }      catch (const bad\_alloc &e)      {          cerr << "错误: 无法分配内存: " << e.what() << endl;          return false;      }      catch (const exception &e)      {          cerr << "错误: " << e.what() << endl;          return false;      }      catch (...)      {          cerr << "错误: 添加通用商品时发生未知错误" << endl;          return false;      }  }  // 修改商品价格  bool Store::manageProductPrice(User \*currentUser, const string &productName, double newPrice)  {      if (!currentUser)      {          cerr << "错误: 用户未登录" << endl;          return false;      }      if (currentUser->getUserType() != "seller")      {          cerr << "错误: 只有商家可以修改商品价格" << endl;          return false;      }      string sellerUsername = currentUser->getUsername();      Product \*product = findProductByName(productName, sellerUsername);      if (!product)      {          cerr << "错误: 未找到您的商品 \"" << productName << "\"" << endl;          return false;      }      if (newPrice < 0)      {          cerr << "错误: 价格不能为负数" << endl;          return false;      }      product->setOriginalPrice(newPrice);      // 保存商家的商品      if (saveProductsForSeller(sellerUsername))      {          cout << "价格保存成功！" << endl;          return true;      }      else      {          cout << "价格保存失败！" << endl;          return false;      }  }  // 修改商品库存  bool Store::manageProductQuantity(User \*currentUser, const string &productName, int newQuantity)  {      if (!currentUser)      {          cerr << "错误: 用户未登录" << endl;          return false;      }      if (currentUser->getUserType() != "seller")      {          cerr << "错误: 只有商家可以修改商品库存" << endl;          return false;      }      string sellerUsername = currentUser->getUsername();      Product \*product = findProductByName(productName, sellerUsername);      if (!product)      {          cerr << "错误: 未找到您的商品 \"" << productName << "\"" << endl;          return false;      }      if (newQuantity < 0)      {          cerr << "错误: 库存不能为负数" << endl;          return false;      }      product->setQuantity(newQuantity);      if (saveProductsForSeller(sellerUsername))      {          cout << "库存修改成功！" << endl;          return true;      }      else      {          cout << "库存修改失败！" << endl;          return false;      }  }  // 修改商品折扣  bool Store::manageProductDiscount(User \*currentUser, const string &productName, double newDiscount)  {      if (!currentUser)      {          cerr << "错误: 用户未登录" << endl;          return false;      }      if (currentUser->getUserType() != "seller")      {          cerr << "错误: 只有商家可以修改商品折扣" << endl;          return false;      }      string sellerUsername = currentUser->getUsername();      Product \*product = findProductByName(productName, sellerUsername);      if (!product)      {          cerr << "错误: 未找到您的商品 \"" << productName << "\"" << endl;          return false;      }      if (newDiscount < 0 || newDiscount > 100)      {          cerr << "错误: 折扣必须在0到100之间" << endl;          return false;      }      product->setDiscountRate(newDiscount);      return saveProductsForSeller(sellerUsername);  }  std::vector<std::string> Store::getUniqueCategoriesForSeller(const std::string &sellerUsername) const  {      std::set<std::string> uniqueCategories;      auto it = sellerProducts.find(sellerUsername);      if (it != sellerProducts.end())      {          for (const Product \*p : it->second)          {              if (p)              {                                                // Ensure product pointer is not null                  std::string category = p->getUserCategory(); // This gets "Book", "Clothing", "Food", or custom tag                  if (!category.empty())                  { // Ensure not to add empty strings                      uniqueCategories.insert(category);                  }              }          }      }      return std::vector<std::string>(uniqueCategories.begin(), uniqueCategories.end());  }  // 应用分类折扣  bool Store::applyCategoryDiscount(User \*currentUser, const string &category, double discount)  {      if (!currentUser)      {          cerr << "错误: 用户未登录" << endl;          return false;      }      if (currentUser->getUserType() != "seller")      {          cerr << "错误: 只有商家可以应用折扣" << endl;          return false;      }      if (discount < 0.0 || discount > 1.0)      {          cerr << "错误: 折扣必须在0到100之间" << endl;          return false;      }      string sellerUsername = currentUser->getUsername();      bool changed = false;      // 只应用折扣到该商家的商品      auto it = sellerProducts.find(sellerUsername);      if (it != sellerProducts.end())      {          for (Product \*p : it->second)          {              // 使用 getUserCategory() 进行比较              if (p && p->getUserCategory() == category)              {                  p->setDiscountRate(discount);                  changed = true;              }          }      }      if (changed)      {          cout << "已对分类 \"" << category << "\" 下的商品应用 " << (discount \* 100) << "% 折扣。" << endl;          return saveProductsForSeller(sellerUsername);      }      cerr << "错误: 未找到分类为 \"" << category << "\" 的商品" << endl;      return false;  } |

#### order.h

|  |
| --- |
| #ifndef ORDER\_H  #define ORDER\_H  #include <string>  #include <vector>  #include <ctime>  #include <iomanip>  // For std::setprecision  #include <iostream> // For display methods  // Forward declarations  class Product; // From store.h  class User;    // From user.h  class Store;   // From store.h  class OrderItem  {  public:      std::string productId;      std::string productName;      int quantity;      double priceAtPurchase;      std::string sellerUsername;      OrderItem(std::string pid, std::string pName, int qty, double price, std::string sUsername)          : productId(std::move(pid)), productName(std::move(pName)), quantity(qty),            priceAtPurchase(price), sellerUsername(std::move(sUsername)) {}      void display() const;      std::string toStringForSave() const;  };  class Order  {  private:      std::string orderId;      std::string customerUsername;      std::vector<OrderItem> items;      double totalAmount;      bool isProcessed; // Indicates if the order has been processed      time\_t orderTimestamp;      std::string status; // e.g., "PENDING\_CONFIRMATION", "COMPLETED", "CANCELLED\_INSUFFICIENT\_STOCK", "CANCELLED\_INSUFFICIENT\_FUNDS", "CANCELLED\_BY\_USER"      static std::string generateOrderId();  public:      Order(std::string custUsername);      // 新增默认构造函数      Order() : customerUsername(""),                totalAmount(0.0),                orderTimestamp(std::time(nullptr)), status("UNINITIALIZED"),                isProcessed(false)      {          orderId = "EMPTY-" + std::to\_string(orderTimestamp);      }      bool addItemFromProduct(const Product &product, int quantity); // Adds item and updates total      void addItem(const OrderItem &item);                           // Adds an existing OrderItem and updates total      void calculateTotalAmount();                                   // Recalculates total based on items      // Getters      std::string getOrderId() const { return orderId; }      std::string getCustomerUsername() const { return customerUsername; }      const std::vector<OrderItem> &getItems() const { return items; }      double getTotalAmount() const { return totalAmount; }      std::string getStatus() const { return status; }      time\_t getTimestamp() const { return orderTimestamp; } // Setters      void setStatus(const std::string &newStatus) { status = newStatus; }      void setProcessed(bool processed) { isProcessed = processed; }      bool getProcessed() const { return isProcessed; }      // 添加必要的 setter 方法      void setTimestamp(time\_t timestamp) { orderTimestamp = timestamp; }      void setOrderId(const std::string &id) { orderId = id; }      void setTotalAmount(double amount) { totalAmount = amount; }      void displaySummary() const;      std::string toStringForSaveHeader() const;      // Persistence (Optional - for saving orders to files)      // bool saveToFile(const std::string& directoryPath) const;      // static Order loadFromFile(const std::string& filePath);  };  #endif // ORDER\_H |

#### order.cpp

|  |
| --- |
| #include "order.h"  #include "../store/store.h"  #include "../user/user.h" // For Product and User definitions  #include <sstream>  #include <chrono>    // For generating unique ID  #include <fstream>   // For file operations (if saving orders)  #include <algorithm> // For std::remove\_if (if needed for item management)  // --- OrderItem Implementation ---  void OrderItem::display() const  {      std::cout << "    - 商品名称: " << productName << " (ID: " << productId << ")" << std::endl;      std::cout << "      购买数量: " << quantity << ", 商品单价: ¥" << priceAtPurchase                << ", 商品总价: ¥" << (quantity \* priceAtPurchase)                << ", 商家: " << sellerUsername << std::endl;  }  // --- Order Implementation ---  std::string Order::generateOrderId()  {      auto now = std::chrono::system\_clock::now();      auto now\_c = std::chrono::system\_clock::to\_time\_t(now); // Convert to time\_t for tm structure      std::tm now\_tm = \*std::localtime(&now\_c);               // Get local time structure      // Get milliseconds part      auto ms = std::chrono::duration\_cast<std::chrono::milliseconds>(now.time\_since\_epoch()) % 1000;      std::stringstream ss;      ss << "ORD-";      ss << std::put\_time(&now\_tm, "%Y%m%d-%H%M%S");                // Format: YYYYMMDD-HHMMSS      ss << "-" << std::setfill('0') << std::setw(3) << ms.count(); // Append milliseconds, zero-padded      return ss.str();  }  Order::Order(std::string custUsername)      : customerUsername(std::move(custUsername)), totalAmount(0.0), status("PENDING\_CREATION")  {      orderId = generateOrderId();      orderTimestamp = std::time(nullptr);  }  bool Order::addItemFromProduct(const Product &product, int quantity)  {      if (quantity <= 0)          return false;      // Note: Stock check should happen before calling this, or this method could also check      // For now, assumes stock check is done externally before adding to order object      items.emplace\_back(product.getName(), product.getName(), quantity, product.getPrice(), product.getSellerUsername());      calculateTotalAmount(); // Recalculate total every time an item is added      return true;  }  void Order::addItem(const OrderItem &item)  {      items.push\_back(item);      calculateTotalAmount(); // Recalculate total every time an item is added  }  void Order::calculateTotalAmount()  {      totalAmount = 0;      for (const auto &item : items)      {          totalAmount += item.priceAtPurchase \* item.quantity;      }  }  void Order::displaySummary() const  {      std::cout << "\n--- 订单汇总 ---" << std::endl;      std::cout << "订单 ID: " << orderId << std::endl;      std::cout << "买家: " << customerUsername << std::endl;      std::cout << "日期: " << std::asctime(std::localtime(&orderTimestamp)); // ctime adds newline      std::cout << "状态: " << status << std::endl;      std::cout << "下单物品:" << std::endl;      std::cout << std::fixed << std::setprecision(2);      for (const auto &item : items)      {          item.display();      }      std::cout << "---------------------" << std::endl;      std::cout << "总花费: ¥" << totalAmount << std::endl;      std::cout << "---------------------" << std::endl;      std::cout << std::resetiosflags(std::ios\_base::fixed | std::ios\_base::floatfield);  }  // 转为CSV格式字符串  std::string OrderItem::toStringForSave() const  {      std::stringstream ss;      ss << std::fixed << std::setprecision(2);      // Simple CSV-like, ensure no commas in your string fields or use a different delimiter/quoting      ss << productId << "," << productName << "," << quantity << "," << priceAtPurchase << "," << sellerUsername;      return ss.str();  }  std::string Order::toStringForSaveHeader() const  {      std::stringstream ss;      ss << std::fixed << std::setprecision(2);      ss << "OrderID:" << orderId << std::endl;      ss << "CustomerUsername:" << customerUsername << std::endl;      ss << "Timestamp:" << orderTimestamp << std::endl; // Save as raw time\_t      ss << "Status:" << status << std::endl;      ss << "TotalAmount:" << totalAmount << std::endl;      ss << "ItemsCount:" << items.size() << std::endl; // Useful for loading      return ss.str();  } |

#### Ordermanager.h

|  |
| --- |
| #ifndef ORDER\_MANAGER\_H  #define ORDER\_MANAGER\_H  #include "../order/order.h"  #include "../store/store.h"  #include "../user/user.h"  #include <deque>  #include <string>  #include <vector>  #include <thread>  #include <mutex>  #include <condition\_variable>  #include <atomic>  #include <memory> // 为shared\_ptr添加  class OrderManager  {  private:      std::deque<std::shared\_ptr<Order>> orderQueue;      std::string completedOrdersDirectory;      std::string pendingOrdersFile;      // 多线程支持      std::thread processingThread;      mutable std::mutex queueMutex;      std::condition\_variable queueCondVar;      std::atomic<bool> stopProcessing;      // Helper to save a single order to its own file      bool saveOrderToFile(const Order &order) const;      // 处理线程的主函数      void processingLoop(Store &store, std::vector<User \*> &allUsers);      // 内部处理订单的方法      void processNextOrderInternal(std::shared\_ptr<Order> currentOrder, Store &store, std::vector<User \*> &allUsers);  public:      OrderManager(const std::string &ordersDir);      ~OrderManager();      // 启动和停止处理线程      void startProcessingThread(Store &store, std::vector<User \*> &allUsers);      void stopProcessingThread();      // 提交订单到队列      std::shared\_ptr<Order> submitOrderRequest(const Order &orderRequest);      // 处理队列中的订单      void processNextOrder(Store &store, std::vector<User \*> &allUsers);      void processAllPendingOrders(Store &store, std::vector<User \*> &allUsers);      // 添加这个声明！      size\_t getPendingOrderCount() const;      // 显示待处理订单      void displayPendingOrders() const;      // 获取指定用户的所有订单（包括待处理和已完成的）      void getUserOrders(const std::string &username,                         std::vector<std::shared\_ptr<Order>> &pendingOrders,                         std::vector<std::shared\_ptr<Order>> &completedOrders);  };  #endif // ORDER\_MANAGER\_H |

#### Ordermanager.cpp

|  |
| --- |
| #include "ordermanager.h"  #include "../store/store.h"  #include "../order/order.h"  #include "../user/user.h"  #include <iostream>  #include <fstream>  #include <filesystem>  #include <iomanip>  #include <chrono>  #include <thread>  #include <algorithm>  // 声明外部变量  extern const std::string USER\_FILE;  extern bool startProcessing; // 用于控制订单处理线程的启动状态  using namespace std;  // 构造函数，初始化订单目录  OrderManager::OrderManager(const std::string &ordersDir)      : completedOrdersDirectory(ordersDir), stopProcessing(false)  {      // 确保订单目录存在      if (!std::filesystem::exists(completedOrdersDirectory))      {          try          {              std::filesystem::create\_directories(completedOrdersDirectory);              cout << "已创建订单目录：" << completedOrdersDirectory << endl;          }          catch (const std::filesystem::filesystem\_error &e)          {              cerr << "创建订单目录失败: " << e.what() << endl;          }      }      // 设置待处理订单文件路径      pendingOrdersFile = completedOrdersDirectory + "/pending\_orders.txt";  }  // 析构函数 - 停止处理线程  OrderManager::~OrderManager()  {      stopProcessingThread();  }  // 启动处理线程  void OrderManager::startProcessingThread(Store &store, std::vector<User \*> &allUsers)  {      // 如果线程已经在运行，先停止      stopProcessingThread();      // 重置停止标志      stopProcessing = false;      // 启动处理线程      processingThread = std::thread(&OrderManager::processingLoop, this, std::ref(store), std::ref(allUsers));      cout << "订单处理线程已启动" << endl;  }  // 停止处理线程  void OrderManager::stopProcessingThread()  {      if (processingThread.joinable())      {          {              std::lock\_guard<std::mutex> lock(queueMutex);              stopProcessing = true; // 设置停止标志          }          // 通知条件变量，唤醒等待的线程          queueCondVar.notify\_one();          // 等待线程结束          processingThread.join();          cout << "订单处理线程已停止" << endl;      }  }  // 处理线程的主循环函数  void OrderManager::processingLoop(Store &store, std::vector<User \*> &allUsers)  {      cout << "订单处理线程开始运行..." << endl;      while (!stopProcessing)      {          std::shared\_ptr<Order> orderToProcess;          bool hasOrder = false;          {              // 获取队列互斥锁              std::unique\_lock<std::mutex> lock(queueMutex);              // 等待条件变量：直到队列非空或收到停止信号              queueCondVar.wait(lock, [this]                                { return !orderQueue.empty() || stopProcessing; });              // 如果收到停止信号且队列为空，则退出              if (stopProcessing && orderQueue.empty())                  break;              // 如果队列非空，取出队首订单              if (!orderQueue.empty())              {                  orderToProcess = orderQueue.front();                  orderQueue.pop\_front();                  hasOrder = true;              }          }          // 如果有订单需要处理，则处理它          if (hasOrder)          {              cout << "\n线程正在处理订单 ID: " << orderToProcess->getOrderId()                   << "，客户: " << orderToProcess->getCustomerUsername() << endl;              // 处理订单              processNextOrderInternal(orderToProcess, store, allUsers);          }      }      cout << "订单处理线程已终止" << endl;  }  // 内部处理订单的方法（被处理线程调用）  void OrderManager::processNextOrderInternal(std::shared\_ptr<Order> currentOrder, Store &store, std::vector<User \*> &allUsers)  {      cerr << "正在处理订单 ID: " << currentOrder->getOrderId()           << "，客户: " << currentOrder->getCustomerUsername() << endl;      currentOrder->setStatus("PROCESSING");      // 找到订单对应的客户      Customer \*customer = dynamic\_cast<Customer \*>(User::findUser(allUsers, currentOrder->getCustomerUsername()));      if (!customer)      {          cerr << "错误: 找不到订单对应的客户 " << currentOrder->getCustomerUsername() << endl;          currentOrder->setStatus("FAILED\_CUSTOMER\_NOT\_FOUND");          saveOrderToFile(\*currentOrder);          currentOrder->setProcessed(true);          return;      }      // 第一阶段：重新验证商品（库存可能在队列等待期间发生变化）      for (const auto &item : currentOrder->getItems())      {          Product \*product = store.findProductByName(item.productId);          if (!product)          {              cerr << "错误: 商品 \"" << item.productName << "\" 不存在或已下架。订单取消。" << endl;              currentOrder->setStatus("FAILED\_PRODUCT\_NOT\_FOUND");              saveOrderToFile(\*currentOrder);              currentOrder->setProcessed(true);              return;          }          if (product->getQuantity() < item.quantity)          {              cerr << "错误: 商品 \"" << item.productName << "\" 库存不足 (需要 "                   << item.quantity << ", 现有 " << product->getQuantity() << ")。订单取消。" << endl;              currentOrder->setStatus("FAILED\_INSUFFICIENT\_STOCK");              saveOrderToFile(\*currentOrder);              currentOrder->setProcessed(true);              return;          }      }      // 第二阶段：验证客户余额      if (customer->checkBalance() < currentOrder->getTotalAmount())      {          cerr << "错误: 客户 " << customer->getUsername() << " 余额不足。订单取消。" << endl;          currentOrder->setStatus("FAILED\_INSUFFICIENT\_FUNDS");          saveOrderToFile(\*currentOrder);          currentOrder->setProcessed(true);          return;      }      // 第三阶段：处理支付和更新库存      if (!customer->withdraw(currentOrder->getTotalAmount()))      {          cerr << "严重错误: 客户 " << customer->getUsername()               << " 扣款失败，尽管余额检查已通过。订单: " << currentOrder->getOrderId() << endl;          currentOrder->setStatus("FAILED\_PAYMENT\_ERROR");          saveOrderToFile(\*currentOrder);          currentOrder->setProcessed(true);          return;      }      bool allSellersPaid = true;      for (const auto &item : currentOrder->getItems())      {          Product \*product = store.findProductByName(item.productId);          // 更新商品库存          product->setQuantity(product->getQuantity() - item.quantity);          // 向卖家转账          User \*seller = User::findUser(allUsers, item.sellerUsername);          if (seller)          {              double itemAmount = item.priceAtPurchase \* item.quantity;              if (!seller->deposit(itemAmount))              {                  cerr << "警告: 向商家 " << item.sellerUsername                       << " 支付商品 " << item.productName                       << " 的款项 ¥" << fixed << setprecision(2) << itemAmount                       << " 失败。订单 ID: " << currentOrder->getOrderId() << endl;                  allSellersPaid = false;              }              else              {                  cout << "已向商家 " << item.sellerUsername                       << " 支付 ¥" << fixed << setprecision(2) << itemAmount << endl;              }          }          else if (!item.sellerUsername.empty())          {              cerr << "警告: 找不到商品 \"" << item.productName                   << "\" 的商家 \"" << item.sellerUsername                   << "\"。订单 ID: " << currentOrder->getOrderId() << endl;              allSellersPaid = false;          }      }      // 保存所有更改（用户和商店）      User::saveUsersToFile(allUsers, USER\_FILE);      store.saveAllProducts();      // 设置最终订单状态      if (allSellersPaid)      {          currentOrder->setStatus("COMPLETED");      }      else      {          currentOrder->setStatus("COMPLETED\_WITH\_PAYMENT\_ISSUES");      }      cout << "订单 ID: " << currentOrder->getOrderId()           << " 处理完成。最终状态: " << currentOrder->getStatus() << endl;      currentOrder->displaySummary();      saveOrderToFile(\*currentOrder);      // 标记订单已处理完成（关键步骤！）      currentOrder->setProcessed(true);  }  // 将订单保存到文件  bool OrderManager::saveOrderToFile(const Order &order) const  {      if (completedOrdersDirectory.empty())      {          std::cerr << "错误: 未设置订单目录。" << std::endl;          return false;      }      // 创建基于订单ID的文件名      std::string filePath = completedOrdersDirectory + "/" + order.getOrderId() + ".txt";      std::ofstream outFile(filePath);      if (!outFile)      {          std::cerr << "错误: 无法打开文件保存订单: " << filePath << std::endl;          return false;      }      // 保存订单头部信息      outFile << order.toStringForSaveHeader();      // 保存订单项      for (const auto &item : order.getItems())      {          outFile << item.toStringForSave() << std::endl;      }      outFile.close();      std::cout << "订单 " << order.getOrderId() << " 已保存到 " << filePath << std::endl;      return true;  }  // 提交订单到处理队列 (线程安全版本)  std::shared\_ptr<Order> OrderManager::submitOrderRequest(const Order &orderRequest)  {      // 创建订单的共享指针副本      auto orderPtr = std::make\_shared<Order>(orderRequest);      // 设置订单状态为等待队列处理      orderPtr->setStatus("PENDING\_IN\_QUEUE");      orderPtr->setProcessed(false);      // 锁定队列      std::lock\_guard<std::mutex> lock(queueMutex);      // 将订单加入队列      orderQueue.push\_back(orderPtr);      // 通知处理线程有新订单      queueCondVar.notify\_one();      cout << "订单 ID: " << orderPtr->getOrderId()           << " 已提交到处理队列，当前队列长度: " << orderQueue.size() << endl;      // 返回订单共享指针      return orderPtr;  }  // 处理队列中的一个订单 (公共方法版，委托给内部方法)  void OrderManager::processNextOrder(Store &store, std::vector<User \*> &allUsers)  {      std::shared\_ptr<Order> orderToProcess;      bool hasOrder = false;      {          std::lock\_guard<std::mutex> lock(queueMutex);          if (!orderQueue.empty())          {              orderToProcess = orderQueue.front();              orderQueue.pop\_front();              hasOrder = true;          }      }      if (hasOrder)      {          processNextOrderInternal(orderToProcess, store, allUsers);      }  }  // 处理所有待处理订单 (线程安全版本)  void OrderManager::processAllPendingOrders(Store &store, std::vector<User \*> &allUsers)  {      std::vector<std::shared\_ptr<Order>> ordersToProcess;      // 首先，从队列中安全地取出所有订单      {          std::lock\_guard<std::mutex> lock(queueMutex);          while (!orderQueue.empty())          {              ordersToProcess.push\_back(orderQueue.front());              orderQueue.pop\_front();          }      }      // 如果没有订单，直接返回      if (ordersToProcess.empty())      {          cout << "当前没有待处理订单。" << endl;          return;      }      cout << "\n--- 开始处理 " << ordersToProcess.size() << " 个待处理订单 ---" << endl;      // 处理获取的订单      for (auto &order : ordersToProcess)      {          processNextOrderInternal(order, store, allUsers);      }      cout << "--- 成功处理 " << ordersToProcess.size() << " 个订单 ---" << endl;  }  // 获取待处理订单数量 (线程安全版本)  size\_t OrderManager::getPendingOrderCount() const  {      std::lock\_guard<std::mutex> lock(queueMutex);      return orderQueue.size();  }  // 显示待处理订单 (线程安全版本)  void OrderManager::displayPendingOrders() const  {      std::lock\_guard<std::mutex> lock(queueMutex);      if (orderQueue.empty())      {          cout << "当前队列中没有待处理订单。" << endl;          return;      }      cout << "\n--- 待处理订单队列 ---" << endl;      int i = 1;      for (const auto &order : orderQueue)      {          cout << i++ << ". 订单 ID: " << order->getOrderId()               << ", 客户: " << order->getCustomerUsername()               << ", 金额: ¥" << fixed << setprecision(2) << order->getTotalAmount()               << ", 状态: " << order->getStatus() << endl;      }      cout << "-----------------------------" << endl;  }  // 获取指定用户的所有订单  void OrderManager::getUserOrders(const std::string &username,                                   std::vector<std::shared\_ptr<Order>> &pendingOrders,                                   std::vector<std::shared\_ptr<Order>> &completedOrders)  {      // 清空传入的向量      pendingOrders.clear();      completedOrders.clear();      // 获取队列中的待处理订单      {          std::lock\_guard<std::mutex> lock(queueMutex);          for (const auto &order : orderQueue)          {              if (order->getCustomerUsername() == username)              {                  pendingOrders.push\_back(order);              }          }      }      // 获取已完成的订单      try      {          // 检查订单目录是否存在          if (!std::filesystem::exists(completedOrdersDirectory))          {              std::cerr << "订单目录不存在: " << completedOrdersDirectory << std::endl;              return;          }          // 遍历订单目录中的所有文件          for (const auto &entry : std::filesystem::directory\_iterator(completedOrdersDirectory))          {              if (entry.is\_regular\_file() && entry.path().extension() == ".txt")              {                  // 从文件名中提取订单ID                  std::string filename = entry.path().filename().string();                  if (filename.find("ORD-") == 0) // 确保是订单文件                  {                      // 尝试读取订单文件                      std::ifstream file(entry.path());                      if (file.is\_open())                      {                          std::string line;                          bool isUserOrder = false;                          std::string orderId, status;                          double totalAmount = 0.0;                          time\_t timestamp = 0;                          std::string productId;                          std::string productName;                          int quantity;                          double priceAtPurchase;                          std::string sellerUsername;                          // 读取订单头部信息                          for (int i = 0; i < 6 && std::getline(file, line); i++)                          {                              switch (i)                              {                              case 0:                                  orderId = line.substr(8); // 识别从传参开始到末尾的字符串                                  break;                              case 1:                                  if (line.substr(17) == username)                                  {                                      isUserOrder = true;                                  }                                  else                                  {                                      isUserOrder = false;                                      break; // 如果不是用户的订单，则跳出循环                                  }                                  break;                              case 2:                                  timestamp = std::stoll(line.substr(10));                                  break;                              case 3:                                  status = line.substr(7);                                  break;                              case 4:                                  totalAmount = std::stod(line.substr(12));                                  break;                              case 5:                                  quantity = std::stod(line.substr(11));                                  break;                              }                          } // 如果是用户的订单，则创建一个新的订单对象并添加到相应列表                          std::vector<OrderItem> items;                          // 读取订单中的所有商品项                          while (std::getline(file, line))                          {                              std::istringstream iss(line);                              std::string part;                              std::vector<std::string> parts;                              // 解析逗号分隔的值                              while (std::getline(iss, part, ','))                              {                                  parts.push\_back(part);                              }                              // 确保解析了足够的部分                              if (parts.size() >= 5)                              {                                  productId = parts[0];                                  productName = parts[1];                                  quantity = std::stoi(parts[2]);                                  priceAtPurchase = std::stod(parts[3]);                                  sellerUsername = parts[4];                                  // 创建OrderItem并添加到items向量                                  OrderItem item(productId, productName, quantity, priceAtPurchase, sellerUsername);                                  items.push\_back(item);                              }                          }                          if (isUserOrder)                          {                              auto order = std::make\_shared<Order>(username);                              order->setStatus(status);                              // 如果有订单ID，设置它（否则会生成新的ID）                              if (!orderId.empty())                              {                                  order->setOrderId(orderId);                              }                              if (totalAmount > 0)                              {                                  // 设置总金额                                  order->setTotalAmount(totalAmount); // 先计算一次                              }                              // 如果有时间戳，设置它                              if (timestamp > 0)                              {                                  // 我们已经添加了 setTimestamp 方法                                  order->setTimestamp(timestamp);                              }                              // 添加所有解析出的订单项到订单中                              for (const auto &item : items)                              {                                  order->addItem(item);                              }                              // 添加到已完成订单列表                              // 根据状态判断是已完成还是失败/取消                              if (status.find("COMPLETED") == 0 ||                                  status.find("FAILED") == 0 ||                                  status.find("CANCELLED") == 0)                              {                                  completedOrders.push\_back(order);                              }                          }                      }                  }              }          }      }      catch (const std::exception &e)      {          std::cerr << "读取订单文件时出错: " << e.what() << std::endl;      }      // 按时间戳排序（可选）      // 使用显式的标准库排序算法      ::std::sort(pendingOrders.begin(), pendingOrders.end(),                  [](const std::shared\_ptr<Order> &a, const std::shared\_ptr<Order> &b)                  {                      return a->getTimestamp() > b->getTimestamp(); // 从新到旧                  });      ::std::sort(completedOrders.begin(), completedOrders.end(),                  [](const std::shared\_ptr<Order> &a, const std::shared\_ptr<Order> &b)                  {                      return a->getTimestamp() > b->getTimestamp(); // 从新到旧                  });  } |

## 综合3.cpp

### Terminal

#### Server\_run.cpp

|  |
| --- |
| #include <iostream>  #include <csignal>  #include "server/server.h"  // 全局变量，用于信号处理  Server \*g\_server = nullptr;  // 信号处理函数，用于优雅地关闭服务器  void signalHandler(int signal)  {      std::cout << "接收到信号: " << signal << std::endl;      if (g\_server)      {          std::cout << "正在停止服务器..." << std::endl;          g\_server->stop();      }  }  int main()  {      // 设置信号处理      signal(SIGINT, signalHandler);  // 处理Ctrl+C      signal(SIGTERM, signalHandler); // 处理终止信号      // 创建服务器实例      Server server;      g\_server = &server;      // 初始化服务器      if (!server.initialize())      {          std::cerr << "服务器初始化失败！" << std::endl;          return 1;      }      // 启动服务器      server.start();      std::cout << "服务器已关闭。" << std::endl;      return 0;  } |

#### Client\_run.cpp

|  |
| --- |
| #include "client/client.h"  /\*\*   \* @file client\_run.cpp   \* @brief 电子商城系统客户端入口点   \*   \* 本文件提供了电子商城系统客户端的入口点，实例化并运行Client类。   \* 该程序通过重构，将原来使用全局变量和函数的设计转变为面向对象的设计。   \*/  int main()  {      // 创建客户端实例      Client client;      // 初始化客户端      if (!client.initialize())      {          std::cerr << "客户端初始化失败" << std::endl;          return 1;      }      // 运行客户端      client.run();      // 退出程序      return 0;  } |

#### Global.cpp

|  |
| --- |
| #include <string>  #include "ordermanager/ordermanager.h"  // 全局变量定义  // 这些变量被多个源文件引用，在重构过程中保持兼容性  namespace  {      // 这些变量仅在此文件中可见，但我们提供全局访问函数      std::string g\_userFile = "./user/users.txt";      std::string g\_storeFile = "./store";      std::string g\_cartFile = "./user/carts";      std::string g\_orderDir = "./order/orders";  }  // 全局变量声明  // 使用 extern 使这些变量在其他源文件中可见  extern const std::string USER\_FILE = g\_userFile;  extern const std::string STORE\_FILE = g\_storeFile;  extern const std::string CART\_FILE = g\_cartFile;  extern const std::string ORDER\_DIR = g\_orderDir;  // 全局订单管理器对象  OrderManager g\_orderManager(ORDER\_DIR); |

#### User.h

|  |
| --- |
| #ifndef USER\_H  #define USER\_H  #include <iostream>  #include <string>  #include <vector>  #include <fstream>  #include <limits>  #include <iomanip>          // For output formatting in cart display  #include "../order/order.h" // For OrderItem  // 前向声明  class Product;  class Store;  // 购物车中的商品项 (现在作为 Customer 类的内部结构或辅助结构)  struct CartItem  {      std::string productId; // 商品名称或唯一ID      std::string productName;      int quantity;      double priceAtAddition; // 添加到购物车时的价格      std::string sellerUsername;      CartItem(std::string pid = "", std::string pName = "", int qty = 0, double price = 0.0, std::string sName = "")          : productId(pid), productName(pName), quantity(qty), priceAtAddition(price), sellerUsername(sName) {}  };  class User  {  protected:      std::string username;      std::string password;      std::string userType;      double balance;      static void clearInputBuffer()      {          std::cin.clear();          std::cin.ignore(std::numeric\_limits<std::streamsize>::max(), '\n');      }  public:      User(std::string username, std::string password, double balance = 0.0,           bool is\_customer = false, bool is\_seller = false, bool is\_admin = false);      User();      virtual ~User() = default;      virtual std::string getUserType() const = 0;      void displayInfo(int index = -1) const;      bool changePassword(const std::string &oldPassword, const std::string &newPassword);      bool deposit(double amount);      bool withdraw(double amount);      double checkBalance() const;      static User \*registerUser(std::vector<User \*> &users);      static User \*userLogin(const std::vector<User \*> &users);      static bool saveUsersToFile(const std::vector<User \*> &users, const std::string &filename);      static std::vector<User \*> loadUsersFromFile(const std::string &filename);      static User \*findUser(const std::vector<User \*> &users, const std::string &username);      static void displayAllUsers(const std::vector<User \*> &users);      static bool isUsernameExists(const std::vector<User \*> &users, const std::string &username);      std::string getUsername() const;      std::string getPassword() const;      void setUsername(const std::string &newUsername);      void setPassword(const std::string &newPassword);      void setBalance(double newBalance);      static User \*login(const std::vector<User \*> &users, const std::string &username, const std::string &password);  };  // 消费者类  class Customer : public User  {  private:      std::string cartDirectoryPath; // 存储购物车文件的目录路径      // 购物车文件操作的辅助方法      std::string getCartFilePath() const;  public:      Customer(std::string username, std::string password, const std::string &cartDir, double balance = 0.0);      Customer(const std::string &cartDir); // 默认构造函数      std::string getUserType() const override { return "customer"; }      std::vector<CartItem> shoppingCartItems; // 直接在 Customer 中存储购物车项      void loadCartFromFile();      bool saveCartToFile() const;      // 购物车管理方法      bool addToCart(const Product &product, int quantity);      bool removeCartItem(const std::string &productName);      bool updateCartItemQuantity(const std::string &productName, int newQuantity, Store &store); // 需要 store 检查库存      void viewCart(Store &store, std::vector<User \*> &allUsers);                                 // 查看并管理购物车，结算也在这里处理      bool isCartEmpty() const;      void clearCartAndFile(); // 结算后清空购物车及文件  };  // 商家类  class Seller : public User  {  public:      Seller(std::string username, std::string password, double balance = 0.0);      Seller();      std::string getUserType() const override { return "seller"; }  };  // 管理员类  class Admin : public User  {  public:      Admin(std::string username, std::string password, double balance = 0.0);      Admin();      std::string getUserType() const override { return "admin"; }  };  #endif // USER\_H |

#### User.cpp

|  |
| --- |
| #include "user.h"  #include "../store/store.h" // 需要 Product 和 Store 的完整定义  #include "../order/order.h"  #include "../ordermanager/ordermanager.h"  #include <iostream>  #include <limits>  #include <string>  #include <fstream>  #include <sstream>  #include <filesystem> // 用于文件操作  #include <algorithm>  // for std::remove\_if  #include <iomanip>    // for std::fixed, std::setprecision  #include <thread>  using namespace std;  // 声明在 start.cpp 中定义的全局常量  extern const string CART\_FILE; // 用于 Customer 构造函数  extern const string USER\_FILE; // 用于保存用户信息  // User 构造函数实现  User::User(std::string u\_name, std::string pwd, double bal, bool is\_customer, bool is\_seller, bool is\_admin)      : username(u\_name), password(pwd), balance(bal)  {      // userType 在派生类中设置  }  User::User() : username(""), password(""), balance(0.0), userType("unknown") {}  // --- Customer 类购物车相关方法实现 ---  std::string Customer::getCartFilePath() const  {      if (this->username.empty())      { // 防止用户名为空时生成无效路径          // cerr << "警告: Customer 用户名为空，无法生成购物车路径。" << endl;          return ""; // 或者抛出异常，或者返回一个默认的无效路径      }      return this->cartDirectoryPath + "/" + this->username + "\_cart.txt";  }  void Customer::loadCartFromFile()  {      shoppingCartItems.clear();      std::string cartPath = getCartFilePath();      if (cartPath.empty())          return; // 用户名为空，无法加载      std::ifstream file(cartPath);      if (!file.is\_open())      {          return; // 文件不存在或无法打开是正常情况（购物车为空）      }      std::string line;      while (getline(file, line))      {          std::stringstream ss(line);          std::string productId, productName, sellerUsernameStr;          int quantity;          double priceAtAddition;          if (getline(ss, productId, ',') &&              getline(ss, productName, ',') &&              (ss >> quantity) &&              (ss.ignore(1) && ss >> priceAtAddition) &&              (ss.ignore(1) && getline(ss, sellerUsernameStr)))          {              shoppingCartItems.emplace\_back(productId, productName, quantity, priceAtAddition, sellerUsernameStr);          }          else          {              std::cerr << "警告: 购物车文件 " << cartPath << " 中存在格式错误的行: " << line << std::endl;          }      }      file.close();  }  bool Customer::saveCartToFile() const  {      std::string cartPath = getCartFilePath();      if (cartPath.empty())      {          std::cerr << "错误: Customer 用户名为空，无法保存购物车。" << std::endl;          return false;      }      std::ofstream file(cartPath); // 会覆盖旧文件      if (!file.is\_open())      {          std::cerr << "错误: 无法打开购物车文件进行写入: " << cartPath << std::endl;          return false;      }      for (const auto &item : shoppingCartItems)      {          file << item.productId << ","               << item.productName << ","               << item.quantity << ","               << item.priceAtAddition << ","               << item.sellerUsername << std::endl;      }      file.close();      return true;  }  // Customer 构造函数实现  Customer::Customer(std::string uname, std::string pwd, const std::string &cartDir, double bal)      : User(uname, pwd, bal, true, false, false), cartDirectoryPath(cartDir)  {      this->userType = "customer";      if (!this->username.empty())      { // 只有当用户名有效时才加载购物车          loadCartFromFile();      }  }  // 默认构造函数  Customer::Customer(const std::string &cartDir)      : User(), cartDirectoryPath(cartDir)  {      this->userType = "customer";      // 注意: 此时 username 为空，loadCartFromFile 不会执行。      // 如果后续通过 setUsername 设置了用户名，需要一种机制来加载对应的购物车。      // 或者，强制 Customer 总是在有用户名的情况下创建。  }  bool Customer::addToCart(const Product &product, int quantity)  {      if (this->username.empty())      {          cout << "错误：无法为匿名用户添加购物车。" << endl;          return false;      }      if (quantity <= 0)      {          cout << "添加数量必须大于0。" << endl;          return false;      }      bool found = false;      for (auto &item : shoppingCartItems)      {          if (item.productId == product.getName())          { // 假设商品名称是唯一ID              item.quantity += quantity;              // 可选：更新 priceAtAddition 为最新价格或保持不变              found = true;              break;          }      }      if (!found)      {          shoppingCartItems.emplace\_back(product.getName(), product.getName(), quantity, product.getPrice(), product.getSellerUsername());      }      cout << "\"" << product.getName() << "\" 已成功加入购物车。" << endl;      return saveCartToFile();  }  bool Customer::removeCartItem(const std::string &productName)  {      if (this->username.empty())          return false;      auto initial\_size = shoppingCartItems.size();      shoppingCartItems.erase(std::remove\_if(shoppingCartItems.begin(), shoppingCartItems.end(),                                             [&](const CartItem &item)                                             { return item.productName == productName; }),                              shoppingCartItems.end());      if (shoppingCartItems.size() < initial\_size)      {          cout << "商品 \"" << productName << "\" 已从购物车移除。" << endl;          return saveCartToFile();      }      else      {          cout << "未在购物车中找到商品: " << productName << endl;          return false;      }  }  bool Customer::updateCartItemQuantity(const std::string &productName, int newQuantity, Store &store)  {      if (this->username.empty())          return false;      if (newQuantity < 0)      {          cout << "数量不能为负。" << endl;          return false;      }      bool updated = false;      for (auto &item : shoppingCartItems)      {          if (item.productName == productName)          {              Product \*p\_info = store.findProductByName(productName); // 检查最新库存              if (p\_info && newQuantity > p\_info->getQuantity())              {                  cout << "库存不足！无法将购物车中 \"" << productName << "\" 的数量修改为 " << newQuantity                       << "。当前库存: " << p\_info->getQuantity() << endl;                  return false;              }              item.quantity = newQuantity;              updated = true;              break;          }      }      if (!updated)      {          cout << "未在购物车中找到商品: " << productName << endl;          return false;      }      // 移除数量为0的商品      shoppingCartItems.erase(std::remove\_if(shoppingCartItems.begin(), shoppingCartItems.end(),                                             [](const CartItem &item)                                             { return item.quantity <= 0; }),                              shoppingCartItems.end());      cout << "购物车商品 \"" << productName << "\" 数量已更新。" << endl;      return saveCartToFile();  }  void Customer::clearCartAndFile()  {      if (this->username.empty())          return;      shoppingCartItems.clear();      std::string cartPath = getCartFilePath();      if (!cartPath.empty() && std::filesystem::exists(cartPath))      {          try          {              std::filesystem::remove(cartPath);          }          catch (const std::filesystem::filesystem\_error &err)          {              std::cerr << "错误: 删除购物车文件失败: " << err.what() << std::endl;          }      }  }  bool Customer::isCartEmpty() const  {      return shoppingCartItems.empty();  }  void Customer::viewCart(Store &store, std::vector<User \*> &allUsers)  {      if (this->username.empty())      {          cout << "错误：匿名用户无法查看购物车。" << endl;          return;      }      // 确保购物车是最新的（如果其他地方可能修改文件但未更新内存）      // loadCartFromFile(); // 通常在每次操作后保存，所以这里可能不需要，除非有外部修改购物车的可能      if (shoppingCartItems.empty())      {          cout << "\n您的购物车是空的。" << endl;          return;      }      double totalCartPriceBasedOnCurrent = 0;      cout << "\n--- 我的购物车 ---" << endl;      cout << std::fixed << std::setprecision(2); // 设置输出格式      for (size\_t i = 0; i < shoppingCartItems.size(); ++i)      {          const auto &item = shoppingCartItems[i];          Product \*p\_info = store.findProductByName(item.productId);          double currentItemPrice = p\_info ? p\_info->getPrice() : item.priceAtAddition;          cout << i + 1 << ". 商品: " << item.productName               << " | 数量: " << item.quantity               << " | 当前单价: ¥" << currentItemPrice               << " | 小计: ¥" << (item.quantity \* currentItemPrice) << endl;          if (p\_info)          {              if (p\_info->getQuantity() < item.quantity)              {                  cout << "   注意: " << item.productName << " 当前库存(" << p\_info->getQuantity() << ")不足购物车数量(" << item.quantity << ")!" << endl;              }              if (p\_info->getPrice() != item.priceAtAddition)              {                  cout << "   提示: " << item.productName << " 加入时价格为 ¥" << item.priceAtAddition << ", 当前价格已变为 ¥" << p\_info->getPrice() << endl;              }          }          else          {              cout << "   警告: 商品 " << item.productName << " 可能已从商店下架!" << endl;          }          totalCartPriceBasedOnCurrent += item.quantity \* currentItemPrice;      }      cout << "--------------------" << endl;      cout << "购物车总计 (按当前价格): ¥" << totalCartPriceBasedOnCurrent << endl;      cout << "--------------------" << endl;      int cartChoice = -1;      do      {          cout << "\n购物车操作:" << endl;          cout << "1. 修改商品数量" << endl;          cout << "2. 移除商品" << endl;          cout << "3. 生成订单并结算" << endl;          cout << "0. 返回商城" << endl;          cout << "请选择: ";          if (!(cin >> cartChoice))          {              cout << "无效输入。" << endl;              User::clearInputBuffer();              cartChoice = -1;              continue;          }          User::clearInputBuffer();          switch (cartChoice)          {          case 1:          {              string nameToUpdate;              int newQty;              cout << "输入要修改数量的商品名称: ";              getline(cin >> ws, nameToUpdate);              cout << "输入新的数量 (输入0将移除该商品): ";              if (!(cin >> newQty))              {                  cout << "无效数量输入。" << endl;                  User::clearInputBuffer();                  break;              }              User::clearInputBuffer();              if (updateCartItemQuantity(nameToUpdate, newQty, store))              {                  // 成功后重新加载并显示购物车 (递归调用或重新加载数据)                  // 为了避免无限递归，这里直接返回，让外层循环重新调用 viewCart                  cout << "购物车已更新。" << endl;              }              return viewCart(store, allUsers); // 重新显示更新后的购物车          }          case 2:          {              string nameToRemove;              cout << "输入要移除的商品名称: ";              getline(cin >> ws, nameToRemove);              if (removeCartItem(nameToRemove))              {                  cout << "购物车已更新。" << endl;              }              return viewCart(store, allUsers); // 重新显示更新后的购物车          }          case 3: // Generate Order and Checkout          {              if (shoppingCartItems.empty())              {                  cout << "购物车是空的，无法生成订单。" << endl;                  break;              }              Order currentOrder(this->getUsername()); // Create a new order object              currentOrder.setStatus("PENDING\_VALIDATION");              // 第一阶段：验证商品并构建订单（预检查）              bool preCheckOk = true;              for (const auto &cart\_item : shoppingCartItems)              {                  Product \*product = store.findProductByName(cart\_item.productId);                  if (!product)                  {                      cout << "错误: 购物车商品 \"" << cart\_item.productName << "\" 已不存在。" << endl;                      currentOrder.setStatus("CANCELLED\_PRODUCT\_NOT\_FOUND");                      preCheckOk = false;                      break;                  }                  if (product->getQuantity() < cart\_item.quantity)                  {                      cout << "错误: 购物车商品 \"" << cart\_item.productName << "\" 库存不足 (需要 "                           << cart\_item.quantity << ", 现有 " << product->getQuantity() << ")。" << endl;                      currentOrder.setStatus("CANCELLED\_INSUFFICIENT\_STOCK");                      preCheckOk = false;                      break;                  }                  currentOrder.addItemFromProduct(\*product, cart\_item.quantity);              }              if (!preCheckOk)              {                  currentOrder.displaySummary(); // Display order with cancelled status                  cout << "订单无法继续，请调整购物车后重试。" << endl;                  break; // Back to cart menu              }              // 第二阶段：验证客户余额（只是初步检查，队列处理时还会再次检查）              if (this->checkBalance() < currentOrder.getTotalAmount())              {                  cout << "错误: 您的余额不足以支付此订单！" << endl;                  currentOrder.setStatus("CANCELLED\_INSUFFICIENT\_FUNDS");                  // currentOrder.displaySummary();                  cout << "您的余额:   ¥" << this->checkBalance() << endl;                  break; // Back to cart menu              }              // 第三阶段：显示订单并请求确认              currentOrder.setStatus("PENDING\_CONFIRMATION");              currentOrder.displaySummary();              cout << "您的当前余额: ¥" << std::fixed << std::setprecision(2) << this->checkBalance() << endl;              cout << "支付后余额将为: ¥" << (this->checkBalance() - currentOrder.getTotalAmount()) << endl;              cout << "--------------------" << endl;              char confirmChoice;              cout << "确认支付并完成订单吗? (y/n): ";              cin >> confirmChoice;              User::clearInputBuffer();              if (tolower(confirmChoice) != 'y')              {                  currentOrder.setStatus("CANCELLED\_BY\_USER");                  currentOrder.displaySummary();                  cout << "订单已取消。" << endl;                  break; // Back to cart menu              }              // Stage 4: Process payment and update inventory (Actual "lock" and transaction)              cout << "\n正在处理订单，请稍候..." << endl;              currentOrder.setStatus("PROCESSING");              // 访问OrderManager，提交订单 - 修改这部分              extern OrderManager g\_orderManager;              auto submittedOrder = g\_orderManager.submitOrderRequest(currentOrder);              cout << "订单已成功提交到处理队列，订单ID: " << submittedOrder->getOrderId() << endl;              cout << "正在等待订单处理，请稍候..." << endl;              // 等待订单处理完成              int waitCounter = 0;              const int MAX\_WAIT\_CYCLES = 100; // 最多等待10秒              while (!submittedOrder->getProcessed() && waitCounter < MAX\_WAIT\_CYCLES)              {                  // 短暂休眠，避免过度消耗CPU                  std::this\_thread::sleep\_for(std::chrono::milliseconds(100));                  waitCounter++;                  // 每秒显示等待状态                  if (waitCounter % 10 == 0)                  {                      cout << "已等待 " << (waitCounter / 10) << " 秒..." << endl;                  }              }              // 检查是否超时              if (waitCounter >= MAX\_WAIT\_CYCLES)              {                  cout << "等待订单处理超时！订单可能仍在处理中。" << endl;              }              else              {                  cout << "订单已处理完成! 状态: " << submittedOrder->getStatus() << endl;                  if (submittedOrder->getStatus() == "COMPLETED" ||                      submittedOrder->getStatus() == "COMPLETED\_WITH\_PAYMENT\_ISSUES")                  {                      clearCartAndFile(); // 成功提交后清空购物车                  }              }              cout << std::resetiosflags(std::ios\_base::fixed | std::ios\_base::floatfield);              return; // 退出viewCart，返回Customer\_page商店菜单              // if (!this->withdraw(currentOrder.getTotalAmount()))              // {              //     currentOrder.setStatus("FAILED\_PAYMENT\_WITHDRAWAL");              //     currentOrder.displaySummary();              //     cout << "错误: 支付失败。请检查余额或联系客服。" << endl;              //     // Potentially log this critical failure              //     break;              // }              // for (const auto &order\_item : currentOrder.getItems())              // {              //     Product \*product = store.findProductByName(order\_item.productId); // Re-fetch, though should be same              //     if (product)              //     { // Should always be true due to pre-checks              //         product->setQuantity(product->getQuantity() - order\_item.quantity);              //         User \*seller = User::findUser(allUsers, order\_item.sellerUsername);              //         if (seller)              //         {              //             if (!seller->deposit(order\_item.priceAtPurchase \* order\_item.quantity))              //             {              //                 cerr << "警告: 向商家 " << seller->getUsername() << " 转账失败 (商品: "              //                      << order\_item.productName << "). 订单号: " << currentOrder.getOrderId() << endl;              //                 // Log this for reconciliation              //             }              //         }              //         else if (!order\_item.sellerUsername.empty())              //         {              //             cerr << "警告: 未找到商品 \"" << order\_item.productName << "\" 的商家 \""              //                  << order\_item.sellerUsername << "\". 订单号: " << currentOrder.getOrderId() << endl;              //         }              //         else              //         {              //             cerr << "警告: 商品 \"" << order\_item.productName << "\" 没有商家信息. 订单号: " << currentOrder.getOrderId() << endl;              //         }              //     }              // }              // User::saveUsersToFile(allUsers, USER\_FILE);              // store.saveAllProducts(); // Or save only affected sellers              // currentOrder.setStatus("COMPLETED");              // cout << "\n订单处理完成！" << endl;              // currentOrder.displaySummary(); // Display final completed order              // // Optional: currentOrder.saveToFile(ORDER\_DIRECTORY\_BASE + "/" + this->getUsername());              // clearCartAndFile(); // Clear cart after successful order              // cout << std::resetiosflags(std::ios\_base::fixed | std::ios\_base::floatfield);              // return; // Exit viewCart, back to Customer\_page store menu          }          case 0:              // 返回商城，什么也不做，循环会结束              break;          default:              cout << "无效选项。" << endl;          }      } while (cartChoice != 0);      cout << std::resetiosflags(std::ios\_base::fixed | std::ios\_base::floatfield); // 重置输出格式  }  // Seller 构造函数实现  Seller::Seller(std::string uname, std::string pwd, double bal)      : User(uname, pwd, bal, false, true, false)  {      this->userType = "seller";  }  Seller::Seller() : User() { this->userType = "seller"; }  // Admin 构造函数实现  Admin::Admin(std::string uname, std::string pwd, double bal)      : User(uname, pwd, bal, false, false, true)  {      this->userType = "admin";  }  Admin::Admin() : User() { this->userType = "admin"; }  // --- 静态方法：用户注册实现 ---  User \*User::registerUser(std::vector<User \*> &users)  {      string newUsername, newPassword;      double initialBalance = 0.0;      int userTypeChoice;      cout << "请输入新用户名: ";      getline(cin >> ws, newUsername);      if (isUsernameExists(users, newUsername))      {          cout << "用户名已存在！" << endl;          return nullptr;      }      cout << "请输入密码: ";      getline(cin >> ws, newPassword);      cout << "选择用户类型 (1: 消费者, 2: 商家): ";      if (!(cin >> userTypeChoice) || (userTypeChoice != 1 && userTypeChoice != 2))      {          cout << "无效的用户类型选择。" << endl;          clearInputBuffer();          return nullptr;      }      clearInputBuffer();      User \*newUser = nullptr;      if (userTypeChoice == 1)      {                                                                                // Customer          newUser = new Customer(newUsername, newPassword, CART\_FILE, initialBalance); // 使用 CART\_FILE      }      else      { // Seller          newUser = new Seller(newUsername, newPassword, initialBalance);      }      if (newUser)      {          users.push\_back(newUser);          cout << (userTypeChoice == 1 ? "消费者" : "商家") << "用户 \"" << newUsername << "\" 注册成功！" << endl;      }      return newUser;  }  // 从文件加载用户信息  std::vector<User \*> User::loadUsersFromFile(const std::string &filename)  {      std::vector<User \*> loadedUsers;      std::ifstream file(filename);      if (!file.is\_open())      {          return loadedUsers;      }      std::string line;      while (getline(file, line))      {          std::stringstream ss(line);          std::string uname, pwd, type;          double bal;          if (              getline(ss, uname, ',') &&              getline(ss, pwd, ',') &&              getline(ss, type, ',') &&              (ss >> bal))          {              if (type == "customer")              {                  loadedUsers.push\_back(new Customer(uname, pwd, CART\_FILE, bal)); // 使用 CART\_FILE              }              else if (type == "seller")              {                  loadedUsers.push\_back(new Seller(uname, pwd, bal));              }              else if (type == "admin")              {                  loadedUsers.push\_back(new Admin(uname, pwd, bal));              }          }      }      file.close();      return loadedUsers;  }  // --- User 类其他成员函数实现 ---  // getUsername, getPassword, displayInfo, changePassword, deposit, withdraw, checkBalance,  // saveUsersToFile, findUser, isUsernameExists, login  // 这些方法的实现应该保持不变，除非它们需要与新的购物车逻辑交互（但看起来不需要）。  std::string User::getUsername() const { return username; }  std::string User::getPassword() const { return password; } // 密码不应这样直接返回，仅为示例  void User::setUsername(const std::string &newUsername) { username = newUsername; }  void User::setPassword(const std::string &newPassword) { password = newPassword; }  void User::setBalance(double newBalance)  {      if (newBalance >= 0)          balance = newBalance;  }  void User::displayInfo(int index) const  {      if (index != -1)      {          cout << "用户 " << index << ": ";      }      cout << "用户名: " << username << ", 类型: " << userType << ", 余额: " << balance << endl;  }  bool User::changePassword(const std::string &oldPassword, const std::string &newPassword)  {      if (this->password == oldPassword)      {          this->password = newPassword;          return true;      }      cout << "原密码错误！" << endl;      return false;  }  bool User::deposit(double amount)  {      if (amount <= 0)      {          cout << "充值金额必须为正！" << endl;          return false;      }      balance += amount;      return true;  }  bool User::withdraw(double amount)  {      if (amount <= 0)      {          cout << "消费金额必须为正！" << endl;          return false;      }      if (balance < amount)      {          cout << "余额不足！" << endl;          return false;      }      balance -= amount;      return true;  }  double User::checkBalance() const  {      return balance;  }  bool User::saveUsersToFile(const std::vector<User \*> &users, const std::string &filename)  {      ofstream file(filename);      if (!file.is\_open())      {          cerr << "错误: 无法打开用户文件进行写入: " << filename << endl;          return false;      }      for (const auto \*user : users)      {          if (user)          {              file << user->getUsername() << ","                   << user->getPassword() << ","                   << user->getUserType() << ","                   << user->checkBalance() << endl;          }      }      file.close();      return true;  }  User \*User::findUser(const std::vector<User \*> &users, const std::string &findUsername)  {      for (User \*user : users)      {          if (user->getUsername() == findUsername)          {              return user;          }      }      return nullptr;  }  bool User::isUsernameExists(const std::vector<User \*> &users, const std::string &checkUsername)  {      return findUser(users, checkUsername) != nullptr;  }  User \*User::userLogin(const std::vector<User \*> &users)  {      string loginUsername, loginPassword;      cout << "请输入用户名: ";      getline(cin >> ws, loginUsername);      cout << "请输入密码: ";      getline(cin >> ws, loginPassword);      User \*foundUser = findUser(users, loginUsername);      if (foundUser && foundUser->getPassword() == loginPassword)      {          cout << "登录成功！欢迎, " << loginUsername << "!" << endl;          // 如果是 Customer，并且之前用户名为空（例如通过默认构造后设置用户名），则加载其购物车          if (Customer \*cust = dynamic\_cast<Customer \*>(foundUser))          {              if (cust->shoppingCartItems.empty() && !cust->getUsername().empty())              {                             // 假设 shoppingCartItems 为空表示未加载                                            // cust->cartDirectoryPath 应该在构造时已设置                  cust->loadCartFromFile(); // 尝试加载购物车              }          }          return foundUser;      }      cout << "用户名或密码错误！" << endl;      return nullptr;  }  User \*User::login(const std::vector<User \*> &users, const std::string &username\_param, const std::string &password\_param)  {      User \*foundUser = findUser(users, username\_param);      if (foundUser && foundUser->getPassword() == password\_param)      {          return foundUser;      }      return nullptr;  }  void User::displayAllUsers(const std::vector<User \*> &users)  {      if (users.empty())      {          cout << "当前没有注册用户。" << endl;          return;      }      cout << "\n--- 所有用户信息 ---" << endl;      for (size\_t i = 0; i < users.size(); ++i)      {          users[i]->displayInfo(i + 1);      }      cout << "--------------------" << endl;  } |

#### Store.h

|  |
| --- |
| #ifndef STORE\_H  #define STORE\_H  #include <string>  #include <vector>  #include <iostream>  #include <fstream>  #include <sstream>  #include <iomanip>  #include <limits>  #include <map>  #include <filesystem>  // 前向声明 User  class User;  // --- Product 基类和派生类 ---  class Product  {  protected:      std::string name;      std::string description;      double originalPrice;      int quantity;      double discountRate;      std::string sellerUsername; // 添加商品所属商家  public:      Product(std::string name, std::string desc, double price, int qty, std::string seller = "")          : name(name), description(desc), originalPrice(price), quantity(qty),            discountRate(0.0), sellerUsername(seller) {}      virtual ~Product() = default;      virtual double getoriginalPrice() const { return originalPrice; }      virtual double getPrice() const { return originalPrice \* (1.0 - discountRate); }      virtual void display() const;      virtual std::string getType() const = 0;      virtual std::string getUserCategory() const { return ""; } // 默认返回空字符串      virtual void save(std::ofstream &ofs) const;      // Getters and Setters      std::string getName() const { return name; }      std::string getDescription() const { return description; }      double getOriginalPrice() const { return originalPrice; }      int getQuantity() const { return quantity; }      double getDiscountRate() const { return discountRate; }      std::string getSellerUsername() const { return sellerUsername; } // 获取商品所属商家      void setName(const std::string &newName) { name = newName; }      void setDescription(const std::string &newDesc) { description = newDesc; }      void setOriginalPrice(double newPrice)      {          if (newPrice >= 0)              originalPrice = newPrice;      }      void setQuantity(int newQuantity)      {          if (newQuantity >= 0)              quantity = newQuantity;      }      void setDiscountRate(double newRate)      {          if (newRate >= 0.0 && newRate <= 1.0)              discountRate = newRate;      }      void setSellerUsername(const std::string &seller) { sellerUsername = seller; } // 设置商品所属商家  };  class Book : public Product  {  private:      std::string author;      std::string isbn;  public:      Book(std::string name, std::string desc, double price, int qty,           std::string auth, std::string isbn\_num, std::string seller = "")          : Product(name, desc, price, qty, seller), author(auth), isbn(isbn\_num) {}      std::string getType() const override { return "Book"; }      std::string getUserCategory() const override { return "Book"; } // 固定类型为 "Book"      void display() const override;      void save(std::ofstream &ofs) const override;      // 新增的 getters      std::string getAuthor() const { return author; }      std::string getIsbn() const { return isbn; }  };  class Clothing : public Product  {  private:      std::string size;      std::string color;  public:      Clothing(std::string name, std::string desc, double price, int qty,               std::string sz, std::string clr, std::string seller = "")          : Product(name, desc, price, qty, seller), size(sz), color(clr) {}      std::string getType() const override { return "Clothing"; }      std::string getUserCategory() const override { return "Clothing"; } // 固定类型为 "Clothing"      void display() const override;      void save(std::ofstream &ofs) const override;      // 新增的 getters      std::string getSize() const { return size; }      std::string getColor() const { return color; }  };  class Food : public Product  {  private:      std::string expirationDate;  public:      Food(std::string name, std::string desc, double price, int qty,           std::string expDate, std::string seller = "")          : Product(name, desc, price, qty, seller), expirationDate(expDate) {}      std::string getType() const override { return "Food"; }      std::string getUserCategory() const override { return "Food"; } // 固定类型为 "Food"      void display() const override;      void save(std::ofstream &ofs) const override;      // 新增的 getters      std::string getExpirationDate() const { return expirationDate; }  };  // 自定义商品类  class GenericProduct : public Product  {  private:      std::string categoryTag; // 用户自定义的分类标签  public:      GenericProduct(std::string name, std::string desc, double price, int qty,                     std::string tag, std::string seller = "")          : Product(name, desc, price, qty, seller), categoryTag(tag) {}      std::string getType() const override { return "Generic"; }           // 固定类型为 "Generic"      std::string getUserCategory() const override { return categoryTag; } // 返回用户自定义标签      void display() const override;      void save(std::ofstream &ofs) const override;      // Getter for categoryTag      std::string getCategoryTag() const { return categoryTag; }      void setCategoryTag(const std::string &tag) { categoryTag = tag; }  };  // --- Store Class ---  class Store  {  private:      std::vector<Product \*> allProducts;                           // 存储所有商家的商品      std::map<std::string, std::vector<Product \*>> sellerProducts; // 每个商家的商品映射      std::string storeDirectory;                                   // 商品文件所在目录      // 辅助方法      std::string getSellerFilename(const std::string &username) const;      bool saveProductsForSeller(const std::string &sellerUsername);      bool ensureDirectoryExists(const std::string &path) const;  public: // 构造函数接收目录名      Store(const std::string &directory);      ~Store();      Product \*findProductByName(const std::string &name, const std::string &sellerUsername = "") const;      // 文件操作      bool loadAllProducts();      bool loadSellerProducts(const std::string &sellerUsername);      bool saveAllProducts();      // 显示功能      void displayAllProducts() const;                                     // 显示所有商品      void displaySellerProducts(const std::string &sellerUsername) const; // 显示指定商家的商品      // 搜索功能      std::vector<Product \*> searchProductsByName(const std::string &searchTerm, const std::string &sellerUsername = "") const;      // 商家商品管理功能      bool createBook(User \*currentUser, const std::string &name, const std::string &desc,                      double price, int qty, const std::string &author, const std::string &isbn);      bool createClothing(User \*currentUser, const std::string &name, const std::string &desc,                          double price, int qty, const std::string &size, const std::string &color);      bool createFood(User \*currentUser, const std::string &name, const std::string &desc,                      double price, int qty, const std::string &expDate);      bool createGenericProduct(User \*currentUser, const std::string &name, const std::string &desc,                                double price, int qty, const std::string &categoryTag);      bool manageProductPrice(User \*currentUser, const std::string &productName, double newPrice);      bool manageProductQuantity(User \*currentUser, const std::string &productName, int newQuantity);      bool manageProductDiscount(User \*currentUser, const std::string &productName, double newDiscount);      bool applyCategoryDiscount(User \*currentUser, const std::string &category, double discount);      // 获取商品      const std::vector<Product \*> &getProducts() const { return allProducts; }      std::vector<Product \*> getSellerProducts(const std::string &sellerUsername) const;      // 获取商家商品的唯一分类      std::vector<std::string> getUniqueCategoriesForSeller(const std::string &sellerUsername) const;  };  #endif // STORE\_H |

#### Store.cpp

|  |
| --- |
| #ifndef STORE\_H  #define STORE\_H  #include <string>  #include <vector>  #include <iostream>  #include <fstream>  #include <sstream>  #include <iomanip>  #include <limits>  #include <map>  #include <filesystem>  // 前向声明 User  class User;  // --- Product 基类和派生类 ---  class Product  {  protected:      std::string name;      std::string description;      double originalPrice;      int quantity;      double discountRate;      std::string sellerUsername; // 添加商品所属商家  public:      Product(std::string name, std::string desc, double price, int qty, std::string seller = "")          : name(name), description(desc), originalPrice(price), quantity(qty),            discountRate(0.0), sellerUsername(seller) {}      virtual ~Product() = default;      virtual double getoriginalPrice() const { return originalPrice; }      virtual double getPrice() const { return originalPrice \* (1.0 - discountRate); }      virtual void display() const;      virtual std::string getType() const = 0;      virtual std::string getUserCategory() const { return ""; } // 默认返回空字符串      virtual void save(std::ofstream &ofs) const;      // Getters and Setters      std::string getName() const { return name; }      std::string getDescription() const { return description; }      double getOriginalPrice() const { return originalPrice; }      int getQuantity() const { return quantity; }      double getDiscountRate() const { return discountRate; }      std::string getSellerUsername() const { return sellerUsername; } // 获取商品所属商家      void setName(const std::string &newName) { name = newName; }      void setDescription(const std::string &newDesc) { description = newDesc; }      void setOriginalPrice(double newPrice)      {          if (newPrice >= 0)              originalPrice = newPrice;      }      void setQuantity(int newQuantity)      {          if (newQuantity >= 0)              quantity = newQuantity;      }      void setDiscountRate(double newRate)      {          if (newRate >= 0.0 && newRate <= 1.0)              discountRate = newRate;      }      void setSellerUsername(const std::string &seller) { sellerUsername = seller; } // 设置商品所属商家  };  class Book : public Product  {  private:      std::string author;      std::string isbn;  public:      Book(std::string name, std::string desc, double price, int qty,           std::string auth, std::string isbn\_num, std::string seller = "")          : Product(name, desc, price, qty, seller), author(auth), isbn(isbn\_num) {}      std::string getType() const override { return "Book"; }      std::string getUserCategory() const override { return "Book"; } // 固定类型为 "Book"      void display() const override;      void save(std::ofstream &ofs) const override;      // 新增的 getters      std::string getAuthor() const { return author; }      std::string getIsbn() const { return isbn; }  };  class Clothing : public Product  {  private:      std::string size;      std::string color;  public:      Clothing(std::string name, std::string desc, double price, int qty,               std::string sz, std::string clr, std::string seller = "")          : Product(name, desc, price, qty, seller), size(sz), color(clr) {}      std::string getType() const override { return "Clothing"; }      std::string getUserCategory() const override { return "Clothing"; } // 固定类型为 "Clothing"      void display() const override;      void save(std::ofstream &ofs) const override;      // 新增的 getters      std::string getSize() const { return size; }      std::string getColor() const { return color; }  };  class Food : public Product  {  private:      std::string expirationDate;  public:      Food(std::string name, std::string desc, double price, int qty,           std::string expDate, std::string seller = "")          : Product(name, desc, price, qty, seller), expirationDate(expDate) {}      std::string getType() const override { return "Food"; }      std::string getUserCategory() const override { return "Food"; } // 固定类型为 "Food"      void display() const override;      void save(std::ofstream &ofs) const override;      // 新增的 getters      std::string getExpirationDate() const { return expirationDate; }  };  // 自定义商品类  class GenericProduct : public Product  {  private:      std::string categoryTag; // 用户自定义的分类标签  public:      GenericProduct(std::string name, std::string desc, double price, int qty,                     std::string tag, std::string seller = "")          : Product(name, desc, price, qty, seller), categoryTag(tag) {}      std::string getType() const override { return "Generic"; }           // 固定类型为 "Generic"      std::string getUserCategory() const override { return categoryTag; } // 返回用户自定义标签      void display() const override;      void save(std::ofstream &ofs) const override;      // Getter for categoryTag      std::string getCategoryTag() const { return categoryTag; }      void setCategoryTag(const std::string &tag) { categoryTag = tag; }  };  // --- Store Class ---  class Store  {  private:      std::vector<Product \*> allProducts;                           // 存储所有商家的商品      std::map<std::string, std::vector<Product \*>> sellerProducts; // 每个商家的商品映射      std::string storeDirectory;                                   // 商品文件所在目录      // 辅助方法      std::string getSellerFilename(const std::string &username) const;      bool saveProductsForSeller(const std::string &sellerUsername);      bool ensureDirectoryExists(const std::string &path) const;  public: // 构造函数接收目录名      Store(const std::string &directory);      ~Store();      Product \*findProductByName(const std::string &name, const std::string &sellerUsername = "") const;      // 文件操作      bool loadAllProducts();      bool loadSellerProducts(const std::string &sellerUsername);      bool saveAllProducts();      // 显示功能      void displayAllProducts() const;                                     // 显示所有商品      void displaySellerProducts(const std::string &sellerUsername) const; // 显示指定商家的商品      // 搜索功能      std::vector<Product \*> searchProductsByName(const std::string &searchTerm, const std::string &sellerUsername = "") const;      // 商家商品管理功能      bool createBook(User \*currentUser, const std::string &name, const std::string &desc,                      double price, int qty, const std::string &author, const std::string &isbn);      bool createClothing(User \*currentUser, const std::string &name, const std::string &desc,                          double price, int qty, const std::string &size, const std::string &color);      bool createFood(User \*currentUser, const std::string &name, const std::string &desc,                      double price, int qty, const std::string &expDate);      bool createGenericProduct(User \*currentUser, const std::string &name, const std::string &desc,                                double price, int qty, const std::string &categoryTag);      bool manageProductPrice(User \*currentUser, const std::string &productName, double newPrice);      bool manageProductQuantity(User \*currentUser, const std::string &productName, int newQuantity);      bool manageProductDiscount(User \*currentUser, const std::string &productName, double newDiscount);      bool applyCategoryDiscount(User \*currentUser, const std::string &category, double discount);      // 获取商品      const std::vector<Product \*> &getProducts() const { return allProducts; }      std::vector<Product \*> getSellerProducts(const std::string &sellerUsername) const;      // 获取商家商品的唯一分类      std::vector<std::string> getUniqueCategoriesForSeller(const std::string &sellerUsername) const;  };  #endif // STORE\_H |

#### Server.h

|  |
| --- |
| #ifndef SERVER\_H  #define SERVER\_H  #include <iostream>  #include <winsock2.h>  #include <ws2tcpip.h>  #include <windows.h>  #include <string>  #include <vector>  #include <thread>  #include <mutex>  #include <atomic>  #include <sstream>  #include <set>  #include <algorithm>  #include <chrono>  #include "../user/user.h"  #include "../store/store.h"  #include "../order/order.h"  #include "../ordermanager/ordermanager.h"  #pragma comment(lib, "ws2\_32.lib")  using namespace std;  class Server  {  private: // 成员变量      SOCKET serverSocket;      std::vector<std::thread> clientThreads;      mutable std::mutex usersMutex;         // 使用mutable允许在const成员函数中修改      mutable std::mutex storeMutex;         // 使用mutable允许在const成员函数中修改      mutable std::mutex loggedInUsersMutex; // 使用mutable允许在const成员函数中修改      std::atomic<bool> serverRunning;      std::set<std::string> loggedInUsers;      // 文件路径常量      const string USER\_FILE;      const string STORE\_FILE;      const string CART\_FILE;      const string ORDER\_DIR;      vector<User \*> users;      Store store;      OrderManager orderManager;      // 处理客户端连接的成员函数      void handleClient(SOCKET clientSocket);      // 解析和处理各种客户端请求的辅助函数      std::string processLoginRequest(const std::string &request);      std::string processRegisterRequest(const std::string &request);      std::string processGetProductsRequest(const std::string &request) const;      std::string processGetProductDetailRequest(const std::string &request) const;      std::string processAddToCartRequest(const std::string &request);      std::string processGetCartRequest(const std::string &request);      std::string processCheckoutRequest(const std::string &request);      std::string processGetSellerInfoRequest(const std::string &request) const;      std::string processChangePasswordRequest(const std::string &request);      std::string processGetSellerProductsRequest(const std::string &request) const;      std::string processAddBookRequest(const std::string &request);      std::string processAddClothingRequest(const std::string &request);      std::string processAddFoodRequest(const std::string &request);      std::string processAddGenericRequest(const std::string &request);      std::string processUpdateProductPriceRequest(const std::string &request);      std::string processUpdateProductQuantityRequest(const std::string &request);      std::string processSetProductDiscountRequest(const std::string &request);      std::string processSetCategoryDiscountRequest(const std::string &request);      std::string processLogoutRequest(const std::string &request);      std::string processCheckBalanceRequest(const std::string &request) const;      std::string processGetUserInfoRequest(const std::string &request) const;      std::string processDepositRequest(const std::string &request);      std::string processSearchProductsRequest(const std::string &request) const;      std::string processDirectPurchaseRequest(const std::string &request);      std::string processUpdateCartItemRequest(const std::string &request);      std::string processRemoveFromCartRequest(const std::string &request);  public:      // 构造函数      Server(const std::string &userFile = "./user/users.txt",             const std::string &storeFile = "./store",             const std::string &cartFile = "./user/carts",             const std::string &orderDir = "./order/orders");      // 析构函数      ~Server();      // 服务器初始化和启动      bool initialize();      void start();      void stop();      // 获取服务器状态      bool isRunning() const;  };  #endif // SERVER\_H |

#### Server.cpp

|  |
| --- |
| #include "server.h"  // 构造函数  Server::Server(const std::string &userFile, const std::string &storeFile,                 const std::string &cartFile, const std::string &orderDir)      : serverSocket(INVALID\_SOCKET),        serverRunning(true),        USER\_FILE(userFile),        STORE\_FILE(storeFile),        CART\_FILE(cartFile),        ORDER\_DIR(orderDir),        store(storeFile),        orderManager(orderDir)  {  }  // 析构函数  Server::~Server()  {      stop();      // 释放资源      for (auto \*user : users)      {          delete user;      }      if (serverSocket != INVALID\_SOCKET)      {          closesocket(serverSocket);      }      WSACleanup();  }  // 初始化服务器  bool Server::initialize()  {      // 设置控制台输出编码为 UTF-8      SetConsoleOutputCP(65001);      SetConsoleCP(65001);      // 初始化 Winsock      WSADATA wsaData;      int result = WSAStartup(MAKEWORD(2, 2), &wsaData);      if (result != 0)      {          std::cerr << "WSAStartup 失败: " << result << std::endl;          return false;      }      // 创建服务器socket      serverSocket = socket(AF\_INET, SOCK\_STREAM, IPPROTO\_TCP);      if (serverSocket == INVALID\_SOCKET)      {          std::cerr << "创建socket失败: " << WSAGetLastError() << std::endl;          WSACleanup();          return false;      }      // 绑定到本地地址和端口      sockaddr\_in serverAddr;      serverAddr.sin\_family = AF\_INET;      serverAddr.sin\_addr.s\_addr = INADDR\_ANY;      serverAddr.sin\_port = htons(8888); // 使用8888端口      if (bind(serverSocket, (sockaddr \*)&serverAddr, sizeof(serverAddr)) == SOCKET\_ERROR)      {          std::cerr << "绑定失败: " << WSAGetLastError() << std::endl;          closesocket(serverSocket);          WSACleanup();          return false;      }      // 开始监听      if (listen(serverSocket, SOMAXCONN) == SOCKET\_ERROR)      {          std::cerr << "监听失败: " << WSAGetLastError() << std::endl;          closesocket(serverSocket);          WSACleanup();          return false;      }      std::cout << "服务器初始化完成。" << std::endl;      // 加载用户数据      users = User::loadUsersFromFile(USER\_FILE);      std::cout << "已加载 " << users.size() << " 个用户数据。" << std::endl;      // 加载商品数据      store.loadAllProducts();      std::cout << "已加载商品数据。" << std::endl;      return true;  }  // 启动服务器  void Server::start()  {      std::cout << "服务器启动，等待连接..." << std::endl;      // 启动订单处理线程      orderManager.startProcessingThread(store, users);      std::cout << "订单处理线程已启动。" << std::endl;      // 主循环，接受客户端连接      while (serverRunning)      {          SOCKET clientSocket = accept(serverSocket, NULL, NULL);          if (clientSocket == INVALID\_SOCKET)          {              std::cerr << "接受连接失败: " << WSAGetLastError() << std::endl;              continue;          }          // 为每个客户端创建新线程          clientThreads.emplace\_back(&Server::handleClient, this, clientSocket);      }      // 等待所有客户端线程结束      for (auto &thread : clientThreads)      {          if (thread.joinable())          {              thread.join();          }      }      // 停止订单处理线程      orderManager.stopProcessingThread();      // 保存用户数据      User::saveUsersToFile(users, USER\_FILE);      std::cout << "用户数据已保存。" << std::endl;  }  // 停止服务器  void Server::stop()  {      serverRunning = false;      // 如果服务器socket有效，关闭它以中断accept调用      if (serverSocket != INVALID\_SOCKET)      {          closesocket(serverSocket);          serverSocket = INVALID\_SOCKET;      }  }  // 获取服务器状态  bool Server::isRunning() const  {      return serverRunning;  }  // 处理客户端请求的函数  void Server::handleClient(SOCKET clientSocket)  {      char buffer[4096] = {0};      std::string response;      bool clientConnected = true;      std::string currentConnectedUser = ""; // 跟踪当前连接的用户名      std::cout << "客户端已连接，处理中..." << std::endl;      while (clientConnected && serverRunning)      {          // 清空缓冲区          memset(buffer, 0, sizeof(buffer));          // 接收客户端请求          int bytesReceived = recv(clientSocket, buffer, sizeof(buffer), 0);          if (bytesReceived <= 0)          {              std::cout << "客户端断开连接或接收错误" << std::endl;              break;          }          // 解析请求          std::string request(buffer);          std::cout << "收到请求: " << request << std::endl;          // 处理请求的核心逻辑          std::string command = request.substr(0, request.find("|"));          response = "ERROR|未知命令";          if (command == "LOGIN")          {              response = processLoginRequest(request);              // 检查是否登录成功，如果是则更新当前连接用户              if (response.substr(0, 13) == "LOGIN\_SUCCESS")              {                  std::istringstream iss(request);                  std::string cmd, username;                  std::getline(iss, cmd, '|');                  std::getline(iss, username, '|');                  currentConnectedUser = username;              }          }          else if (command == "REGISTER")          {              response = processRegisterRequest(request);          }          else if (command == "GET\_PRODUCTS")          {              response = processGetProductsRequest(request);          }          else if (command == "GET\_PRODUCT\_DETAIL")          {              response = processGetProductDetailRequest(request);          }          else if (command == "ADD\_TO\_CART")          {              response = processAddToCartRequest(request);          }          else if (command == "GET\_CART")          {              response = processGetCartRequest(request);          }          else if (command == "CHECKOUT")          {              response = processCheckoutRequest(request);          }          else if (command == "GET\_SELLER\_INFO")          {              response = processGetSellerInfoRequest(request);          }          else if (command == "CHANGE\_PASSWORD")          {              response = processChangePasswordRequest(request);          }          else if (command == "GET\_SELLER\_PRODUCTS")          {              response = processGetSellerProductsRequest(request);          }          else if (command == "ADD\_BOOK")          {              response = processAddBookRequest(request);          }          else if (command == "ADD\_CLOTHING")          {              response = processAddClothingRequest(request);          }          else if (command == "ADD\_FOOD")          {              response = processAddFoodRequest(request);          }          else if (command == "ADD\_GENERIC")          {              response = processAddGenericRequest(request);          }          else if (command == "UPDATE\_PRODUCT\_PRICE")          {              response = processUpdateProductPriceRequest(request);          }          else if (command == "UPDATE\_PRODUCT\_QUANTITY")          {              response = processUpdateProductQuantityRequest(request);          }          else if (command == "SET\_PRODUCT\_DISCOUNT")          {              response = processSetProductDiscountRequest(request);          }          else if (command == "SET\_CATEGORY\_DISCOUNT")          {              response = processSetCategoryDiscountRequest(request);          }          else if (command == "LOGOUT")          {              response = processLogoutRequest(request);              // 检查是否登出成功，如果是则清除当前连接用户              if (response.substr(0, 14) == "LOGOUT\_SUCCESS" &&                  request.find(currentConnectedUser) != std::string::npos)              {                  currentConnectedUser = "";              }          }          else if (command == "CHECK\_BALANCE")          {              response = processCheckBalanceRequest(request);          }          else if (command == "GET\_USER\_INFO")          {              response = processGetUserInfoRequest(request);          }          else if (command == "DEPOSIT")          {              response = processDepositRequest(request);          }          else if (command == "SEARCH\_PRODUCTS")          {              response = processSearchProductsRequest(request);          }          else if (command == "DIRECT\_PURCHASE")          {              response = processDirectPurchaseRequest(request);          }          else if (command == "UPDATE\_CART\_ITEM")          {              response = processUpdateCartItemRequest(request);          }          else if (command == "REMOVE\_FROM\_CART")          {              response = processRemoveFromCartRequest(request);          }          std::cout << "处理结果: " << response << std::endl;          // 发送响应给客户端          send(clientSocket, response.c\_str(), response.size(), 0);      }      // 检查此连接是否有关联的用户，如果有则从已登录用户集合中移除      if (!currentConnectedUser.empty())      {          loggedInUsersMutex.lock();          auto it = loggedInUsers.find(currentConnectedUser);          if (it != loggedInUsers.end())          {              loggedInUsers.erase(it);              std::cout << "用户 " << currentConnectedUser << " 已因连接断开而被自动登出" << std::endl;          }          loggedInUsersMutex.unlock();      }      // 客户端断开连接，关闭socket      closesocket(clientSocket);      std::cout << "客户端连接已关闭" << std::endl;  }  // 处理登录请求  std::string Server::processLoginRequest(const std::string &request)  {      std::istringstream iss(request);      std::string cmd, username, password;      std::getline(iss, cmd, '|');      std::getline(iss, username, '|');      std::getline(iss, password, '|');      // 检查用户是否已经登录      loggedInUsersMutex.lock();      bool alreadyLoggedIn = (loggedInUsers.find(username) != loggedInUsers.end());      loggedInUsersMutex.unlock();      if (alreadyLoggedIn)      {          return "LOGIN\_FAILED|该用户已在其他客户端登录";      }      else      {          usersMutex.lock();          User \*user = User::login(users, username, password);          usersMutex.unlock();          if (user)          {              // 添加到已登录用户集合              loggedInUsersMutex.lock();              loggedInUsers.insert(username);              loggedInUsersMutex.unlock();              return "LOGIN\_SUCCESS|" + user->getUsername() + "|" + user->getUserType();          }          else          {              return "LOGIN\_FAILED|用户名或密码错误";          }      }  }  // 处理注册请求  std::string Server::processRegisterRequest(const std::string &request)  {      std::istringstream iss(request);      std::string cmd, username, password, type;      std::getline(iss, cmd, '|');      std::getline(iss, username, '|');      std::getline(iss, password, '|');      std::getline(iss, type, '|');      usersMutex.lock();      bool exists = User::isUsernameExists(users, username);      if (!exists)      {          User \*newUser = nullptr;          if (type == "customer")          {              newUser = new Customer(username, password, CART\_FILE, 0.0);          }          else if (type == "seller")          {              newUser = new Seller(username, password, 0.0);          }          if (newUser)          {              users.push\_back(newUser);              User::saveUsersToFile(users, USER\_FILE);              usersMutex.unlock();              return "REGISTER\_SUCCESS|" + username;          }          else          {              usersMutex.unlock();              return "REGISTER\_FAILED|创建用户失败";          }      }      else      {          usersMutex.unlock();          return "REGISTER\_FAILED|用户名已存在";      }  }  // 处理获取商品列表请求  std::string Server::processGetProductsRequest(const std::string &request) const  {      storeMutex.lock();      const auto &products = store.getProducts();      std::ostringstream oss;      oss << "PRODUCTS|" << products.size();      for (const auto &product : products)      {          oss << "|" << product->getName()              << "," << product->getType()              << "," << product->getoriginalPrice()              << "," << product->getPrice()              << "," << product->getQuantity()              << "," << product->getSellerUsername();      }      storeMutex.unlock();      return oss.str();  }  // 处理获取商品详情请求  std::string Server::processGetProductDetailRequest(const std::string &request) const  {      std::istringstream iss(request);      std::string cmd, productName, sellerUsername;      std::getline(iss, cmd, '|');      std::getline(iss, productName, '|');      std::getline(iss, sellerUsername, '|');      storeMutex.lock();      Product \*product = store.findProductByName(productName, sellerUsername);      if (product)      {          std::ostringstream oss;          oss << "PRODUCT\_DETAIL|" << product->getName()              << "|" << product->getType()              << "|" << product->getDescription()              << "|" << product->getPrice()              << "|" << product->getQuantity()              << "|" << product->getSellerUsername();          // 根据不同类型添加特定属性          if (product->getType() == "Book")          {              Book \*book = dynamic\_cast<Book \*>(product);              oss << "|" << book->getAuthor() << "|" << book->getIsbn();          }          else if (product->getType() == "Clothing")          {              Clothing \*clothing = dynamic\_cast<Clothing \*>(product);              oss << "|" << clothing->getSize() << "|" << clothing->getColor();          }          else if (product->getType() == "Food")          {              Food \*food = dynamic\_cast<Food \*>(product);              oss << "|" << food->getExpirationDate();          }          storeMutex.unlock();          return oss.str();      }      else      {          storeMutex.unlock();          return "PRODUCT\_NOT\_FOUND|产品不存在";      }  }  // 处理添加到购物车请求  std::string Server::processAddToCartRequest(const std::string &request)  {      std::istringstream iss(request);      std::string cmd, username, productName, sellerUsername;      int quantity;      std::getline(iss, cmd, '|');      std::getline(iss, username, '|');      std::getline(iss, productName, '|');      std::getline(iss, sellerUsername, '|');      iss >> quantity;      usersMutex.lock();      storeMutex.lock();      User \*user = User::findUser(users, username);      Customer \*customer = dynamic\_cast<Customer \*>(user);      Product \*product = store.findProductByName(productName, sellerUsername);      if (customer && product)      {          if (product->getQuantity() < quantity)          {              storeMutex.unlock();              usersMutex.unlock();              return "CART\_ERROR|库存不足，当前库存: " + std::to\_string(product->getQuantity());          }          else          {              // 直接操作 Customer 的 shoppingCartItems              bool found = false;              customer->loadCartFromFile(); // 确保购物车是最新的              for (auto &item : customer->shoppingCartItems)              {                  if (item.productName == productName && item.sellerUsername == sellerUsername)                  {                      item.quantity += quantity;                      found = true;                      break;                  }              }              if (!found)              {                  // 添加新商品到购物车                  CartItem newItem;                  newItem.productId = product->getName(); // 为了兼容性，保留 productId                  newItem.productName = product->getName();                  newItem.sellerUsername = sellerUsername;                  newItem.quantity = quantity;                  newItem.priceAtAddition = product->getPrice();                  customer->shoppingCartItems.push\_back(newItem);              }              customer->saveCartToFile();              storeMutex.unlock();              usersMutex.unlock();              return "CART\_UPDATED|添加成功";          }      }      else      {          storeMutex.unlock();          usersMutex.unlock();          return "CART\_ERROR|用户或产品不存在";      }  }  // 其他方法的实现将在后续补充...  // 处理获取购物车请求  std::string Server::processGetCartRequest(const std::string &request)  {      std::istringstream iss(request);      std::string cmd, username;      std::getline(iss, cmd, '|');      std::getline(iss, username, '|');      usersMutex.lock();      User \*user = User::findUser(users, username);      Customer \*customer = dynamic\_cast<Customer \*>(user);      if (customer)      {          customer->loadCartFromFile();          std::ostringstream oss;          oss << "CART|" << customer->shoppingCartItems.size();          for (const auto &item : customer->shoppingCartItems)          {              oss << "|" << item.productName                  << "," << item.quantity                  << "," << item.priceAtAddition                  << "," << item.sellerUsername;          }          usersMutex.unlock();          return oss.str();      }      else      {          usersMutex.unlock();          return "CART\_ERROR|用户不存在或不是消费者";      }  }  // 处理结算购物车请求  std::string Server::processCheckoutRequest(const std::string &request)  {      std::istringstream iss(request);      std::string cmd, username;      std::getline(iss, cmd, '|');      std::getline(iss, username, '|');      usersMutex.lock();      storeMutex.lock();      User \*user = User::findUser(users, username);      Customer \*customer = dynamic\_cast<Customer \*>(user);      if (customer)      {          customer->loadCartFromFile();          if (customer->isCartEmpty())          {              storeMutex.unlock();              usersMutex.unlock();              return "CHECKOUT\_ERROR|购物车为空";          }          else          {              Order order(username);              double totalAmount = 0.0;              bool allItemsValid = true;              for (const auto &item : customer->shoppingCartItems)              {                  Product \*product = store.findProductByName(item.productName, item.sellerUsername);                  if (product && product->getQuantity() >= item.quantity)                  {                      order.addItemFromProduct(\*product, item.quantity);                      totalAmount += item.quantity \* product->getPrice();                  }                  else                  {                      allItemsValid = false;                      break;                  }              }              if (allItemsValid && customer->checkBalance() >= totalAmount)              {                  customer->withdraw(totalAmount);                  order.calculateTotalAmount();                  auto orderPtr = orderManager.submitOrderRequest(order);                  // 等待订单处理完成                  auto startTime = std::chrono::steady\_clock::now();                  bool orderProcessed = false;                  // 最多等待10秒                  while (!orderProcessed &&                         std::chrono::duration\_cast<std::chrono::seconds>(                             std::chrono::steady\_clock::now() - startTime)                                 .count() < 10)                  {                      if (orderPtr->getProcessed())                      {                          orderProcessed = true;                          break;                      }                      // 处理订单并更新库存（如果有订单在队列中）                      orderManager.processNextOrder(store, users);                      // 短暂暂停，避免CPU过度使用                      std::this\_thread::sleep\_for(std::chrono::milliseconds(100));                  }                  if (orderProcessed)                  {                      if (orderPtr->getStatus() == "COMPLETED" ||                          orderPtr->getStatus() == "COMPLETED\_WITH\_PAYMENT\_ISSUES")                      {                          // 清空购物车                          customer->clearCartAndFile();                          // 保存用户数据                          User::saveUsersToFile(users, USER\_FILE);                          storeMutex.unlock();                          usersMutex.unlock();                          return "CHECKOUT\_SUCCESS|订单已提交|" + orderPtr->getOrderId();                      }                      else                      {                          // 订单处理失败，返回错误信息                          storeMutex.unlock();                          usersMutex.unlock();                          return "CHECKOUT\_ERROR|订单处理失败: " + orderPtr->getStatus();                      }                  }                  else                  {                      // 订单处理超时，返回等待信息                      storeMutex.unlock();                      usersMutex.unlock();                      return "CHECKOUT\_PENDING|订单正在处理中，请稍后查询|" + orderPtr->getOrderId();                  }              }              else if (!allItemsValid)              {                  storeMutex.unlock();                  usersMutex.unlock();                  return "CHECKOUT\_ERROR|部分商品库存不足";              }              else              {                  storeMutex.unlock();                  usersMutex.unlock();                  return "CHECKOUT\_ERROR|余额不足";              }          }      }      else      {          storeMutex.unlock();          usersMutex.unlock();          return "CHECKOUT\_ERROR|用户不存在或不是消费者";      }  }  // 处理获取商家信息请求  std::string Server::processGetSellerInfoRequest(const std::string &request) const  {      std::istringstream iss(request);      std::string cmd, username;      std::getline(iss, cmd, '|');      std::getline(iss, username, '|');      usersMutex.lock();      User \*user = User::findUser(users, username);      if (user && user->getUserType() == "seller")      {          std::ostringstream oss;          oss << "SELLER\_INFO|" << user->getUsername() << "|"              << user->getUserType() << "|" << user->checkBalance();          usersMutex.unlock();          return oss.str();      }      else      {          usersMutex.unlock();          return "SELLER\_ERROR|用户不存在或不是商家";      }  }  // 处理修改密码请求  std::string Server::processChangePasswordRequest(const std::string &request)  {      std::istringstream iss(request);      std::string cmd, username, oldPassword, newPassword;      std::getline(iss, cmd, '|');      std::getline(iss, username, '|');      std::getline(iss, oldPassword, '|');      std::getline(iss, newPassword, '|');      usersMutex.lock();      User \*user = User::findUser(users, username);      if (user)      {          if (user->changePassword(oldPassword, newPassword))          {              User::saveUsersToFile(users, USER\_FILE);              usersMutex.unlock();              return "PASSWORD\_CHANGED|密码修改成功";          }          else          {              usersMutex.unlock();              return "PASSWORD\_ERROR|密码修改失败，可能原密码错误";          }      }      else      {          usersMutex.unlock();          return "PASSWORD\_ERROR|用户不存在";      }  }  // 处理获取商家商品请求  std::string Server::processGetSellerProductsRequest(const std::string &request) const  {      std::istringstream iss(request);      std::string cmd, username;      std::getline(iss, cmd, '|');      std::getline(iss, username, '|');      storeMutex.lock();      std::vector<Product \*> sellerProducts = store.getSellerProducts(username);      std::ostringstream oss;      oss << "SELLER\_PRODUCTS|" << sellerProducts.size();      for (const auto &product : sellerProducts)      {          oss << "|" << product->getName()              << "," << product->getType()              << "," << product->getOriginalPrice()              << "," << product->getQuantity()              << "," << product->getDiscountRate();      }      storeMutex.unlock();      return oss.str();  }  // 处理添加书籍请求  std::string Server::processAddBookRequest(const std::string &request)  {      std::istringstream iss(request);      std::string cmd, username, name, description, priceStr, quantityStr, author, isbn;      std::getline(iss, cmd, '|');      std::getline(iss, username, '|');      std::getline(iss, name, '|');      std::getline(iss, description, '|');      std::getline(iss, priceStr, '|');      std::getline(iss, quantityStr, '|');      std::getline(iss, author, '|');      std::getline(iss, isbn, '|');      double price = std::stod(priceStr);      int quantity = std::stoi(quantityStr);      usersMutex.lock();      storeMutex.lock();      User \*user = User::findUser(users, username);      if (user && user->getUserType() == "seller")      {          if (store.createBook(user, name, description, price, quantity, author, isbn))          {              storeMutex.unlock();              usersMutex.unlock();              return "PRODUCT\_ADDED|书籍添加成功";          }          else          {              storeMutex.unlock();              usersMutex.unlock();              return "PRODUCT\_ERROR|书籍添加失败，可能是名称重复";          }      }      else      {          storeMutex.unlock();          usersMutex.unlock();          return "PRODUCT\_ERROR|用户不存在或不是商家";      }  }  // 处理添加服装请求  std::string Server::processAddClothingRequest(const std::string &request)  {      std::istringstream iss(request);      std::string cmd, username, name, description, priceStr, quantityStr, size, color;      std::getline(iss, cmd, '|');      std::getline(iss, username, '|');      std::getline(iss, name, '|');      std::getline(iss, description, '|');      std::getline(iss, priceStr, '|');      std::getline(iss, quantityStr, '|');      std::getline(iss, size, '|');      std::getline(iss, color, '|');      double price = std::stod(priceStr);      int quantity = std::stoi(quantityStr);      usersMutex.lock();      storeMutex.lock();      User \*user = User::findUser(users, username);      if (user && user->getUserType() == "seller")      {          if (store.createClothing(user, name, description, price, quantity, size, color))          {              storeMutex.unlock();              usersMutex.unlock();              return "PRODUCT\_ADDED|服装添加成功";          }          else          {              storeMutex.unlock();              usersMutex.unlock();              return "PRODUCT\_ERROR|服装添加失败，可能是名称重复";          }      }      else      {          storeMutex.unlock();          usersMutex.unlock();          return "PRODUCT\_ERROR|用户不存在或不是商家";      }  }  // 处理添加食品请求  std::string Server::processAddFoodRequest(const std::string &request)  {      std::istringstream iss(request);      std::string cmd, username, name, description, priceStr, quantityStr, expDate;      std::getline(iss, cmd, '|');      std::getline(iss, username, '|');      std::getline(iss, name, '|');      std::getline(iss, description, '|');      std::getline(iss, priceStr, '|');      std::getline(iss, quantityStr, '|');      std::getline(iss, expDate, '|');      double price = std::stod(priceStr);      int quantity = std::stoi(quantityStr);      usersMutex.lock();      storeMutex.lock();      User \*user = User::findUser(users, username);      if (user && user->getUserType() == "seller")      {          if (store.createFood(user, name, description, price, quantity, expDate))          {              storeMutex.unlock();              usersMutex.unlock();              return "PRODUCT\_ADDED|食品添加成功";          }          else          {              storeMutex.unlock();              usersMutex.unlock();              return "PRODUCT\_ERROR|食品添加失败，可能是名称重复";          }      }      else      {          storeMutex.unlock();          usersMutex.unlock();          return "PRODUCT\_ERROR|用户不存在或不是商家";      }  }  // 处理添加通用商品请求  std::string Server::processAddGenericRequest(const std::string &request)  {      std::istringstream iss(request);      std::string cmd, username, name, description, priceStr, quantityStr, categoryTag;      std::getline(iss, cmd, '|');      std::getline(iss, username, '|');      std::getline(iss, name, '|');      std::getline(iss, description, '|');      std::getline(iss, priceStr, '|');      std::getline(iss, quantityStr, '|');      std::getline(iss, categoryTag, '|');      double price = std::stod(priceStr);      int quantity = std::stoi(quantityStr);      usersMutex.lock();      storeMutex.lock();      User \*user = User::findUser(users, username);      if (user && user->getUserType() == "seller")      {          if (store.createGenericProduct(user, name, description, price, quantity, categoryTag))          {              storeMutex.unlock();              usersMutex.unlock();              return "PRODUCT\_ADDED|通用商品添加成功";          }          else          {              storeMutex.unlock();              usersMutex.unlock();              return "PRODUCT\_ERROR|通用商品添加失败，可能是名称重复";          }      }      else      {          storeMutex.unlock();          usersMutex.unlock();          return "PRODUCT\_ERROR|用户不存在或不是商家";      }  }  // 处理更新商品价格请求  std::string Server::processUpdateProductPriceRequest(const std::string &request)  {      std::istringstream iss(request);      std::string cmd, username, productName, priceStr;      std::getline(iss, cmd, '|');      std::getline(iss, username, '|');      std::getline(iss, productName, '|');      std::getline(iss, priceStr, '|');      double newPrice = std::stod(priceStr);      usersMutex.lock();      storeMutex.lock();      User \*user = User::findUser(users, username);      if (user && user->getUserType() == "seller")      {          if (store.manageProductPrice(user, productName, newPrice))          {              storeMutex.unlock();              usersMutex.unlock();              return "PRODUCT\_UPDATED|价格修改成功";          }          else          {              storeMutex.unlock();              usersMutex.unlock();              return "PRODUCT\_ERROR|价格修改失败，可能商品不存在或不属于该商家";          }      }      else      {          storeMutex.unlock();          usersMutex.unlock();          return "PRODUCT\_ERROR|用户不存在或不是商家";      }  }  // 处理更新商品库存请求  std::string Server::processUpdateProductQuantityRequest(const std::string &request)  {      std::istringstream iss(request);      std::string cmd, username, productName, quantityStr;      std::getline(iss, cmd, '|');      std::getline(iss, username, '|');      std::getline(iss, productName, '|');      std::getline(iss, quantityStr, '|');      int newQuantity = std::stoi(quantityStr);      usersMutex.lock();      storeMutex.lock();      User \*user = User::findUser(users, username);      if (user && user->getUserType() == "seller")      {          if (store.manageProductQuantity(user, productName, newQuantity))          {              storeMutex.unlock();              usersMutex.unlock();              return "PRODUCT\_UPDATED|库存修改成功";          }          else          {              storeMutex.unlock();              usersMutex.unlock();              return "PRODUCT\_ERROR|库存修改失败，可能商品不存在或不属于该商家";          }      }      else      {          storeMutex.unlock();          usersMutex.unlock();          return "PRODUCT\_ERROR|用户不存在或不是商家";      }  }  // 处理设置商品折扣请求  std::string Server::processSetProductDiscountRequest(const std::string &request)  {      std::istringstream iss(request);      std::string cmd, username, productName, discountStr;      std::getline(iss, cmd, '|');      std::getline(iss, username, '|');      std::getline(iss, productName, '|');      std::getline(iss, discountStr, '|');      double discount = std::stod(discountStr);      usersMutex.lock();      storeMutex.lock();      User \*user = User::findUser(users, username);      if (user && user->getUserType() == "seller")      {          if (store.manageProductDiscount(user, productName, discount))          {              storeMutex.unlock();              usersMutex.unlock();              return "PRODUCT\_UPDATED|折扣设置成功";          }          else          {              storeMutex.unlock();              usersMutex.unlock();              return "PRODUCT\_ERROR|折扣设置失败，可能商品不存在或不属于该商家";          }      }      else      {          storeMutex.unlock();          usersMutex.unlock();          return "PRODUCT\_ERROR|用户不存在或不是商家";      }  }  // 处理设置类别折扣请求  std::string Server::processSetCategoryDiscountRequest(const std::string &request)  {      std::istringstream iss(request);      std::string cmd, username, category, discountStr;      std::getline(iss, cmd, '|');      std::getline(iss, username, '|');      std::getline(iss, category, '|');      std::getline(iss, discountStr, '|');      double discount = std::stod(discountStr);      usersMutex.lock();      storeMutex.lock();      User \*user = User::findUser(users, username);      if (user && user->getUserType() == "seller")      {          if (store.applyCategoryDiscount(user, category, discount))          {              storeMutex.unlock();              usersMutex.unlock();              return "PRODUCT\_UPDATED|类别折扣设置成功";          }          else          {              storeMutex.unlock();              usersMutex.unlock();              return "PRODUCT\_ERROR|类别折扣设置失败，可能该类别不存在或没有商品";          }      }      else      {          storeMutex.unlock();          usersMutex.unlock();          return "PRODUCT\_ERROR|用户不存在或不是商家";      }  }  // 处理登出请求  std::string Server::processLogoutRequest(const std::string &request)  {      std::istringstream iss(request);      std::string cmd, username;      std::getline(iss, cmd, '|');      std::getline(iss, username, '|');      loggedInUsersMutex.lock();      auto it = loggedInUsers.find(username);      if (it != loggedInUsers.end())      {          loggedInUsers.erase(it);          loggedInUsersMutex.unlock();          return "LOGOUT\_SUCCESS|用户已成功登出";      }      else      {          loggedInUsersMutex.unlock();          return "LOGOUT\_FAILED|用户未登录";      }  }  // 处理检查余额请求  std::string Server::processCheckBalanceRequest(const std::string &request) const  {      std::istringstream iss(request);      std::string cmd, username;      std::getline(iss, cmd, '|');      std::getline(iss, username, '|');      usersMutex.lock();      User \*user = User::findUser(users, username);      if (user)      {          double balance = user->checkBalance();          usersMutex.unlock();          return "BALANCE|" + std::to\_string(balance);      }      else      {          usersMutex.unlock();          return "BALANCE\_ERROR|用户不存在";      }  }  // 处理获取用户信息请求  std::string Server::processGetUserInfoRequest(const std::string &request) const  {      std::istringstream iss(request);      std::string cmd, username;      std::getline(iss, cmd, '|');      std::getline(iss, username, '|');      usersMutex.lock();      User \*user = User::findUser(users, username);      if (user)      {          std::ostringstream oss;          oss << "USER\_INFO|" << user->getUsername() << "|"              << user->getUserType() << "|" << user->checkBalance();          usersMutex.unlock();          return oss.str();      }      else      {          usersMutex.unlock();          return "USER\_ERROR|用户不存在";      }  }  // 处理充值请求  std::string Server::processDepositRequest(const std::string &request)  {      std::istringstream iss(request);      std::string cmd, username, amountStr;      std::getline(iss, cmd, '|');      std::getline(iss, username, '|');      std::getline(iss, amountStr, '|');      double amount = std::stod(amountStr);      usersMutex.lock();      User \*user = User::findUser(users, username);      if (user)      {          if (amount <= 0)          {              usersMutex.unlock();              return "BALANCE\_ERROR|充值金额必须大于0";          }          if (user->deposit(amount))          {              User::saveUsersToFile(users, USER\_FILE);              double newBalance = user->checkBalance();              usersMutex.unlock();              return "BALANCE\_UPDATED|充值成功|" + std::to\_string(newBalance);          }          else          {              usersMutex.unlock();              return "BALANCE\_ERROR|充值失败";          }      }      else      {          usersMutex.unlock();          return "BALANCE\_ERROR|用户不存在";      }  }  // 处理搜索商品请求  std::string Server::processSearchProductsRequest(const std::string &request) const  {      std::istringstream iss(request);      std::string cmd, searchTerm, sellerUsername;      std::getline(iss, cmd, '|');      std::getline(iss, searchTerm, '|');      std::getline(iss, sellerUsername, '|');      storeMutex.lock();      std::vector<Product \*> searchResults = store.searchProductsByName(searchTerm, sellerUsername);      std::ostringstream oss;      oss << "SEARCH\_RESULTS|" << searchResults.size();      for (const auto &product : searchResults)      {          oss << "|" << product->getName()              << "," << product->getType()              << "," << product->getPrice()              << "," << product->getQuantity()              << "," << product->getSellerUsername();      }      storeMutex.unlock();      return oss.str();  }  // 处理直接购买请求  std::string Server::processDirectPurchaseRequest(const std::string &request)  {      std::istringstream iss(request);      std::string cmd, username, productName, sellerUsername, quantityStr;      std::getline(iss, cmd, '|');      std::getline(iss, username, '|');      std::getline(iss, productName, '|');      std::getline(iss, sellerUsername, '|');      std::getline(iss, quantityStr, '|');      int quantity = std::stoi(quantityStr);      usersMutex.lock();      storeMutex.lock();      User \*user = User::findUser(users, username);      Customer \*customer = dynamic\_cast<Customer \*>(user);      Product \*product = store.findProductByName(productName, sellerUsername);      if (customer && product)      {          if (product->getQuantity() < quantity)          {              storeMutex.unlock();              usersMutex.unlock();              return "PURCHASE\_ERROR|库存不足，当前库存: " + std::to\_string(product->getQuantity());          }          double totalCost = product->getPrice() \* quantity;          if (customer->checkBalance() < totalCost)          {              storeMutex.unlock();              usersMutex.unlock();              return "PURCHASE\_ERROR|余额不足，需要: " + std::to\_string(totalCost) + ", 当前余额: " + std::to\_string(customer->checkBalance());          }          // 创建订单          Order order(username);          order.addItemFromProduct(\*product, quantity);          order.calculateTotalAmount();          // 扣款          customer->withdraw(totalCost);          // 提交订单          auto orderPtr = orderManager.submitOrderRequest(order);          // 等待订单处理完成          auto startTime = std::chrono::steady\_clock::now();          bool orderProcessed = false;          // 最多等待10秒          while (!orderProcessed &&                 std::chrono::duration\_cast<std::chrono::seconds>(                     std::chrono::steady\_clock::now() - startTime)                         .count() < 10)          {              if (orderPtr->getProcessed())              {                  orderProcessed = true;                  break;              }              // 处理订单并更新库存              orderManager.processNextOrder(store, users);              // 短暂暂停，避免CPU过度使用              std::this\_thread::sleep\_for(std::chrono::milliseconds(100));          }          if (orderProcessed)          {              if (orderPtr->getStatus() == "COMPLETED" ||                  orderPtr->getStatus() == "COMPLETED\_WITH\_PAYMENT\_ISSUES")              {                  // 保存用户数据                  User::saveUsersToFile(users, USER\_FILE);                  storeMutex.unlock();                  usersMutex.unlock();                  return "PURCHASE\_SUCCESS|购买成功|" + orderPtr->getOrderId();              }              else              {                  // 订单处理失败，返回错误信息                  storeMutex.unlock();                  usersMutex.unlock();                  return "PURCHASE\_ERROR|订单处理失败: " + orderPtr->getStatus();              }          }          else          {              // 订单处理超时，返回等待信息              storeMutex.unlock();              usersMutex.unlock();              return "PURCHASE\_PENDING|订单正在处理中，请稍后查询|" + orderPtr->getOrderId();          }      }      else      {          storeMutex.unlock();          usersMutex.unlock();          return "PURCHASE\_ERROR|用户或商品不存在";      }  }  // 处理更新购物车商品数量请求  std::string Server::processUpdateCartItemRequest(const std::string &request)  {      std::istringstream iss(request);      std::string cmd, username, productName, sellerUsername, quantityStr;      std::getline(iss, cmd, '|');      std::getline(iss, username, '|');      std::getline(iss, productName, '|');      std::getline(iss, sellerUsername, '|');      std::getline(iss, quantityStr, '|');      int newQuantity = std::stoi(quantityStr);      usersMutex.lock();      storeMutex.lock();      User \*user = User::findUser(users, username);      Customer \*customer = dynamic\_cast<Customer \*>(user);      if (customer)      {          customer->loadCartFromFile();          bool updated = false;          for (auto &item : customer->shoppingCartItems)          {              if (item.productName == productName && item.sellerUsername == sellerUsername)              {                  // 检查新数量是否有效                  Product \*product = store.findProductByName(productName, sellerUsername);                  if (product && product->getQuantity() >= newQuantity)                  {                      item.quantity = newQuantity;                      updated = true;                  }                  else if (product)                  {                      storeMutex.unlock();                      usersMutex.unlock();                      return "CART\_ERROR|库存不足，当前库存: " + std::to\_string(product->getQuantity());                  }                  break;              }          }          if (updated)          {              customer->saveCartToFile();              storeMutex.unlock();              usersMutex.unlock();              return "CART\_UPDATED|商品数量已更新";          }          else          {              storeMutex.unlock();              usersMutex.unlock();              return "CART\_ERROR|购物车中未找到该商品";          }      }      else      {          storeMutex.unlock();          usersMutex.unlock();          return "CART\_ERROR|用户不存在或不是消费者";      }  }  // 处理从购物车移除商品请求  std::string Server::processRemoveFromCartRequest(const std::string &request)  {      std::istringstream iss(request);      std::string cmd, username, productName, sellerUsername;      std::getline(iss, cmd, '|');      std::getline(iss, username, '|');      std::getline(iss, productName, '|');      std::getline(iss, sellerUsername, '|');      usersMutex.lock();      User \*user = User::findUser(users, username);      Customer \*customer = dynamic\_cast<Customer \*>(user);      if (customer)      {          bool removed = false;          customer->loadCartFromFile();          auto it = std::remove\_if(customer->shoppingCartItems.begin(), customer->shoppingCartItems.end(),                                   [&productName, &sellerUsername](const CartItem &item)                                   {                                       return (item.productName == productName && item.sellerUsername == sellerUsername);                                   });          if (it != customer->shoppingCartItems.end())          {              customer->shoppingCartItems.erase(it, customer->shoppingCartItems.end());              customer->saveCartToFile();              usersMutex.unlock();              return "CART\_UPDATED|商品已从购物车移除";              removed = true;          }          if (!removed)          {              usersMutex.unlock();              return "CART\_ERROR|购物车中未找到该商品";          }      }      else      {          usersMutex.unlock();          return "CART\_ERROR|用户不存在或不是消费者";      }      usersMutex.unlock();      return "CART\_ERROR|未知错误";  } |

#### Client.h

|  |
| --- |
| #ifndef CLIENT\_H  #define CLIENT\_H  #include <iostream>  #include <winsock2.h>  #include <ws2tcpip.h>  #include <windows.h>  #include <string>  #include <vector>  #include <sstream>  #include <limits>  #include <iomanip>  #pragma comment(lib, "ws2\_32.lib")  class Client  {  private:      SOCKET clientSocket;      std::string currentUsername;      std::string currentUserType;      bool isLoggedIn; // 辅助函数      bool connectToServer();      bool sendRequest(const std::string &request, std::string &response);      void clearInputBuffer() const;      // 菜单显示函数      int showMainMenu() const;      int showCustomerMenu() const;      int showSellerMenu() const;      // 用户操作处理函数      void handleLogin();      void handleRegister();      void handleBrowseProducts();      void handleProductDetail();      void handleViewCart();      void handleAddToCart();      void handleCheckout();      void handleLogout();      void handleViewUserInfo();      void handleCheckBalance();      void handleDeposit();      void handleEnterStore();      void handleSearchProducts();      void handleDirectPurchase();      // 商家相关函数      void handleSellerInfo();      void handleChangePassword();      void handleManageProducts();      void handleCheckIncome();  public:      Client();      ~Client();      bool initialize();      void run();      void cleanup();  };  #endif // CLIENT\_H |

#### Client.cpp

|  |
| --- |
| #include "client.h"  Client::Client() : clientSocket(INVALID\_SOCKET), isLoggedIn(false)  {      // 构造函数，初始化成员变量      // 注意：clientSocket 已在初始化列表中设置为 INVALID\_SOCKET      // isLoggedIn 已在初始化列表中设置为 false      // currentUsername 和 currentUserType 默认为空字符串  }  Client::~Client()  {      // 析构函数，确保资源被释放      cleanup();  }  bool Client::initialize()  {      // 设置控制台编码为 UTF-8      SetConsoleOutputCP(65001);      SetConsoleCP(65001);      // 初始化 Winsock      WSADATA wsaData;      int result = WSAStartup(MAKEWORD(2, 2), &wsaData);      if (result != 0)      {          std::cerr << "WSAStartup 失败: " << result << std::endl;          return false;      }      // 连接到服务器      if (!connectToServer())      {          std::cerr << "无法连接到服务器，请检查服务器是否运行" << std::endl;          WSACleanup();          return false;      }      return true;  }  void Client::run()  {      int menuResult = 1; // 1=继续, 0=退出程序, -1=退出登录      while (menuResult != 0)      {          if (menuResult == -1)          { // 处理退出登录              handleLogout();              menuResult = 1;          }          if (!isLoggedIn)          {              menuResult = showMainMenu();              if (menuResult == 1)              { // 登录                  handleLogin();              }              else if (menuResult == 2)              { // 注册                  handleRegister();              }          }          else          { // 已登录              if (currentUserType == "customer")              {                  menuResult = showCustomerMenu();                  if (menuResult == 1)                  { // 查看个人信息                      handleViewUserInfo();                  }                  else if (menuResult == 2)                  { // 修改密码                      handleChangePassword();                  }                  else if (menuResult == 3)                  { // 查询余额                      handleCheckBalance();                  }                  else if (menuResult == 4)                  { // 充值                      handleDeposit();                  }                  else if (menuResult == 5)                  { // 进入商城                      handleEnterStore();                  }              }              else if (currentUserType == "seller")              {                  menuResult = showSellerMenu();                  if (menuResult == 1)                  { // 查看商户信息                      handleSellerInfo();                  }                  else if (menuResult == 2)                  { // 修改密码                      handleChangePassword();                  }                  else if (menuResult == 3)                  { // 进入商城(管理)                      handleManageProducts();                  }                  else if (menuResult == 4)                  { // 查看收入                      handleCheckIncome();                  }              }          }      }      // 最后确保正确登出      if (isLoggedIn)      {          handleLogout();      }  }  bool Client::connectToServer()  {      // 创建客户端socket      clientSocket = socket(AF\_INET, SOCK\_STREAM, IPPROTO\_TCP);      if (clientSocket == INVALID\_SOCKET)      {          std::cerr << "创建socket失败: " << WSAGetLastError() << std::endl;          return false;      }      // 设置服务器地址      sockaddr\_in serverAddr;      serverAddr.sin\_family = AF\_INET;      // 默认连接到本地服务器，可以根据需要修改      inet\_pton(AF\_INET, "127.0.0.1", &serverAddr.sin\_addr);      serverAddr.sin\_port = htons(8888); // 使用8888端口      // 连接到服务器      if (connect(clientSocket, (sockaddr \*)&serverAddr, sizeof(serverAddr)) == SOCKET\_ERROR)      {          std::cerr << "连接失败: " << WSAGetLastError() << std::endl;          closesocket(clientSocket);          clientSocket = INVALID\_SOCKET;          return false;      }      std::cout << "已连接到服务器。" << std::endl;      return true;  }  bool Client::sendRequest(const std::string &request, std::string &response)  {      // 发送请求      if (send(clientSocket, request.c\_str(), request.size(), 0) == SOCKET\_ERROR)      {          std::cerr << "发送失败: " << WSAGetLastError() << std::endl;          return false;      }      // 接收响应      char buffer[4096] = {0};      int bytesReceived = recv(clientSocket, buffer, sizeof(buffer), 0);      if (bytesReceived <= 0)      {          std::cerr << "接收失败或连接关闭: " << WSAGetLastError() << std::endl;          return false;      }      response = std::string(buffer, bytesReceived);      return true;  }  void Client::clearInputBuffer() const  {      std::cin.clear();                                                   // 清除错误状态      std::cin.ignore(std::numeric\_limits<std::streamsize>::max(), '\n'); // 忽略剩余输入  }  int Client::showMainMenu() const  {      int choice;      std::cout << "\n===== 欢迎使用网络电子商城系统 =====\n";      std::cout << "1. 登录\n";      std::cout << "2. 注册\n";      std::cout << "0. 退出\n";      std::cout << "请选择: ";      std::cin >> choice;      clearInputBuffer();      if (choice < 0 || choice > 2)      {          std::cout << "无效选择，请重试。\n";          return 1; // 默认返回继续      }      return choice;  }  int Client::showCustomerMenu() const  {      int choice;      std::cout << "\n\n\n--- 消费者菜单 (" << currentUsername << ") ---\n";      std::cout << "1. 查看个人信息\n";      std::cout << "2. 修改密码\n";      std::cout << "3. 查询余额\n";      std::cout << "4. 充值\n";      std::cout << "5. 进入商城\n";      std::cout << "6. 退出登录\n";      std::cout << "0. 退出程序\n";      std::cout << "请选择一个选项: ";      std::cin >> choice;      clearInputBuffer();      if (choice == 6)          return -1; // 退出登录      if (choice == 0)          return 0; // 退出程序      if (choice < 0 || choice > 6)      {          std::cout << "无效选择，请重试。\n";          return 1; // 默认返回继续      }      return choice;  }  int Client::showSellerMenu() const  {      int choice;      std::cout << "\n===== 商家菜单 (" << currentUsername << ") =====\n";      std::cout << "1. 查看商户信息\n";      std::cout << "2. 修改密码\n";      std::cout << "3. 进入商城 (管理)\n";      std::cout << "4. 查看收入\n";      std::cout << "5. 退出登录\n";      std::cout << "0. 退出程序\n";      std::cout << "请选择: ";      std::cin >> choice;      clearInputBuffer();      if (choice == 5)          return -1; // 退出登录      if (choice == 0)          return 0; // 退出程序      if (choice < 0 || choice > 5)      {          std::cout << "无效选择，请重试。\n";          return 1; // 默认返回继续      }      return choice;  }  void Client::handleLogin()  {      std::string username, password;      std::cout << "\n===== 用户登录 =====\n";      std::cout << "用户名: ";      std::getline(std::cin, username);      std::cout << "密码: ";      std::getline(std::cin, password);      std::string request = "LOGIN|" + username + "|" + password;      std::string response;      if (sendRequest(request, response))      {          std::istringstream iss(response);          std::string status, message, userType;          std::getline(iss, status, '|');          if (status == "LOGIN\_SUCCESS")          {              std::getline(iss, message, '|');              std::getline(iss, userType, '|');              isLoggedIn = true;              currentUsername = message;              currentUserType = userType;              std::cout << "登录成功！欢迎 " << currentUsername << "(" << currentUserType << ")" << std::endl;          }          else          {              std::getline(iss, message, '|');              std::cout << "登录失败: " << message << std::endl;          }      }      else      {          std::cout << "与服务器通信失败。" << std::endl;      }  }  void Client::handleRegister()  {      std::string username, password, confirmPassword, typeChoice;      int choice;      std::cout << "\n===== 用户注册 =====\n";      std::cout << "用户名: ";      std::getline(std::cin, username);      std::cout << "密码: ";      std::getline(std::cin, password);      std::cout << "确认密码: ";      std::getline(std::cin, confirmPassword);      if (password != confirmPassword)      {          std::cout << "两次输入的密码不一致，请重试。" << std::endl;          return;      }      std::cout << "账户类型 (1: 消费者, 2: 商家): ";      std::cin >> choice;      clearInputBuffer();      if (choice == 1)      {          typeChoice = "customer";      }      else if (choice == 2)      {          typeChoice = "seller";      }      else      {          std::cout << "无效选择，请重试。" << std::endl;          return;      }      std::string request = "REGISTER|" + username + "|" + password + "|" + typeChoice;      std::string response;      if (sendRequest(request, response))      {          std::istringstream iss(response);          std::string status, message;          std::getline(iss, status, '|');          std::getline(iss, message, '|');          if (status == "REGISTER\_SUCCESS")          {              std::cout << "注册成功！用户名: " << message << std::endl;          }          else          {              std::cout << "注册失败: " << message << std::endl;          }      }      else      {          std::cout << "与服务器通信失败。" << std::endl;      }  }  void Client::handleBrowseProducts()  {      std::string request = "GET\_PRODUCTS";      std::string response;      if (sendRequest(request, response))      {          std::istringstream iss(response);          std::string status, countStr;          std::getline(iss, status, '|');          if (status == "PRODUCTS")          {              std::getline(iss, countStr, '|');              int count = std::stoi(countStr);              std::cout << "\n===== 商品列表 =====\n";              std::cout << "共 " << count << " 个商品\n\n";              for (int i = 0; i < count; i++)              {                  std::string item;                  std::getline(iss, item, '|');                  std::istringstream itemStream(item);                  std::string name, type, orpriceStr, priceStr, quantityStr, seller;                  std::getline(itemStream, name, ',');                  std::getline(itemStream, type, ',');                  std::getline(itemStream, orpriceStr, ',');                  std::getline(itemStream, priceStr, ',');                  std::getline(itemStream, quantityStr, ',');                  std::getline(itemStream, seller, ',');                  std::cout << i + 1 << ". " << name << " (" << type << ")\n";                  if (orpriceStr == priceStr)                  {                      std::cout << "   价格: " << priceStr << " 元, 库存: " << quantityStr << ", 商家: " << seller << "\n\n";                  }                  else                  {                      std::cout << "   折扣价: " << priceStr << " 元 (原价: " << orpriceStr << " 元), 库存: " << quantityStr << ", 商家: " << seller << "\n\n";                  }              }          }          else          {              std::cout << "获取商品列表失败。" << std::endl;          }      }      else      {          std::cout << "与服务器通信失败。" << std::endl;      }      std::cout << "按回车键继续...";      std::cin.get();  }  void Client::handleProductDetail()  {      std::string productName, sellerName;      std::cout << "\n===== 商品详情 =====\n";      std::cout << "输入商品名称: ";      std::getline(std::cin, productName);      std::cout << "输入商家名称: ";      std::getline(std::cin, sellerName);      std::string request = "GET\_PRODUCT\_DETAIL|" + productName + "|" + sellerName;      std::string response;      if (sendRequest(request, response))      {          std::istringstream iss(response);          std::string status;          std::getline(iss, status, '|');          if (status == "PRODUCT\_DETAIL")          {              std::string name, type, description, priceStr, quantityStr, seller;              std::getline(iss, name, '|');              std::getline(iss, type, '|');              std::getline(iss, description, '|');              std::getline(iss, priceStr, '|');              std::getline(iss, quantityStr, '|');              std::getline(iss, seller, '|');              std::cout << "商品名称: " << name << "\n";              std::cout << "类型: " << type << "\n";              std::cout << "描述: " << description << "\n";              std::cout << "价格: " << priceStr << " 元\n";              std::cout << "库存: " << quantityStr << "\n";              std::cout << "商家: " << seller << "\n";              if (type == "Book")              {                  std::string author, isbn;                  std::getline(iss, author, '|');                  std::getline(iss, isbn, '|');                  std::cout << "作者: " << author << "\n";                  std::cout << "ISBN: " << isbn << "\n";              }              else if (type == "Clothing")              {                  std::string size, color;                  std::getline(iss, size, '|');                  std::getline(iss, color, '|');                  std::cout << "尺寸: " << size << "\n";                  std::cout << "颜色: " << color << "\n";              }              else if (type == "Food")              {                  std::string expDate;                  std::getline(iss, expDate, '|');                  std::cout << "保质期: " << expDate << "\n";              }          }          else          {              std::string message;              std::getline(iss, message, '|');              std::cout << "获取商品详情失败: " << message << std::endl;          }      }      else      {          std::cout << "与服务器通信失败。" << std::endl;      }      std::cout << "按回车键继续...";      std::cin.get();  }  void Client::handleViewCart()  {      if (!isLoggedIn || currentUserType != "customer")      {          std::cout << "请先以消费者身份登录。" << std::endl;          return;      }      std::string request = "GET\_CART|" + currentUsername;      std::string response;      if (sendRequest(request, response))      {          std::istringstream iss(response);          std::string status, countStr;          std::getline(iss, status, '|');          if (status == "CART")          {              std::getline(iss, countStr, '|');              int count = std::stoi(countStr);              if (count == 0)              {                  std::cout << "\n您的购物车是空的。" << std::endl;                  std::cout << "按回车键继续...";                  std::cin.get();                  return;              }              // 解析购物车项目              std::vector<std::pair<std::string, std::string>> cartItems; // 商品名称，商家名称              std::vector<int> quantities;                                // 数量              std::vector<double> prices;                                 // 单价              double totalAmount = 0.0;              std::cout << "\n--- 我的购物车 ---" << std::endl;              std::cout << std::fixed << std::setprecision(2); // 设置输出格式              for (int i = 0; i < count; i++)              {                  std::string item;                  std::getline(iss, item, '|');                  std::istringstream itemStream(item);                  std::string name, quantityStr, priceStr, seller;                  std::getline(itemStream, name, ',');                  std::getline(itemStream, quantityStr, ',');                  std::getline(itemStream, priceStr, ',');                  std::getline(itemStream, seller, ',');                  int quantity = std::stoi(quantityStr);                  double price = std::stod(priceStr);                  double itemTotal = quantity \* price;                  totalAmount += itemTotal;                  cartItems.push\_back(std::make\_pair(name, seller));                  quantities.push\_back(quantity);                  prices.push\_back(price);                  std::cout << i + 1 << ". 商品: " << name                            << " | 数量: " << quantity                            << " | 当前单价: ¥" << price                            << " | 小计: ¥" << itemTotal << std::endl;                  // 获取商品详情，检查库存和价格变化                  std::string detailRequest = "GET\_PRODUCT\_DETAIL|" + name + "|" + seller;                  std::string detailResponse;                  if (sendRequest(detailRequest, detailResponse))                  {                      std::istringstream detailIss(detailResponse);                      std::string detailStatus;                      std::getline(detailIss, detailStatus, '|');                      if (detailStatus == "PRODUCT\_DETAIL")                      {                          std::string productName, productType, productDesc, currentPriceStr, currentQuantityStr, productSeller;                          std::getline(detailIss, productName, '|');                          std::getline(detailIss, productType, '|');                          std::getline(detailIss, productDesc, '|');                          std::getline(detailIss, currentPriceStr, '|');                          std::getline(detailIss, currentQuantityStr, '|');                          double currentPrice = std::stod(currentPriceStr);                          int currentQuantity = std::stoi(currentQuantityStr);                          if (currentQuantity < quantity)                          {                              std::cout << "   注意: " << name << " 当前库存(" << currentQuantity                                        << ")不足购物车数量(" << quantity << ")!" << std::endl;                          }                          if (currentPrice != price)                          {                              std::cout << "   提示: " << name << " 加入时价格为 ¥" << price                                        << ", 当前价格已变为 ¥" << currentPrice << std::endl;                          }                      }                      else                      {                          std::cout << "   警告: 商品 " << name << " 可能已从商店下架!" << std::endl;                      }                  }              }              std::cout << "--------------------" << std::endl;              std::cout << "购物车总计: ¥" << totalAmount << std::endl;              std::cout << "--------------------" << std::endl;              // 购物车操作菜单              int cartChoice = -1;              do              {                  std::cout << "\n购物车操作:" << std::endl;                  std::cout << "1. 修改商品数量" << std::endl;                  std::cout << "2. 移除商品" << std::endl;                  std::cout << "3. 生成订单并结算" << std::endl;                  std::cout << "0. 返回商城" << std::endl;                  std::cout << "请选择: ";                  if (!(std::cin >> cartChoice))                  {                      std::cout << "无效输入。" << std::endl;                      clearInputBuffer();                      cartChoice = -1;                      continue;                  }                  clearInputBuffer();                  switch (cartChoice)                  {                  case 1: // 修改商品数量                  {                      std::string nameToUpdate;                      std::string sellerName;                      int newQty;                      std::cout << "输入要修改数量的商品名称: ";                      std::getline(std::cin >> std::ws, nameToUpdate);                      // 查找商品对应的商家                      bool found = false;                      for (size\_t i = 0; i < cartItems.size(); i++)                      {                          if (cartItems[i].first == nameToUpdate)                          {                              sellerName = cartItems[i].second;                              found = true;                              break;                          }                      }                      if (!found)                      {                          std::cout << "未在购物车中找到商品: " << nameToUpdate << std::endl;                          break;                      }                      std::cout << "输入新的数量 (输入0将移除该商品): ";                      if (!(std::cin >> newQty) || newQty < 0)                      {                          std::cout << "无效数量输入。" << std::endl;                          clearInputBuffer();                          break;                      }                      clearInputBuffer();                      // 发送更新请求                      std::string updateRequest;                      if (newQty == 0)                      {                          updateRequest = "REMOVE\_FROM\_CART|" + currentUsername + "|" + nameToUpdate + "|" + sellerName;                      }                      else                      {                          updateRequest = "UPDATE\_CART\_ITEM|" + currentUsername + "|" + nameToUpdate + "|" + sellerName + "|" + std::to\_string(newQty);                      }                      std::string updateResponse;                      if (sendRequest(updateRequest, updateResponse))                      {                          std::istringstream updateIss(updateResponse);                          std::string updateStatus, updateMessage;                          std::getline(updateIss, updateStatus, '|');                          std::getline(updateIss, updateMessage, '|');                          if (updateStatus == "CART\_UPDATED")                          {                              std::cout << "购物车已更新: " << updateMessage << std::endl;                              cartChoice = 0; // 刷新购物车                              handleViewCart();                          }                          else                          {                              std::cout << "更新失败: " << updateMessage << std::endl;                          }                      }                      else                      {                          std::cout << "与服务器通信失败。" << std::endl;                      }                      break;                  }                  case 2: // 移除商品                  {                      std::string nameToRemove;                      std::string sellerName;                      std::cout << "输入要移除的商品名称: ";                      std::getline(std::cin >> std::ws, nameToRemove);                      // 查找商品对应的商家                      bool found = false;                      for (size\_t i = 0; i < cartItems.size(); i++)                      {                          if (cartItems[i].first == nameToRemove)                          {                              sellerName = cartItems[i].second;                              found = true;                              break;                          }                      }                      if (!found)                      {                          std::cout << "未在购物车中找到商品: " << nameToRemove << std::endl;                          break;                      }                      // 发送移除请求                      std::string removeRequest = "REMOVE\_FROM\_CART|" + currentUsername + "|" + nameToRemove + "|" + sellerName;                      std::string removeResponse;                      if (sendRequest(removeRequest, removeResponse))                      {                          std::istringstream removeIss(removeResponse);                          std::string removeStatus, removeMessage;                          std::getline(removeIss, removeStatus, '|');                          std::getline(removeIss, removeMessage, '|');                          if (removeStatus == "CART\_UPDATED")                          {                              std::cout << "购物车已更新: " << removeMessage << std::endl;                              cartChoice = 0; // 刷新购物车                              handleViewCart();                          }                          else                          {                              std::cout << "移除失败: " << removeMessage << std::endl;                          }                      }                      else                      {                          std::cout << "与服务器通信失败。" << std::endl;                      }                      break;                  }                  case 3: // 生成订单并结算                  {                      // 首先获取用户余额                      std::string balanceRequest = "CHECK\_BALANCE|" + currentUsername;                      std::string balanceResponse;                      double balance = 0.0;                      if (sendRequest(balanceRequest, balanceResponse))                      {                          std::istringstream balanceIss(balanceResponse);                          std::string balanceStatus, balanceStr;                          std::getline(balanceIss, balanceStatus, '|');                          if (balanceStatus == "BALANCE")                          {                              std::getline(balanceIss, balanceStr, '|');                              balance = std::stod(balanceStr);                          }                      }                      // 显示订单预览                      std::cout << "\n===== 订单预览 =====\n";                      for (size\_t i = 0; i < cartItems.size(); i++)                      {                          std::cout << i + 1 << ". " << cartItems[i].first                                    << " x " << quantities[i]                                    << " @ ¥" << prices[i]                                    << " = ¥" << (quantities[i] \* prices[i]) << std::endl;                      }                      std::cout << "--------------------" << std::endl;                      std::cout << "订单总计: ¥" << totalAmount << std::endl;                      std::cout << "您的当前余额: ¥" << std::fixed << std::setprecision(2) << balance << std::endl;                      std::cout << "支付后余额将为: ¥" << (balance - totalAmount) << std::endl;                      std::cout << "--------------------" << std::endl;                      if (balance < totalAmount)                      {                          std::cout << "错误: 您的余额不足以支付此订单！" << std::endl;                          break;                      }                      // 确认支付                      char confirmChoice;                      std::cout << "确认支付并完成订单吗? (y/n): ";                      std::cin >> confirmChoice;                      clearInputBuffer();                      if (tolower(confirmChoice) != 'y')                      {                          std::cout << "订单已取消。" << std::endl;                          break;                      }                      // 提交结算请求                      std::cout << "正在处理订单，请稍候..." << std::endl;                      std::string checkoutRequest = "CHECKOUT|" + currentUsername;                      std::string checkoutResponse;                      if (sendRequest(checkoutRequest, checkoutResponse))                      {                          std::istringstream checkoutIss(checkoutResponse);                          std::string checkoutStatus, checkoutMessage, orderId;                          std::getline(checkoutIss, checkoutStatus, '|');                          std::getline(checkoutIss, checkoutMessage, '|');                          if (checkoutStatus == "CHECKOUT\_SUCCESS")                          {                              std::getline(checkoutIss, orderId, '|');                              std::cout << "结算成功: " << checkoutMessage << std::endl;                              std::cout << "订单ID: " << orderId << std::endl;                              // 订单处理成功，返回商城                              cartChoice = 0;                          }                          else if (checkoutStatus == "CHECKOUT\_PENDING")                          {                              std::getline(checkoutIss, orderId, '|');                              std::cout << "订单已提交但正在处理中: " << checkoutMessage << std::endl;                              std::cout << "订单ID: " << orderId << std::endl;                              std::cout << "您可以稍后在订单记录中查看订单状态。" << std::endl;                              // 返回商城                              cartChoice = 0;                          }                          else                          {                              std::cout << "结算失败: " << checkoutMessage << std::endl;                          }                      }                      else                      {                          std::cout << "与服务器通信失败。" << std::endl;                      }                      break;                  }                  case 0: // 返回商城                      break;                  case -2:                     // 刷新购物车                      return handleViewCart(); // 递归调用以刷新购物车                  default:                      std::cout << "无效选项。" << std::endl;                  }              } while (cartChoice != 0);          }          else          {              std::string message;              std::getline(iss, message, '|');              std::cout << "获取购物车失败: " << message << std::endl;          }      }      else      {          std::cout << "与服务器通信失败。" << std::endl;      }      // std::cout << "按回车键继续...";      // std::cin.get();  }  void Client::handleAddToCart()  {      if (!isLoggedIn || currentUserType != "customer")      {          std::cout << "请先以消费者身份登录。" << std::endl;          return;      }      std::string productName, sellerName;      int quantity;      std::cout << "\n===== 添加商品到购物车 =====\n";      std::cout << "输入商品名称: ";      std::getline(std::cin, productName);      std::cout << "输入商家名称: ";      std::getline(std::cin, sellerName);      std::cout << "输入数量: ";      std::cin >> quantity;      clearInputBuffer();      if (quantity <= 0)      {          std::cout << "数量必须大于0。" << std::endl;          return;      }      std::string request = "ADD\_TO\_CART|" + currentUsername + "|" + productName + "|" + sellerName + "|" + std::to\_string(quantity);      std::string response;      if (sendRequest(request, response))      {          std::istringstream iss(response);          std::string status, message;          std::getline(iss, status, '|');          std::getline(iss, message, '|');          if (status == "CART\_UPDATED")          {              std::cout << "成功: " << message << std::endl;          }          else          {              std::cout << "添加失败: " << message << std::endl;          }      }      else      {          std::cout << "与服务器通信失败。" << std::endl;      }      std::cout << "按回车键继续...";      std::cin.get();  }  void Client::handleCheckout()  {      if (!isLoggedIn || currentUserType != "customer")      {          std::cout << "请先以消费者身份登录。" << std::endl;          return;      }      std::string request = "CHECKOUT|" + currentUsername;      std::string response;      if (sendRequest(request, response))      {          std::istringstream iss(response);          std::string status, message, orderId;          std::getline(iss, status, '|');          std::getline(iss, message, '|');          if (status == "CHECKOUT\_SUCCESS")          {              std::getline(iss, orderId, '|');              std::cout << "结算成功: " << message << "\n";              std::cout << "订单号: " << orderId << std::endl;          }          else          {              std::cout << "结算失败: " << message << std::endl;          }      }      else      {          std::cout << "与服务器通信失败。" << std::endl;      }      std::cout << "按回车键继续...";      std::cin.get();  }  // 商家功能实现  void Client::handleSellerInfo()  {      if (!isLoggedIn || currentUserType != "seller")      {          std::cout << "请先以商家身份登录。" << std::endl;          return;      }      std::string request = "GET\_SELLER\_INFO|" + currentUsername;      std::string response;      if (sendRequest(request, response))      {          std::istringstream iss(response);          std::string status;          std::getline(iss, status, '|');          if (status == "SELLER\_INFO")          {              std::string username, userType, balanceStr;              std::getline(iss, username, '|');              std::getline(iss, userType, '|');              std::getline(iss, balanceStr, '|');              std::cout << "\n===== 商家信息 =====\n";              std::cout << "用户名: " << username << "\n";              std::cout << "账户类型: " << userType << "\n";              std::cout << "账户余额: " << balanceStr << " 元\n";          }          else          {              std::string message;              std::getline(iss, message, '|');              std::cout << "获取商家信息失败: " << message << std::endl;          }      }      else      {          std::cout << "与服务器通信失败。" << std::endl;      }      std::cout << "按回车键继续...";      std::cin.get();  }  void Client::handleChangePassword()  {      if (!isLoggedIn)      {          std::cout << "请先登录。" << std::endl;          return;      }      std::string oldPassword, newPassword;      std::cout << "\n===== 修改密码 =====\n";      std::cout << "请输入原密码: ";      std::getline(std::cin, oldPassword);      std::cout << "请输入新密码: ";      std::getline(std::cin, newPassword);      std::string request = "CHANGE\_PASSWORD|" + currentUsername + "|" + oldPassword + "|" + newPassword;      std::string response;      if (sendRequest(request, response))      {          std::istringstream iss(response);          std::string status, message;          std::getline(iss, status, '|');          std::getline(iss, message, '|');          if (status == "PASSWORD\_CHANGED")          {              std::cout << "密码修改成功。" << std::endl;          }          else          {              std::cout << "密码修改失败: " << message << std::endl;          }      }      else      {          std::cout << "与服务器通信失败。" << std::endl;      }      std::cout << "按回车键继续...";      std::cin.get();  }  void Client::handleManageProducts()  {      if (!isLoggedIn || currentUserType != "seller")      {          std::cout << "请先以商家身份登录。" << std::endl;          return;      }      while (true)      {          int choice;          std::cout << "\n===== 商品管理 =====\n";          std::cout << "1. 查看我的商品\n";          std::cout << "2. 添加新商品\n";          std::cout << "3. 修改商品价格\n";          std::cout << "4. 修改商品库存\n";          std::cout << "5. 设置商品折扣\n";          std::cout << "0. 返回上级菜单\n";          std::cout << "请选择: ";          std::cin >> choice;          clearInputBuffer();          if (choice == 0)          {              break;          }          switch (choice)          {          case 1:          {              // 查看我的商品              std::string request = "GET\_SELLER\_PRODUCTS|" + currentUsername;              std::string response;              if (sendRequest(request, response))              {                  std::istringstream iss(response);                  std::string status, countStr;                  std::getline(iss, status, '|');                  if (status == "SELLER\_PRODUCTS")                  {                      std::getline(iss, countStr, '|');                      int count = std::stoi(countStr);                      std::cout << "\n===== 我的商品 =====\n";                      std::cout << "共 " << count << " 个商品\n\n";                      for (int i = 0; i < count; i++)                      {                          std::string item;                          std::getline(iss, item, '|');                          std::istringstream itemStream(item);                          std::string name, type, priceStr, quantityStr, discountStr;                          std::getline(itemStream, name, ',');                          std::getline(itemStream, type, ',');                          std::getline(itemStream, priceStr, ',');                          std::getline(itemStream, quantityStr, ',');                          std::getline(itemStream, discountStr, ',');                          double discount = std::stod(discountStr);                          double displayPrice = std::stod(priceStr) \* (1.0 - discount);                          std::cout << i + 1 << ". " << name << " (" << type << ")\n";                          std::cout << "   原价: " << priceStr << " 元, 折扣: " << discount \* 100 << "%\n";                          std::cout << "   售价: " << displayPrice << " 元, 库存: " << quantityStr << "\n\n";                      }                  }                  else                  {                      std::string message;                      std::getline(iss, message, '|');                      std::cout << "获取商品列表失败: " << message << std::endl;                  }              }              else              {                  std::cout << "与服务器通信失败。" << std::endl;              }              std::cout << "按回车键继续...";              std::cin.get();              break;          }          case 2:          {              // 添加新商品              int productType;              std::string name, description;              double price;              int quantity;              std::cout << "\n===== 添加新商品 =====\n";              std::cout << "选择商品类型 (1: 书籍, 2: 服装, 3: 食品, 4: 其他): ";              std::cin >> productType;              clearInputBuffer();              std::cout << "商品名称: ";              std::getline(std::cin, name);              std::cout << "商品描述: ";              std::getline(std::cin, description);              std::cout << "商品价格: ";              std::cin >> price;              std::cout << "商品库存: ";              std::cin >> quantity;              clearInputBuffer();              if (price < 0 || quantity < 0)              {                  std::cout << "价格和库存不能为负数。" << std::endl;                  break;              }              std::string request;              if (productType == 1)              { // 书籍                  std::string author, isbn;                  std::cout << "作者: ";                  std::getline(std::cin, author);                  std::cout << "ISBN: ";                  std::getline(std::cin, isbn);                  request = "ADD\_BOOK|" + currentUsername + "|" + name + "|" + description + "|" +                            std::to\_string(price) + "|" + std::to\_string(quantity) + "|" + author + "|" + isbn;              }              else if (productType == 2)              { // 服装                  std::string size, color;                  std::cout << "尺寸: ";                  std::getline(std::cin, size);                  std::cout << "颜色: ";                  std::getline(std::cin, color);                  request = "ADD\_CLOTHING|" + currentUsername + "|" + name + "|" + description + "|" +                            std::to\_string(price) + "|" + std::to\_string(quantity) + "|" + size + "|" + color;              }              else if (productType == 3)              { // 食品                  std::string expDate;                  std::cout << "保质期 (YYYY-MM-DD): ";                  std::getline(std::cin, expDate);                  request = "ADD\_FOOD|" + currentUsername + "|" + name + "|" + description + "|" +                            std::to\_string(price) + "|" + std::to\_string(quantity) + "|" + expDate;              }              else if (productType == 4)              { // 其他                  std::string category;                  std::cout << "商品分类: ";                  std::getline(std::cin, category);                  request = "ADD\_GENERIC|" + currentUsername + "|" + name + "|" + description + "|" +                            std::to\_string(price) + "|" + std::to\_string(quantity) + "|" + category;              }              else              {                  std::cout << "无效的商品类型。" << std::endl;                  break;              }              std::string response;              if (sendRequest(request, response))              {                  std::istringstream iss(response);                  std::string status, message;                  std::getline(iss, status, '|');                  std::getline(iss, message, '|');                  if (status == "PRODUCT\_ADDED")                  {                      std::cout << "添加成功: " << message << std::endl;                  }                  else                  {                      std::cout << "添加失败: " << message << std::endl;                  }              }              else              {                  std::cout << "与服务器通信失败。" << std::endl;              }              std::cout << "按回车键继续...";              std::cin.get();              break;          }          case 3:          {              // 修改商品价格              std::string productName;              double newPrice;              std::cout << "\n===== 修改商品价格 =====\n";              std::cout << "商品名称: ";              std::getline(std::cin, productName);              std::cout << "新价格: ";              std::cin >> newPrice;              clearInputBuffer();              if (newPrice < 0)              {                  std::cout << "价格不能为负数。" << std::endl;                  break;              }              std::string request = "UPDATE\_PRODUCT\_PRICE|" + currentUsername + "|" + productName + "|" + std::to\_string(newPrice);              std::string response;              if (sendRequest(request, response))              {                  std::istringstream iss(response);                  std::string status, message;                  std::getline(iss, status, '|');                  std::getline(iss, message, '|');                  if (status == "PRODUCT\_UPDATED")                  {                      std::cout << "价格修改成功: " << message << std::endl;                  }                  else                  {                      std::cout << "价格修改失败: " << message << std::endl;                  }              }              else              {                  std::cout << "与服务器通信失败。" << std::endl;              }              std::cout << "按回车键继续...";              std::cin.get();              break;          }          case 4:          {              // 修改商品库存              std::string productName;              int newQuantity;              std::cout << "\n===== 修改商品库存 =====\n";              std::cout << "商品名称: ";              std::getline(std::cin, productName);              std::cout << "新库存: ";              std::cin >> newQuantity;              clearInputBuffer();              if (newQuantity < 0)              {                  std::cout << "库存不能为负数。" << std::endl;                  break;              }              std::string request = "UPDATE\_PRODUCT\_QUANTITY|" + currentUsername + "|" + productName + "|" + std::to\_string(newQuantity);              std::string response;              if (sendRequest(request, response))              {                  std::istringstream iss(response);                  std::string status, message;                  std::getline(iss, status, '|');                  std::getline(iss, message, '|');                  if (status == "PRODUCT\_UPDATED")                  {                      std::cout << "库存修改成功: " << message << std::endl;                  }                  else                  {                      std::cout << "库存修改失败: " << message << std::endl;                  }              }              else              {                  std::cout << "与服务器通信失败。" << std::endl;              }              std::cout << "按回车键继续...";              std::cin.get();              break;          }          case 5:          {              // 设置商品折扣              int discountType;              std::cout << "\n===== 设置商品折扣 =====\n";              std::cout << "选择折扣类型 (1: 单个商品, 2: 商品分类): ";              std::cin >> discountType;              clearInputBuffer();              if (discountType == 1)              {                  std::string productName;                  double discount;                  std::cout << "商品名称: ";                  std::getline(std::cin, productName);                  std::cout << "折扣率 (0-100): ";                  std::cin >> discount;                  std::cin.ignore(std::numeric\_limits<std::streamsize>::max(), '\n');                  if (discount < 0 || discount > 100)                  {                      std::cout << "折扣率必须在0-100之间。" << std::endl;                      break;                  }                  std::string request = "SET\_PRODUCT\_DISCOUNT|" + currentUsername + "|" + productName + "|" + std::to\_string(discount / 100);                  std::string response;                  if (sendRequest(request, response))                  {                      std::istringstream iss(response);                      std::string status, message;                      std::getline(iss, status, '|');                      std::getline(iss, message, '|');                      if (status == "PRODUCT\_UPDATED")                      {                          std::cout << "折扣设置成功: " << message << std::endl;                      }                      else                      {                          std::cout << "折扣设置失败: " << message << std::endl;                      }                  }                  else                  {                      std::cout << "与服务器通信失败。" << std::endl;                  }              }              else if (discountType == 2)              {                  std::string category;                  double discount;                  std::cout << "商品分类: ";                  std::getline(std::cin, category);                  std::cout << "折扣率 (0.0-1.0): ";                  std::cin >> discount;                  std::cin.ignore(std::numeric\_limits<std::streamsize>::max(), '\n');                  if (discount < 0.0 || discount > 1.0)                  {                      std::cout << "折扣率必须在0.0-1.0之间。" << std::endl;                      break;                  }                  std::string request = "SET\_CATEGORY\_DISCOUNT|" + currentUsername + "|" + category + "|" + std::to\_string(discount);                  std::string response;                  if (sendRequest(request, response))                  {                      std::istringstream iss(response);                      std::string status, message;                      std::getline(iss, status, '|');                      std::getline(iss, message, '|');                      if (status == "CATEGORY\_UPDATED")                      {                          std::cout << "分类折扣设置成功: " << message << std::endl;                      }                      else                      {                          std::cout << "分类折扣设置失败: " << message << std::endl;                      }                  }                  else                  {                      std::cout << "与服务器通信失败。" << std::endl;                  }              }              else              {                  std::cout << "无效的折扣类型。" << std::endl;              }              std::cout << "按回车键继续...";              std::cin.get();              break;          }          default:              std::cout << "无效选择，请重试。" << std::endl;              break;          }      }  }  void Client::handleCheckIncome()  {      if (!isLoggedIn || currentUserType != "seller")      {          std::cout << "请先以商家身份登录。" << std::endl;          return;      }      std::string request = "CHECK\_BALANCE|" + currentUsername;      std::string response;      if (sendRequest(request, response))      {          std::istringstream iss(response);          std::string status, balanceStr;          std::getline(iss, status, '|');          if (status == "BALANCE")          {              std::getline(iss, balanceStr, '|');              double balance = std::stod(balanceStr);              std::cout << "\n===== 账户收入 =====\n";              std::cout << std::fixed << std::setprecision(2);              std::cout << "当前收入: " << balance << " 元\n";          }          else          {              std::string message;              std::getline(iss, message, '|');              std::cout << "查询收入失败: " << message << std::endl;          }      }      else      {          std::cout << "与服务器通信失败。" << std::endl;      }      std::cout << "按回车键继续...";      std::cin.get();  }  void Client::handleViewUserInfo()  {      if (!isLoggedIn)      {          std::cout << "请先登录。" << std::endl;          return;      }      std::string request = "GET\_USER\_INFO|" + currentUsername;      std::string response;      if (sendRequest(request, response))      {          std::istringstream iss(response);          std::string status;          std::getline(iss, status, '|');          if (status == "USER\_INFO")          {              std::string username, userType, balanceStr;              std::getline(iss, username, '|');              std::getline(iss, userType, '|');              std::getline(iss, balanceStr, '|');              std::cout << "\n===== 用户信息 =====\n";              std::cout << "用户名: " << username << "\n";              std::cout << "账户类型: " << userType << "\n";              std::cout << "账户余额: " << balanceStr << " 元\n";          }          else          {              std::string message;              std::getline(iss, message, '|');              std::cout << "获取用户信息失败: " << message << std::endl;          }      }      else      {          std::cout << "与服务器通信失败。" << std::endl;      }      std::cout << "按回车键继续...";      std::cin.get();  }  void Client::handleCheckBalance()  {      if (!isLoggedIn)      {          std::cout << "请先登录。" << std::endl;          return;      }      std::string request = "CHECK\_BALANCE|" + currentUsername;      std::string response;      if (sendRequest(request, response))      {          std::istringstream iss(response);          std::string status, balanceStr;          std::getline(iss, status, '|');          if (status == "BALANCE")          {              std::getline(iss, balanceStr, '|');              double balance = std::stod(balanceStr);              std::cout << "\n===== 账户余额 =====\n";              std::cout << std::fixed << std::setprecision(2);              std::cout << "当前余额: " << balance << " 元\n";          }          else          {              std::string message;              std::getline(iss, message, '|');              std::cout << "查询余额失败: " << message << std::endl;          }      }      else      {          std::cout << "与服务器通信失败。" << std::endl;      }      std::cout << "按回车键继续...";      std::cin.get();  }  void Client::handleDeposit()  {      if (!isLoggedIn)      {          std::cout << "请先登录。" << std::endl;          return;      }      double amount;      std::cout << "\n===== 账户充值 =====\n";      std::cout << "请输入充值金额: ";      if (!(std::cin >> amount) || amount <= 0)      {          std::cout << "无效金额。" << std::endl;          clearInputBuffer();          return;      }      clearInputBuffer();      std::string request = "DEPOSIT|" + currentUsername + "|" + std::to\_string(amount);      std::string response;      if (sendRequest(request, response))      {          std::istringstream iss(response);          std::string status, message, balanceStr;          std::getline(iss, status, '|');          if (status == "DEPOSIT\_SUCCESS")          {              std::getline(iss, message, '|');              std::getline(iss, balanceStr, '|');              double balance = std::stod(balanceStr);              std::cout << "充值成功。当前余额: " << std::fixed << std::setprecision(2) << balance << " 元" << std::endl;          }          else          {              std::getline(iss, message, '|');              std::cout << "充值失败: " << message << std::endl;          }      }      else      {          std::cout << "与服务器通信失败。" << std::endl;      }      std::cout << "按回车键继续...";      std::cin.get();  }  void Client::handleEnterStore()  {      int choice = -1;      do      {          std::cout << "\n--- 商城 (用户: " << currentUsername << ") ---\n";          std::cout << "1. 显示所有商品\n";          std::cout << "2. 按名称搜索商品\n";          std::cout << "3. 直接购买商品\n";          std::cout << "4. 加入购物车\n";          std::cout << "5. 查看我的购物车\n";          std::cout << "0. 退出商城\n";          std::cout << "请选择: ";          if (!(std::cin >> choice))          {              std::cout << "无效输入。" << std::endl;              clearInputBuffer();              choice = -1;              continue;          }          clearInputBuffer();          switch (choice)          {          case 1:              handleBrowseProducts();              break;          case 2:              handleSearchProducts();              break;          case 3:              handleDirectPurchase();              break;          case 4:              handleAddToCart();              break;          case 5:              handleViewCart();              break;          case 0:              std::cout << "正在退出商城..." << std::endl;              break;          default:              std::cout << "无效选项。" << std::endl;          }      } while (choice != 0);  }  void Client::handleLogout()  {      if (!isLoggedIn)      {          std::cout << "您当前未登录！" << std::endl;          return;      }      std::string request = "LOGOUT|" + currentUsername;      std::string response;      if (sendRequest(request, response))      {          std::istringstream iss(response);          std::string status, message;          std::getline(iss, status, '|');          std::getline(iss, message, '|');          if (status == "LOGOUT\_SUCCESS")          {              isLoggedIn = false;              currentUsername = "";              currentUserType = "";              std::cout << "已退出登录: " << message << std::endl;          }          else          {              std::cout << "登出失败: " << message << std::endl;          }      }      else      {          std::cout << "与服务器通信失败。" << std::endl;      }      std::cout << "按回车键继续...";      std::cin.get();  }  void Client::handleSearchProducts()  {      if (!isLoggedIn)      {          std::cout << "请先登录。" << std::endl;          return;      }      std::string searchTerm;      std::cout << "输入搜索名称: ";      std::getline(std::cin >> std::ws, searchTerm);      std::string request = "SEARCH\_PRODUCTS|" + searchTerm;      std::string response;      if (sendRequest(request, response))      {          std::istringstream iss(response);          std::string status, countStr;          std::getline(iss, status, '|');          if (status == "SEARCH\_RESULTS")          {              std::getline(iss, countStr, '|');              int count = std::stoi(countStr);              if (count == 0)              {                  std::cout << "未找到匹配商品。" << std::endl;              }              else              {                  std::cout << "\n--- 搜索结果 ---" << std::endl;                  for (int i = 0; i < count; i++)                  {                      std::string item;                      std::getline(iss, item, '|');                      std::istringstream itemStream(item);                      std::string name, type, priceStr, quantityStr, seller, description;                      std::getline(itemStream, name, ',');                      std::getline(itemStream, type, ',');                      std::getline(itemStream, priceStr, ',');                      std::getline(itemStream, quantityStr, ',');                      std::getline(itemStream, seller, ',');                      std::getline(itemStream, description, ',');                      std::cout << "-----------------" << std::endl;                      std::cout << "商品名称: " << name << std::endl;                      std::cout << "类型: " << type << std::endl;                      std::cout << "价格: " << priceStr << " 元" << std::endl;                      std::cout << "库存: " << quantityStr << std::endl;                      std::cout << "商家: " << seller << std::endl;                      std::cout << "描述: " << description << std::endl;                  }                  std::cout << "-----------------" << std::endl;              }          }          else          {              std::string message;              std::getline(iss, message, '|');              std::cout << "搜索失败: " << message << std::endl;          }      }      else      {          std::cout << "与服务器通信失败。" << std::endl;      }      std::cout << "按回车键继续...";      std::cin.get();  }  void Client::handleDirectPurchase()  {      if (!isLoggedIn || currentUserType != "customer")      {          std::cout << "请先以消费者身份登录后再购买商品！" << std::endl;          return;      }      std::string productName, sellerName;      int quantity;      std::cout << "\n===== 直接购买商品 =====\n";      std::cout << "输入要购买的商品名称: ";      std::getline(std::cin >> std::ws, productName);      std::cout << "输入商家名称: ";      std::getline(std::cin >> std::ws, sellerName);      // 先获取商品详情      std::string detailRequest = "GET\_PRODUCT\_DETAIL|" + productName + "|" + sellerName;      std::string detailResponse;      if (!sendRequest(detailRequest, detailResponse))      {          std::cout << "与服务器通信失败，无法获取商品详情。" << std::endl;          std::cout << "按回车键继续...";          std::cin.get();          return;      }      std::istringstream detailIss(detailResponse);      std::string detailStatus;      std::getline(detailIss, detailStatus, '|');      if (detailStatus != "PRODUCT\_DETAIL")      {          std::string errorMsg;          std::getline(detailIss, errorMsg, '|');          std::cout << "未找到商品: " << errorMsg << std::endl;          std::cout << "按回车键继续...";          std::cin.get();          return;      }      // 商品存在，获取价格和库存      std::string name, type, description, priceStr, quantityStr, seller;      std::getline(detailIss, name, '|');      std::getline(detailIss, type, '|');      std::getline(detailIss, description, '|');      std::getline(detailIss, priceStr, '|');      std::getline(detailIss, quantityStr, '|');      std::getline(detailIss, seller, '|');      double price = std::stod(priceStr);      int stock = std::stoi(quantityStr);      // 输入购买数量      std::cout << "输入购买数量: ";      if (!(std::cin >> quantity) || quantity <= 0)      {          std::cout << "无效购买数量！" << std::endl;          clearInputBuffer();          std::cout << "按回车键继续...";          std::cin.get();          return;      }      clearInputBuffer();      // 检查库存      if (stock < quantity)      {          std::cout << "库存不足！当前库存: " << stock << std::endl;          std::cout << "按回车键继续...";          std::cin.get();          return;      }      // 检查余额      double totalCost = price \* quantity;      // 获取当前余额      std::string balanceRequest = "CHECK\_BALANCE|" + currentUsername;      std::string balanceResponse;      double balance = 0.0;      if (sendRequest(balanceRequest, balanceResponse))      {          std::istringstream balanceIss(balanceResponse);          std::string balanceStatus, balanceStr;          std::getline(balanceIss, balanceStatus, '|');          if (balanceStatus == "BALANCE")          {              std::getline(balanceIss, balanceStr, '|');              balance = std::stod(balanceStr);          }      }      if (balance < totalCost)      {          std::cout << "余额不足！当前余额: " << std::fixed << std::setprecision(2) << balance << " 元" << std::endl;          std::cout << "按回车键继续...";          std::cin.get();          return;      }      // 显示订单信息并请求确认      std::cout << "\n===== 订单确认 =====\n";      std::cout << "商品: " << name << std::endl;      std::cout << "类型: " << type << std::endl;      std::cout << "单价: " << std::fixed << std::setprecision(2) << price << " 元" << std::endl;      std::cout << "数量: " << quantity << std::endl;      std::cout << "总价: " << std::fixed << std::setprecision(2) << totalCost << " 元" << std::endl;      std::cout << "您的当前余额: " << std::fixed << std::setprecision(2) << balance << " 元" << std::endl;      std::cout << "支付后余额将为: " << std::fixed << std::setprecision(2) << (balance - totalCost) << " 元" << std::endl;      std::cout << "--------------------" << std::endl;      char confirmChoice;      std::cout << "确认支付并完成订单吗? (y/n): ";      std::cin >> confirmChoice;      clearInputBuffer();      if (tolower(confirmChoice) != 'y')      {          std::cout << "订单已取消。" << std::endl;          return;      }      // 提交结算请求      std::cout << "正在处理订单，请稍候..." << std::endl;      std::string checkoutRequest = "CHECKOUT|" + currentUsername;      std::string checkoutResponse;      if (sendRequest(checkoutRequest, checkoutResponse))      {          std::istringstream checkoutIss(checkoutResponse);          std::string checkoutStatus, checkoutMessage, orderId;          std::getline(checkoutIss, checkoutStatus, '|');          std::getline(checkoutIss, checkoutMessage, '|');          if (checkoutStatus == "CHECKOUT\_SUCCESS")          {              std::getline(checkoutIss, orderId, '|');              std::cout << "结算成功: " << checkoutMessage << std::endl;              std::cout << "订单ID: " << orderId << std::endl;              // 订单处理成功，返回商城              // cartChoice = 0;          }          else if (checkoutStatus == "CHECKOUT\_PENDING")          {              std::getline(checkoutIss, orderId, '|');              std::cout << "订单已提交但正在处理中: " << checkoutMessage << std::endl;              std::cout << "订单ID: " << orderId << std::endl;              std::cout << "您可以稍后在订单记录中查看订单状态。" << std::endl;              // 返回商城              // cartChoice = 0;          }          else          {              std::cout << "结算失败: " << checkoutMessage << std::endl;          }      }      else      {          std::cout << "与服务器通信失败。" << std::endl;      }      std::cout << "按回车键继续...";      std::cin.get();  }  void Client::cleanup()  {      // 断开连接并清理资源      if (clientSocket != INVALID\_SOCKET)      {          closesocket(clientSocket);          clientSocket = INVALID\_SOCKET;      }      // 清理 Winsock      WSACleanup();  } |

#### Page.h

|  |
| --- |
| #ifndef PAGE\_H  #define PAGE\_H  #include "../store/store.h" // 包含 Store 类定义  #include <vector>  #include <string> // 需要包含 string  // 前向声明 User 类，因为 Page 方法使用了 User\*  // 避免在头文件中包含 user.h，减少编译依赖  class User;  // --- Page 类定义 ---  // 用于管理不同状态下的菜单显示  class Page  {  protected:      // 私有静态辅助方法：清除输入缓冲区 (Page 类内部使用)      // 定义移到 page.cpp      static void clearInputBuffer();  public:      // --- 成员方法：菜单显示 ---      // 主菜单 (未登录)      int pagemain(std::vector<User \*> &users, User \*&currentUser);      // 消费者菜单 (登录后)      int pagemainCustomer(User \*currentUser, std::vector<User \*> &users);      // 商家菜单 (登录后)      int pagemainSeller(User \*currentUser, std::vector<User \*> &users);      // 管理员菜单 (如果需要可以添加)      int pagemainAdmin(User \*currentUser, std::vector<User \*> &users);      virtual int pagestore(User \*currentUser, std::vector<User \*> &users, Store &store) = 0;  };  // --- Page 派生类 ---  class Customer\_page : public Page  {  public:      // 实现商城页面 (确保有 Store& store 参数)      int pagestore(User \*currentUser, std::vector<User \*> &users, Store &store) override;      // 可以覆盖基类的账户菜单方法，如果需要特定行为  };  class Seller\_page : public Page  {  public:      // 实现商城页面 (确保有 Store& store 参数)      int pagestore(User \*currentUser, std::vector<User \*> &users, Store &store) override;      // 可以覆盖基类的账户菜单方法，如果需要特定行为  };  // //应该暂不需要  // class Admin\_page : public Page  // {  //     int pagestore(User \*currentUser, std::vector<User \*> &users);  // };  #endif // PAGE\_H |

#### Page.cpp

|  |
| --- |
| #include "page.h"  #include "../user/user.h"  #include "../store/store.h" // 需要 Store 定义  #include "../ordermanager/ordermanager.h"  #include <iostream>  #include <cstdio> // for printf if used  #include <limits>  #include <vector>  #include <string>  #include <iomanip>  using namespace std;  extern const string USER\_FILE;  // STORE\_FILE is managed by the Store object passed in  // --- 基类 Page 静态方法 ---  void Page::clearInputBuffer()  {      cin.clear();      cin.ignore(numeric\_limits<streamsize>::max(), '\n');  }  // --- 基类 Page 菜单实现 ---  // 这些函数现在只处理用户账户菜单，并在选择商城时返回特定值 3  int Page::pagemain(vector<User \*> &users, User \*&currentUser)  {      int choice = 0;      while (true)      {          printf("\n\n\n--- 主菜单 ---\n");          cout << "1. 注册新用户" << endl;          cout << "2. 用户登录" << endl;          cout << "3. 进入商城 (游客)" << endl; // 返回 3 表示进入商城          cout << "4. 退出程序" << endl;          cout << "请选择一个选项: ";          if (!(cin >> choice))          {              cout << "无效输入。" << endl;              clearInputBuffer();              continue;          }          clearInputBuffer();          switch (choice)          {          case 1:              currentUser = User::registerUser(users);              if (currentUser)              {                  User::saveUsersToFile(users, USER\_FILE);                  // 注册成功后可以选择留在主菜单或直接进入用户菜单                  // return 1; // 如果想直接进入用户菜单              }              return 1;              break; // 留在主菜单          case 2:              currentUser = User::userLogin(users);              if (currentUser)                  return 1; // 登录成功，返回 1 给 main 处理              break;        // 登录失败，留在主菜单          case 3:              return 3; // 返回 3，通知 main 调用 pagestore          case 4:              cout << "退出程序..." << endl;              return 0; // 返回 0 给 main 退出循环          default:              cout << "无效选项。" << endl;          }      }  }  int Page::pagemainCustomer(User \*currentUser, vector<User \*> &users)  {      int choice = 0;      while (true)      {          printf("\n\n\n--- 消费者菜单 (%s) ---\n", currentUser->getUsername().c\_str());          cout << "1. 查看个人信息" << endl;          cout << "2. 修改密码" << endl;          cout << "3. 查询余额" << endl;          cout << "4. 充值" << endl;          cout << "5. 进入商城" << endl; // 返回 3 表示进入商城          cout << "6. 退出登录" << endl;          cout << "请选择一个选项: ";          if (!(cin >> choice))          {              cout << "无效输入。" << endl;              clearInputBuffer();              continue;          }          clearInputBuffer();          switch (choice)          {          case 1:              currentUser->displayInfo();              break;          case 2:          {              string oldP, newP;              cout << "请输入原密码: ";              getline(cin >> ws, oldP);              cout << "请输入新密码: ";              getline(cin >> ws, newP);              if (currentUser->changePassword(oldP, newP))              {                  User::saveUsersToFile(users, USER\_FILE);                  cout << "密码修改成功。" << endl;              }              else              {                  cout << "密码修改失败。" << endl;              }          }          break;          case 3:          {              cout << fixed << setprecision(2); // 设置输出格式              cout << "当前余额: " << currentUser->checkBalance() << endl;              break;          }          case 4:          {              double amt;              cout << "请输入充值金额: ";              if (!(cin >> amt) || amt <= 0)              { // 检查无效输入和非正数                  cout << "无效金额。" << endl;                  clearInputBuffer();                  break;              }              clearInputBuffer();              if (currentUser->deposit(amt))              {                  User::saveUsersToFile(users, USER\_FILE);                  cout << "充值成功。当前余额: " << currentUser->checkBalance() << endl;              }              else              {                  // deposit 内部应该已经打印了错误信息              }          }          break;          case 5:              return 3; // 返回 3，通知 main 调用 pagestore          case 6:              cout << "退出登录..." << endl;              return 2; // 返回 2 给 main 处理退出登录          default:              cout << "无效选项。" << endl;          }      }  }  int Page::pagemainSeller(User \*currentUser, vector<User \*> &users)  {      int choice = 0;      while (true)      {          printf("\n\n\n--- 商家菜单 (%s) ---\n", currentUser->getUsername().c\_str());          cout << "1. 查看商户信息" << endl;          cout << "2. 修改密码" << endl;          cout << "3. 进入商城 (管理)" << endl; // 返回 3 表示进入商城          cout << "4. 查看收入" << endl;          cout << "0. 退出登录" << endl;          cout << "请选择一个选项: ";          if (!(cin >> choice))          {              cout << "无效输入。" << endl;              clearInputBuffer();              continue;          }          clearInputBuffer();          switch (choice)          {          case 1:              currentUser->displayInfo();              break;          case 2:          {              string oldP, newP;              cout << "请输入原密码: ";              getline(cin >> ws, oldP);              cout << "请输入新密码: ";              getline(cin >> ws, newP);              if (currentUser->changePassword(oldP, newP))              {                  User::saveUsersToFile(users, USER\_FILE);                  cout << "密码修改成功。" << endl;              }              else              {                  cout << "密码修改失败。" << endl;              }          }          break;          case 3:              return 3; // 返回 3，通知 main 调用 pagestore          case 4:          {              double income = currentUser->checkBalance(); // 假设收入就是余额              cout << fixed << setprecision(2);            // 设置输出格式              cout << "当前收入: " << income << endl;              break;          }          case 0:              cout << "退出登录..." << endl;              return 2; // 返回 2 给 main 处理退出登录          default:              cout << "无效选项。" << endl;          }      }  }  int Page::pagemainAdmin(User \*currentUser, vector<User \*> &users)  {      // 管理员菜单实现      int choice = 0;      while (true)      {          printf("\n\n\n--- 管理员菜单 (%s) ---\n", currentUser->getUsername().c\_str());          cout << "1. 查看所有用户信息" << endl;          // 可以添加其他管理员功能          cout << "2. 退出登录" << endl;          cout << "请选择一个选项: ";          if (!(cin >> choice))          {              cout << "无效输入。" << endl;              clearInputBuffer();              continue;          }          clearInputBuffer();          switch (choice)          {          case 1:              User::displayAllUsers(users); // 假设 User 类有这个静态方法              break;          case 2:              cout << "退出登录..." << endl;              return 2; // 返回 2 给 main 处理退出登录          default:              cout << "无效选项。" << endl;          }      }  }  // --- Customer\_page 商城实现 ---  int Customer\_page::pagestore(User \*currentUser, std::vector<User \*> &users, Store &store)  {      int choice = -1;      Customer \*customer = dynamic\_cast<Customer \*>(currentUser); // 用于调用购物车方法      string username\_display = "游客";      if (currentUser)      {          username\_display = currentUser->getUsername();      }      do      {          cout << "\n--- 商城 (用户: " << username\_display << ") ---" << endl;          cout << "1. 显示所有商品" << endl;          cout << "2. 按名称搜索商品" << endl;          cout << "3. 直接购买商品" << endl; // 保持不变          cout << "4. 加入购物车" << endl;          cout << "5. 查看我的购物车" << endl;          cout << "0. 退出商城" << endl;          cout << "请选择: ";          if (!(cin >> choice))          {              cout << "无效输入。" << endl;              Page::clearInputBuffer();              choice = -1;              continue;          }          Page::clearInputBuffer();          switch (choice)          {          case 1:              store.displayAllProducts();              break;          case 2:          {              string searchTerm;              cout << "输入搜索名称: ";              getline(cin >> ws, searchTerm);              vector<Product \*> results = store.searchProductsByName(searchTerm);              if (results.empty())              {                  cout << "未找到匹配商品。" << endl;              }              else              {                  cout << "\n--- 搜索结果 ---" << endl;                  for (const auto \*p : results)                  {                      cout << "-----------------" << endl;                      p->display();                  }                  cout << "-----------------" << endl;              }              break;          }          case 3: // 直接购买商品逻辑 - 现在通过订单系统处理          {              if (!customer)              { // 必须是登录的 Customer                  cout << "请先以消费者身份登录后再购买商品！" << endl;                  break;              }              // --- 开始直接购买逻辑 ---              string productName\_direct;              int quantity\_direct;              cout << "输入要购买的商品名称: ";              getline(cin >> ws, productName\_direct);              Product \*product\_direct = store.findProductByName(productName\_direct);              if (!product\_direct)              {                  cout << "未找到商品 \"" << productName\_direct << "\"！" << endl;                  break;              }              cout << "输入购买数量: ";              if (!(cin >> quantity\_direct) || quantity\_direct <= 0)              {                  cout << "无效购买数量！" << endl;                  Page::clearInputBuffer();                  break;              }              Page::clearInputBuffer();              double totalCost\_direct = product\_direct->getPrice() \* quantity\_direct;              if (product\_direct->getQuantity() < quantity\_direct)              {                  cout << "库存不足！" << endl;                  break;              }              if (customer->checkBalance() < totalCost\_direct)              {                  cout << "余额不足！" << endl;                  break;              }              // 创建订单对象              Order currentOrder(customer->getUsername());              currentOrder.setStatus("PENDING\_VALIDATION");              // 添加商品到订单              if (!currentOrder.addItemFromProduct(\*product\_direct, quantity\_direct))              {                  cout << "添加商品到订单失败！" << endl;                  break;              }              // 计算订单总额              currentOrder.calculateTotalAmount();              // 显示订单信息并请求确认              currentOrder.setStatus("PENDING\_CONFIRMATION");              currentOrder.displaySummary();              cout << "您的当前余额: ¥" << std::fixed << std::setprecision(2) << customer->checkBalance() << endl;              cout << "支付后余额将为: ¥" << (customer->checkBalance() - currentOrder.getTotalAmount()) << endl;              cout << "--------------------" << endl;              char confirmChoice;              cout << "确认支付并完成订单吗? (y/n): ";              cin >> confirmChoice;              clearInputBuffer();              if (tolower(confirmChoice) != 'y')              {                  currentOrder.setStatus("CANCELLED\_BY\_USER");                  cout << "订单已取消。" << endl;                  break;              }              // 提交订单到处理队列              cout << "正在处理订单，请稍候..." << endl;              extern OrderManager g\_orderManager;              auto submittedOrder = g\_orderManager.submitOrderRequest(currentOrder);              cout << "订单已成功提交到处理队列，订单ID: " << submittedOrder->getOrderId() << endl;              cout << "正在等待订单处理，请稍候..." << endl;              // 等待订单处理完成              int waitCounter = 0;              const int MAX\_WAIT\_CYCLES = 100; // 最多等待10秒              while (!submittedOrder->getProcessed() && waitCounter < MAX\_WAIT\_CYCLES)              {                  // 短暂休眠，避免过度消耗CPU                  std::this\_thread::sleep\_for(std::chrono::milliseconds(100));                  waitCounter++;                  // 每秒显示等待状态                  if (waitCounter % 10 == 0)                  {                      cout << "已等待 " << (waitCounter / 10) << " 秒..." << endl;                  }              }              // 检查是否超时              if (waitCounter >= MAX\_WAIT\_CYCLES)              {                  cout << "等待订单处理超时！订单可能仍在处理中。" << endl;                  cout << "您可以稍后查询订单状态。" << endl;              }              else              {                  cout << "订单已处理完成! 状态: " << submittedOrder->getStatus() << endl;                  if (submittedOrder->getStatus() == "COMPLETED" ||                      submittedOrder->getStatus() == "COMPLETED\_WITH\_PAYMENT\_ISSUES")                  {                      cout << "购买成功！已购买 " << quantity\_direct << " 件 \"" << productName\_direct << "\"" << endl;                  }                  else if (submittedOrder->getStatus().find("FAILED") != std::string::npos)                  {                      cout << "订单处理失败! 原因: ";                      if (submittedOrder->getStatus() == "FAILED\_PAYMENT\_ERROR")                          cout << "支付处理时出错" << endl;                      else if (submittedOrder->getStatus() == "FAILED\_INSUFFICIENT\_FUNDS")                          cout << "账户余额不足" << endl;                      else if (submittedOrder->getStatus() == "FAILED\_PRODUCT\_NOT\_FOUND")                          cout << "商品不存在或已下架" << endl;                      else if (submittedOrder->getStatus() == "FAILED\_INSUFFICIENT\_STOCK")                          cout << "商品库存不足" << endl;                      else                          cout << "未知原因，请联系客服" << endl;                  }              }              break;          }          case 4: // 加入购物车          {              if (!customer)              {                  cout << "请先以消费者身份登录后再使用购物车功能！" << endl;                  break;              }              string productName\_cart;              int quantity\_to\_add;              cout << "输入要加入购物车的商品名称: ";              getline(cin >> ws, productName\_cart);              Product \*product\_to\_add = store.findProductByName(productName\_cart);              if (!product\_to\_add)              {                  cout << "未找到商品 \"" << productName\_cart << "\"！" << endl;                  break;              }              cout << "输入数量: ";              if (!(cin >> quantity\_to\_add) || quantity\_to\_add <= 0)              {                  cout << "无效的数量！" << endl;                  Page::clearInputBuffer();                  break;              }              Page::clearInputBuffer();              customer->addToCart(\*product\_to\_add, quantity\_to\_add);              break;          }          case 5: // 查看我的购物车          {              if (!customer)              {                  cout << "请先以消费者身份登录后再查看购物车！" << endl;                  break;              }              customer->viewCart(store, users); // viewCart 现在是 Customer 的成员函数              break;          }          case 0:              cout << "正在退出商城..." << endl;              break;          default:              cout << "无效选项。" << endl;          }      } while (choice != 0);      return 1;  }  // 只替换Seller\_page::pagestore方法，保留文件其他部分不变  // --- Seller\_page 商城实现 ---  int Seller\_page::pagestore(User \*currentUser, std::vector<User \*> &users, Store &store)  {      // 确保当前用户是商家      if (!currentUser || currentUser->getUserType() != "seller")      {          cout << "错误：只有商家才能访问商家管理页面。" << endl;          return 1; // 直接退出      }      string sellerUsername = currentUser->getUsername();      int choice = -1;      do      {          cout << "\n--- 商城管理 (商家: " << sellerUsername << ") ---" << endl;          cout << "1. 显示我的商品" << endl;          cout << "2. 按名称搜索我的商品" << endl;          cout << "3. 添加新商品" << endl;          cout << "4. 修改商品价格" << endl;          cout << "5. 修改商品库存" << endl;          cout << "6. 修改商品折扣" << endl;          cout << "7. 应用分类折扣" << endl;          // cout << "8. 查看余额" << endl;          cout << "0. 退出商城管理" << endl;          cout << "请选择: ";          if (!(cin >> choice))          {              cout << "无效输入。" << endl;              clearInputBuffer();              choice = -1;              continue;          }          clearInputBuffer();          switch (choice)          {          case 1:              store.displaySellerProducts(sellerUsername);              break;          case 2:          {              string searchTerm;              cout << "输入搜索名称: ";              getline(cin >> ws, searchTerm);              vector<Product \*> results = store.searchProductsByName(searchTerm, sellerUsername);              if (results.empty())              {                  cout << "未找到匹配商品。" << endl;              }              else              {                  cout << "\n--- 搜索结果 ---" << endl;                  for (const auto \*p : results)                  {                      cout << "-----------------" << endl;                      p->display();                  }                  cout << "-----------------" << endl;              }              break;          }          case 3:          {              // --- 添加商品逻辑 ---              int typeChoice;              cout << "选择商品类型 (1: 图书, 2: 服装, 3: 食品, 4: 其他类型): ";              if (!(cin >> typeChoice) || typeChoice < 1 || typeChoice > 4)              {                  cout << "无效类型。" << endl;                  clearInputBuffer();                  break;              }              clearInputBuffer();              string name, desc, spec1, spec2;              double price;              int qty;              cout << "输入商品名称: ";              getline(cin >> ws, name);              cout << "输入描述: ";              getline(cin >> ws, desc);              cout << "输入价格: ";              while (!(cin >> price) || price < 0)              {                  cout << "无效价格\n";                  clearInputBuffer();                  cout << "输入价格: ";              }              clearInputBuffer();              cout << "输入库存数量: ";              while (!(cin >> qty) || qty < 0)              {                  cout << "无效数量\n";                  clearInputBuffer();                  cout << "输入库存数量: ";              }              clearInputBuffer();              bool success = false;              if (typeChoice == 1)              { // Book                  cout << "输入作者: ";                  getline(cin >> ws, spec1);                  cout << "输入ISBN: ";                  getline(cin >> ws, spec2);                  success = store.createBook(currentUser, name, desc, price, qty, spec1, spec2);              }              else if (typeChoice == 2)              { // Clothing                  cout << "输入尺码: ";                  getline(cin >> ws, spec1);                  cout << "输入颜色: ";                  getline(cin >> ws, spec2);                  success = store.createClothing(currentUser, name, desc, price, qty, spec1, spec2);              }              else if (typeChoice == 3)              { // Food                  cout << "输入过期日期 (YYYY-MM-DD): ";                  getline(cin >> ws, spec1);                  success = store.createFood(currentUser, name, desc, price, qty, spec1);              }              else if (typeChoice == 4)              { // Generic                  cout << "输入分类标签: ";                  getline(cin >> ws, spec1);                  success = store.createGenericProduct(currentUser, name, desc, price, qty, spec1);              }              else              {                  cout << "无效类型。" << endl;                  break;              }              if (success)                  cout << "商品 \"" << name << "\" 添加成功！" << endl;              else                  cout << "商品添加失败！" << endl;              break;          }          case 4:          { // 修改价格              string name;              double price;              cout << "输入要修改价格的商品名称: ";              getline(cin >> ws, name);              cout << "输入新价格: ";              if (!(cin >> price) || price < 0)              {                  cout << "无效价格\n";                  clearInputBuffer();                  break;              }              clearInputBuffer();              if (store.manageProductPrice(currentUser, name, price))                  cout << "价格修改成功。\n";              else                  cout << "价格修改失败。\n";              break;          }          case 5:          { // 修改库存              string name;              int qty;              cout << "输入要修改库存的商品名称: ";              getline(cin >> ws, name);              cout << "输入新库存: ";              if (!(cin >> qty) || qty < 0)              {                  cout << "无效库存\n";                  clearInputBuffer();                  break;              }              clearInputBuffer();              if (store.manageProductQuantity(currentUser, name, qty))                  cout << "库存修改成功。\n";              else                  cout << "库存修改失败。\n";              break;          }          case 6:          { // 修改折扣              string name;              double discount;              cout << "输入要修改折扣的商品名称: ";              getline(cin >> ws, name);              cout << "输入新折扣率 (0 到 100): ";              if (!(cin >> discount) || discount < 0 || discount > 100)              {                  cout << "无效折扣率\n";                  clearInputBuffer();                  break;              }              clearInputBuffer();              if (store.manageProductDiscount(currentUser, name, discount / 100))                  cout << "折扣修改成功。\n";              else                  cout << "折扣修改失败。\n";              break;          }          case 7:          { // 应用分类折扣              string category;              double discount;              std::vector<std::string> sellerCategories = store.getUniqueCategoriesForSeller(sellerUsername);              if (sellerCategories.empty())              {                  cout << "您当前没有任何商品分类可以应用折扣。" << endl;                  break;              }              cout << "您当前的商品分类有: ";              for (size\_t i = 0; i < sellerCategories.size(); ++i)              {                  cout << "\"" << sellerCategories[i] << "\"";                  if (i < sellerCategories.size() - 1)                  {                      cout << ", ";                  }              }              cout << endl;              cout << "输入要应用折扣的分类名称 (从以上列表选择): ";              getline(cin >> ws, category);              bool isValidCategory = false;              for (const auto &cat : sellerCategories)              {                  if (cat == category)                  {                      isValidCategory = true;                      break;                  }              }              if (!isValidCategory)              {                  cout << "输入的分类 \"" << category << "\" 不是您拥有的有效分类或不存在。" << endl;                  break;              }              cout << "输入折扣率 (0 到 100): ";              if (!(cin >> discount) || discount < 0 || discount > 100)              {                  cout << "无效折扣率\n";                  clearInputBuffer();                  break;              }              clearInputBuffer();              if (store.applyCategoryDiscount(currentUser, category, discount / 100))                  cout << "分类折扣应用成功。\n";              else                  cout << "分类折扣应用失败。\n";              break;          }          // case 8:          // {          //     double balance = currentUser->checkBalance();          //     cout << "当前余额: " << balance << endl;          //     break;          // }          case 0:              cout << "正在退出商城管理..." << endl;              break;          default:              cout << "无效选项。" << endl;          }      } while (choice != 0);      return 1; // 返回1表示正常退出商城，继续主循环  } |

#### Order.h

|  |
| --- |
| #ifndef ORDER\_H  #define ORDER\_H  #include <string>  #include <vector>  #include <ctime>  #include <iomanip>  // For std::setprecision  #include <iostream> // For display methods  // Forward declarations  class Product; // From store.h  class User;    // From user.h  class Store;   // From store.h  class OrderItem  {  public:      std::string productId;      std::string productName;      int quantity;      double priceAtPurchase;      std::string sellerUsername;      OrderItem(std::string pid, std::string pName, int qty, double price, std::string sUsername)          : productId(std::move(pid)), productName(std::move(pName)), quantity(qty),            priceAtPurchase(price), sellerUsername(std::move(sUsername)) {}      void display() const;      std::string toStringForSave() const;  };  class Order  {  private:      std::string orderId;      std::string customerUsername;      std::vector<OrderItem> items;      double totalAmount;      bool isProcessed; // Indicates if the order has been processed      time\_t orderTimestamp;      std::string status; // e.g., "PENDING\_CONFIRMATION", "COMPLETED", "CANCELLED\_INSUFFICIENT\_STOCK", "CANCELLED\_INSUFFICIENT\_FUNDS", "CANCELLED\_BY\_USER"      static std::string generateOrderId();  public:      Order(std::string custUsername);      // 新增默认构造函数      Order() : customerUsername(""),                totalAmount(0.0),                orderTimestamp(std::time(nullptr)), status("UNINITIALIZED"),                isProcessed(false)      {          orderId = "EMPTY-" + std::to\_string(orderTimestamp);      }      bool addItemFromProduct(const Product &product, int quantity); // Adds item and updates total      void calculateTotalAmount();                                   // Recalculates total based on items    // Getters      std::string getOrderId() const { return orderId; }      std::string getCustomerUsername() const { return customerUsername; }      const std::vector<OrderItem> &getItems() const { return items; }      double getTotalAmount() const { return totalAmount; }      std::string getStatus() const { return status; }      time\_t getTimestamp() const { return orderTimestamp; } // Setters      void setStatus(const std::string &newStatus) { status = newStatus; }      void setProcessed(bool processed) { isProcessed = processed; }      bool getProcessed() const { return isProcessed; }      // 移除重复的isProcessed()方法，使用getProcessed()代替      void displaySummary() const;      std::string toStringForSaveHeader() const;      // Persistence (Optional - for saving orders to files)      // bool saveToFile(const std::string& directoryPath) const;      // static Order loadFromFile(const std::string& filePath);  };  #endif // ORDER\_H |

#### Order.cpp

|  |
| --- |
| #include "order.h"  #include "../store/store.h"  #include "../user/user.h" // For Product and User definitions  #include <sstream>  #include <chrono>    // For generating unique ID  #include <fstream>   // For file operations (if saving orders)  #include <algorithm> // For std::remove\_if (if needed for item management)  // --- OrderItem Implementation ---  void OrderItem::display() const  {      std::cout << "    - 商品名称: " << productName << " (ID: " << productId << ")" << std::endl;      std::cout << "      购买数量: " << quantity << ", 商品单价: ¥" << priceAtPurchase                << ", 商品总价: ¥" << (quantity \* priceAtPurchase)                << ", 商家: " << sellerUsername << std::endl;  }  // --- Order Implementation ---  std::string Order::generateOrderId()  {      auto now = std::chrono::system\_clock::now();      auto now\_c = std::chrono::system\_clock::to\_time\_t(now); // Convert to time\_t for tm structure      std::tm now\_tm = \*std::localtime(&now\_c);               // Get local time structure      // Get milliseconds part      auto ms = std::chrono::duration\_cast<std::chrono::milliseconds>(now.time\_since\_epoch()) % 1000;      std::stringstream ss;      ss << "ORD-";      ss << std::put\_time(&now\_tm, "%Y%m%d-%H%M%S");                // Format: YYYYMMDD-HHMMSS      ss << "-" << std::setfill('0') << std::setw(3) << ms.count(); // Append milliseconds, zero-padded      return ss.str();  }  Order::Order(std::string custUsername)      : customerUsername(std::move(custUsername)), totalAmount(0.0), status("PENDING\_CREATION")  {      orderId = generateOrderId();      orderTimestamp = std::time(nullptr);  }  bool Order::addItemFromProduct(const Product &product, int quantity)  {      if (quantity <= 0)          return false;      // Note: Stock check should happen before calling this, or this method could also check      // For now, assumes stock check is done externally before adding to order object      items.emplace\_back(product.getName(), product.getName(), quantity, product.getPrice(), product.getSellerUsername());      calculateTotalAmount(); // Recalculate total every time an item is added      return true;  }  void Order::calculateTotalAmount()  {      totalAmount = 0;      for (const auto &item : items)      {          totalAmount += item.priceAtPurchase \* item.quantity;      }  }  void Order::displaySummary() const  {      std::cout << "\n--- 订单汇总 ---" << std::endl;      std::cout << "订单 ID: " << orderId << std::endl;      std::cout << "买家: " << customerUsername << std::endl;      std::cout << "日期: " << std::asctime(std::localtime(&orderTimestamp)); // ctime adds newline      std::cout << "状态: " << status << std::endl;      std::cout << "下单物品:" << std::endl;      std::cout << std::fixed << std::setprecision(2);      for (const auto &item : items)      {          item.display();      }      std::cout << "---------------------" << std::endl;      std::cout << "总花费: ¥" << totalAmount << std::endl;      std::cout << "---------------------" << std::endl;      std::cout << std::resetiosflags(std::ios\_base::fixed | std::ios\_base::floatfield);  }  // 转为CSV格式字符串  std::string OrderItem::toStringForSave() const  {      std::stringstream ss;      ss << std::fixed << std::setprecision(2);      // Simple CSV-like, ensure no commas in your string fields or use a different delimiter/quoting      ss << productId << "," << productName << "," << quantity << "," << priceAtPurchase << "," << sellerUsername;      return ss.str();  }  std::string Order::toStringForSaveHeader() const  {      std::stringstream ss;      ss << std::fixed << std::setprecision(2);      ss << "OrderID:" << orderId << std::endl;      ss << "CustomerUsername:" << customerUsername << std::endl;      ss << "Timestamp:" << orderTimestamp << std::endl; // Save as raw time\_t      ss << "Status:" << status << std::endl;      ss << "TotalAmount:" << totalAmount << std::endl;      ss << "ItemsCount:" << items.size() << std::endl; // Useful for loading      return ss.str();  }  // Optional: Implement saveToFile and loadFromFile if you want to persist orders  // bool Order::saveToFile(const std::string& directoryPath) const { ... }  // Order Order::loadFromFile(const std::string& filePath) { ... } |

#### Ordermanager.h

|  |
| --- |
| #ifndef ORDER\_MANAGER\_H  #define ORDER\_MANAGER\_H  #include "../order/order.h"  #include "../store/store.h"  #include "../user/user.h"  #include <deque>  #include <string>  #include <vector>  #include <thread>  #include <mutex>  #include <condition\_variable>  #include <atomic>  #include <memory> // 为shared\_ptr添加  class OrderManager  {  private:      std::deque<std::shared\_ptr<Order>> orderQueue;      std::string completedOrdersDirectory;      std::string pendingOrdersFile;      // 多线程支持      std::thread processingThread;      mutable std::mutex queueMutex;      std::condition\_variable queueCondVar;      std::atomic<bool> stopProcessing;      // Helper to save a single order to its own file      bool saveOrderToFile(const Order &order) const;      // 处理线程的主函数      void processingLoop(Store &store, std::vector<User \*> &allUsers);      // 内部处理订单的方法      void processNextOrderInternal(std::shared\_ptr<Order> currentOrder, Store &store, std::vector<User \*> &allUsers);  public:      OrderManager(const std::string &ordersDir);      ~OrderManager();      // 启动和停止处理线程      void startProcessingThread(Store &store, std::vector<User \*> &allUsers);      void stopProcessingThread();      // 提交订单到队列      std::shared\_ptr<Order> submitOrderRequest(const Order &orderRequest);      // 处理队列中的订单      void processNextOrder(Store &store, std::vector<User \*> &allUsers);      void processAllPendingOrders(Store &store, std::vector<User \*> &allUsers);      // 添加这个声明！      size\_t getPendingOrderCount() const;      // 显示待处理订单      void displayPendingOrders() const;  };  #endif // ORDER\_MANAGER\_H |

#### Ordermanager.cpp

|  |
| --- |
| #include "ordermanager.h"  #include "../store/store.h"  #include "../order/order.h"  #include "../user/user.h"  #include <iostream>  #include <fstream>  #include <filesystem>  #include <iomanip>  #include <chrono>  #include <thread>  // 声明外部变量  extern const std::string USER\_FILE;  extern bool startProcessing; // 用于控制订单处理线程的启动状态  using namespace std;  // 构造函数，初始化订单目录  OrderManager::OrderManager(const std::string &ordersDir)      : completedOrdersDirectory(ordersDir), stopProcessing(false)  {      // 确保订单目录存在      if (!std::filesystem::exists(completedOrdersDirectory))      {          try          {              std::filesystem::create\_directories(completedOrdersDirectory);              cout << "已创建订单目录：" << completedOrdersDirectory << endl;          }          catch (const std::filesystem::filesystem\_error &e)          {              cerr << "创建订单目录失败: " << e.what() << endl;          }      }      // 设置待处理订单文件路径      pendingOrdersFile = completedOrdersDirectory + "/pending\_orders.txt";  }  // 析构函数 - 停止处理线程  OrderManager::~OrderManager()  {      stopProcessingThread();  }  // 启动处理线程  void OrderManager::startProcessingThread(Store &store, std::vector<User \*> &allUsers)  {      // 如果线程已经在运行，先停止      stopProcessingThread();      // 重置停止标志      stopProcessing = false;      // 启动处理线程      processingThread = std::thread(&OrderManager::processingLoop, this, std::ref(store), std::ref(allUsers));      cout << "订单处理线程已启动" << endl;  }  // 停止处理线程  void OrderManager::stopProcessingThread()  {      if (processingThread.joinable())      {          {              std::lock\_guard<std::mutex> lock(queueMutex);              stopProcessing = true; // 设置停止标志          }          // 通知条件变量，唤醒等待的线程          queueCondVar.notify\_one();          // 等待线程结束          processingThread.join();          cout << "订单处理线程已停止" << endl;      }  }  // 处理线程的主循环函数  void OrderManager::processingLoop(Store &store, std::vector<User \*> &allUsers)  {      cout << "订单处理线程开始运行..." << endl;      while (!stopProcessing)      {          std::shared\_ptr<Order> orderToProcess;          bool hasOrder = false;          {              // 获取队列互斥锁              std::unique\_lock<std::mutex> lock(queueMutex);              // 等待条件变量：直到队列非空或收到停止信号              queueCondVar.wait(lock, [this]                                { return !orderQueue.empty() || stopProcessing; });              // 如果收到停止信号且队列为空，则退出              if (stopProcessing && orderQueue.empty())                  break;              // 如果队列非空，取出队首订单              if (!orderQueue.empty())              {                  orderToProcess = orderQueue.front();                  orderQueue.pop\_front();                  hasOrder = true;              }          }          // 如果有订单需要处理，则处理它          if (hasOrder)          {              cout << "\n线程正在处理订单 ID: " << orderToProcess->getOrderId()                   << "，客户: " << orderToProcess->getCustomerUsername() << endl;              // 处理订单              processNextOrderInternal(orderToProcess, store, allUsers);          }      }      cout << "订单处理线程已终止" << endl;  }  // 内部处理订单的方法（被处理线程调用）  void OrderManager::processNextOrderInternal(std::shared\_ptr<Order> currentOrder, Store &store, std::vector<User \*> &allUsers)  {      cout << "正在处理订单 ID: " << currentOrder->getOrderId()           << "，客户: " << currentOrder->getCustomerUsername() << endl;      currentOrder->setStatus("PROCESSING");      // 找到订单对应的客户      Customer \*customer = dynamic\_cast<Customer \*>(User::findUser(allUsers, currentOrder->getCustomerUsername()));      if (!customer)      {          cerr << "错误: 找不到订单对应的客户 " << currentOrder->getCustomerUsername() << endl;          currentOrder->setStatus("FAILED\_CUSTOMER\_NOT\_FOUND");          saveOrderToFile(\*currentOrder);   // 注意这里需要解引用          currentOrder->setProcessed(true); // 标记处理完成          return;      }      // 第一阶段：重新验证商品（库存可能在队列等待期间发生变化）      for (const auto &item : currentOrder->getItems())      {          Product \*product = store.findProductByName(item.productId);          if (!product)          {              cerr << "错误: 商品 \"" << item.productName << "\" 不存在或已下架。订单取消。" << endl;              currentOrder->setStatus("FAILED\_PRODUCT\_NOT\_FOUND");              saveOrderToFile(\*currentOrder);              currentOrder->setProcessed(true);              return;          }          if (product->getQuantity() < item.quantity)          {              cerr << "错误: 商品 \"" << item.productName << "\" 库存不足 (需要 "                   << item.quantity << ", 现有 " << product->getQuantity() << ")。订单取消。" << endl;              currentOrder->setStatus("FAILED\_INSUFFICIENT\_STOCK");              saveOrderToFile(\*currentOrder);              currentOrder->setProcessed(true);              return;          }      }      // 第二阶段：验证客户余额      if (customer->checkBalance() < currentOrder->getTotalAmount())      {          cerr << "错误: 客户 " << customer->getUsername() << " 余额不足。订单取消。" << endl;          currentOrder->setStatus("FAILED\_INSUFFICIENT\_FUNDS");          saveOrderToFile(\*currentOrder);          currentOrder->setProcessed(true);          return;      }      // 第三阶段：处理支付和更新库存      if (!customer->withdraw(currentOrder->getTotalAmount()))      {          cerr << "严重错误: 客户 " << customer->getUsername()               << " 扣款失败，尽管余额检查已通过。订单: " << currentOrder->getOrderId() << endl;          currentOrder->setStatus("FAILED\_PAYMENT\_ERROR");          saveOrderToFile(\*currentOrder);          currentOrder->setProcessed(true);          return;      }      bool allSellersPaid = true;      for (const auto &item : currentOrder->getItems())      {          Product \*product = store.findProductByName(item.productId);          // 更新商品库存          product->setQuantity(product->getQuantity() - item.quantity);          // 向卖家转账          User \*seller = User::findUser(allUsers, item.sellerUsername);          if (seller)          {              double itemAmount = item.priceAtPurchase \* item.quantity;              if (!seller->deposit(itemAmount))              {                  cerr << "警告: 向商家 " << item.sellerUsername                       << " 支付商品 " << item.productName                       << " 的款项 ¥" << fixed << setprecision(2) << itemAmount                       << " 失败。订单 ID: " << currentOrder->getOrderId() << endl;                  allSellersPaid = false;              }              else              {                  cout << "已向商家 " << item.sellerUsername                       << " 支付 ¥" << fixed << setprecision(2) << itemAmount << endl;              }          }          else if (!item.sellerUsername.empty())          {              cerr << "警告: 找不到商品 \"" << item.productName                   << "\" 的商家 \"" << item.sellerUsername                   << "\"。订单 ID: " << currentOrder->getOrderId() << endl;              allSellersPaid = false;          }      }      // 保存所有更改（用户和商店）      User::saveUsersToFile(allUsers, USER\_FILE);      store.saveAllProducts();      // 设置最终订单状态      if (allSellersPaid)      {          currentOrder->setStatus("COMPLETED");      }      else      {          currentOrder->setStatus("COMPLETED\_WITH\_PAYMENT\_ISSUES");      }      cout << "订单 ID: " << currentOrder->getOrderId()           << " 处理完成。最终状态: " << currentOrder->getStatus() << endl;      currentOrder->displaySummary();      saveOrderToFile(\*currentOrder);      // 标记订单已处理完成（关键步骤！）      currentOrder->setProcessed(true);  }  // 提交订单到处理队列 (线程安全版本)  std::shared\_ptr<Order> OrderManager::submitOrderRequest(const Order &orderRequest)  {      // 创建订单的共享指针副本      auto orderPtr = std::make\_shared<Order>(orderRequest);      // 设置订单状态为等待队列处理      orderPtr->setStatus("PENDING\_IN\_QUEUE");      orderPtr->setProcessed(false);      // 锁定队列      std::lock\_guard<std::mutex> lock(queueMutex);      // 将订单加入队列      orderQueue.push\_back(orderPtr);      // 通知处理线程有新订单      queueCondVar.notify\_one();      cout << "订单 ID: " << orderPtr->getOrderId()           << " 已提交到处理队列，当前队列长度: " << orderQueue.size() << endl;      // 返回订单共享指针      return orderPtr;  }  // 处理队列中的一个订单 (公共方法版，委托给内部方法)  void OrderManager::processNextOrder(Store &store, std::vector<User \*> &allUsers)  {      std::shared\_ptr<Order> orderToProcess;      bool hasOrder = false;      {          std::lock\_guard<std::mutex> lock(queueMutex);          if (!orderQueue.empty())          {              orderToProcess = orderQueue.front();              orderQueue.pop\_front();              hasOrder = true;          }      }      if (hasOrder)      {          processNextOrderInternal(orderToProcess, store, allUsers);      }  }  // 处理所有待处理订单 (线程安全版本)  void OrderManager::processAllPendingOrders(Store &store, std::vector<User \*> &allUsers)  {      std::vector<std::shared\_ptr<Order>> ordersToProcess;      // 首先，从队列中安全地取出所有订单      {          std::lock\_guard<std::mutex> lock(queueMutex);          while (!orderQueue.empty())          {              ordersToProcess.push\_back(orderQueue.front());              orderQueue.pop\_front();          }      }      // 如果没有订单，直接返回      if (ordersToProcess.empty())      {          cout << "当前没有待处理订单。" << endl;          return;      }      cout << "\n--- 开始处理 " << ordersToProcess.size() << " 个待处理订单 ---" << endl;      // 处理获取的订单      for (auto &order : ordersToProcess)      {          processNextOrderInternal(order, store, allUsers);      }      cout << "--- 成功处理 " << ordersToProcess.size() << " 个订单 ---" << endl;  }  // 获取待处理订单数量 (线程安全版本)  size\_t OrderManager::getPendingOrderCount() const  {      std::lock\_guard<std::mutex> lock(queueMutex);      return orderQueue.size();  }  // 显示待处理订单 (线程安全版本)  void OrderManager::displayPendingOrders() const  {      std::lock\_guard<std::mutex> lock(queueMutex);      if (orderQueue.empty())      {          cout << "当前队列中没有待处理订单。" << endl;          return;      }      cout << "\n--- 待处理订单队列 ---" << endl;      int i = 1;      for (const auto &order : orderQueue)      {          cout << i++ << ". 订单 ID: " << order->getOrderId()               << ", 客户: " << order->getCustomerUsername()               << ", 金额: ¥" << fixed << setprecision(2) << order->getTotalAmount()               << ", 状态: " << order->getStatus() << endl;      }      cout << "-----------------------------" << endl;  }  // 将订单保存到文件  bool OrderManager::saveOrderToFile(const Order &order) const  {      if (completedOrdersDirectory.empty())      {          cerr << "错误: 未设置订单目录。" << endl;          return false;      }      // 创建基于订单ID的文件名      std::string filePath = completedOrdersDirectory + "/" + order.getOrderId() + ".txt";      std::ofstream outFile(filePath);      if (!outFile)      {          cerr << "错误: 无法打开文件保存订单: " << filePath << endl;          return false;      }      // 保存订单头部信息      outFile << order.toStringForSaveHeader();      // 保存订单项      for (const auto &item : order.getItems())      {          outFile << item.toStringForSave() << endl;      }      outFile.close();      cout << "订单 " << order.getOrderId() << " 已保存到 " << filePath << endl;      return true;  } |

### imgUI

#### server\_main.cpp

|  |
| --- |
| #include <iostream>  #include <string>  #include <signal.h>  #include "network/server.h"  #include <windows.h>  // 全局服务器指针，用于信号处理  NetworkServer \*g\_server = nullptr;  // 信号处理函数  void signalHandler(int signal)  {      std::cout << "\n收到退出信号，正在关闭服务器..." << std::endl;      if (g\_server)      {          g\_server->stop();      }  }  int main()  {      // 设置控制台输出编码为UTF-8  #ifdef \_WIN32      SetConsoleOutputCP(65001);      SetConsoleCP(65001);  #endif      std::cout << "=== 电子商城服务端 ===" << std::endl;      // 设置信号处理      signal(SIGINT, signalHandler);      signal(SIGTERM, signalHandler);      // 获取服务器配置      int port = 8888;      std::cout << "服务器将在端口 " << port << " 上启动" << std::endl;      // 创建服务器      NetworkServer server(port);      g\_server = &server;      // 启动服务器      if (!server.start())      {          std::cerr << "服务器启动失败" << std::endl;          return 1;      }      std::cout << "服务器已启动，正在监听端口 " << port << std::endl;      std::cout << "按 Ctrl+C 停止服务器" << std::endl;      // 服务器主循环      while (server.isServerRunning())      {          // 服务器在后台运行，这里可以添加管理命令          std::this\_thread::sleep\_for(std::chrono::milliseconds(100));      }      std::cout << "服务器已停止" << std::endl;      return 0;  } |

#### client\_main.cpp

|  |
| --- |
| #include <iostream>  #include <string>  #include "client/client\_ui.h"  #include <windows.h>  int main()  {      // 设置控制台输出编码为UTF-8  #ifdef \_WIN32      SetConsoleOutputCP(65001);      SetConsoleCP(65001);  #endif      std::cout << "=== 电子商城客户端 ===" << std::endl;      // 创建客户端UI      ClientUI clientUI(800, 600);      // 初始化UI      if (!clientUI.init())      {          std::cerr << "客户端UI初始化失败，程序退出" << std::endl;          return 1;      }      // 连接到服务器      std::string serverAddress = "127.0.0.1";      int serverPort = 8888;      std::cout << "正在连接到服务器 " << serverAddress << ":" << serverPort << "..." << std::endl;      if (!clientUI.connectToServer(serverAddress, serverPort))      {          std::cerr << "无法连接到服务器，请确保服务器正在运行" << std::endl;          std::cout << "按任意键退出..." << std::endl;          std::cin.get();          return 1;      }      std::cout << "已成功连接到服务器" << std::endl;      // 启动客户端主循环      clientUI.mainLoop();      // 断开连接      clientUI.disconnectFromServer();      std::cout << "客户端已退出" << std::endl;      return 0;  } |

#### user.h

|  |
| --- |
| #ifndef USER\_H  #define USER\_H  #include <iostream>  #include <string>  #include <vector>  #include <fstream>  #include <limits>  #include "../store/store.h" // 需要 Product 和 Store 的完整定义  #include "../order/order.h"  #include "../order/ordermanager.h"  #include <limits>  #include <sstream>  #include <filesystem> // 用于文件操作  #include <algorithm>  // for std::remove\_if  #include <iomanip>    // for std::fixed, std::setprecision  #include <thread>  // 前向声明  class Product;  class Store;  // 购物车中的商品项 (现在作为 Customer 类的内部结构或辅助结构)  struct CartItem  {      std::string productId; // 商品名称或唯一ID      std::string productName;      int quantity;      double priceAtAddition; // 添加到购物车时的价格      std::string sellerUsername;      CartItem(std::string pid = "", std::string pName = "", int qty = 0, double price = 0.0, std::string sName = "")          : productId(pid), productName(pName), quantity(qty), priceAtAddition(price), sellerUsername(sName) {}  };  class User  {  protected:      std::string username;      std::string password;      std::string userType;      double balance;      static void clearInputBuffer()      {          std::cin.clear();          std::cin.ignore(std::numeric\_limits<std::streamsize>::max(), '\n');      }  public:      User(std::string username, std::string password, double balance = 0.0,           bool is\_customer = false, bool is\_seller = false, bool is\_admin = false);      User();      virtual ~User() = default;      virtual std::string getUserType() const = 0;      void displayInfo(int index = -1) const;      bool changePassword(const std::string &oldPassword, const std::string &newPassword);      bool deposit(double amount);      bool withdraw(double amount);      double checkBalance() const;      static User \*registerUser(std::vector<User \*> &users);      static User \*userLogin(const std::vector<User \*> &users);      static bool saveUsersToFile(const std::vector<User \*> &users, const std::string &filename);      static std::vector<User \*> loadUsersFromFile(const std::string &filename);      static User \*findUser(const std::vector<User \*> &users, const std::string &username);      static void displayAllUsers(const std::vector<User \*> &users);      static bool isUsernameExists(const std::vector<User \*> &users, const std::string &username);      std::string getUsername() const;      std::string getPassword() const;      void setUsername(const std::string &newUsername);      void setPassword(const std::string &newPassword);      void setBalance(double newBalance);      static User \*login(const std::vector<User \*> &users, const std::string &username, const std::string &password);  };  // 消费者类  class Customer : public User  {  private:      std::string cartDirectoryPath; // 存储购物车文件的目录路径      // 购物车文件操作的辅助方法      std::string getCartFilePath() const;  public:      Customer(std::string username, std::string password, const std::string &cartDir, double balance = 0.0);      Customer(const std::string &cartDir); // 默认构造函数      std::string getUserType() const override { return "customer"; }      std::vector<CartItem> shoppingCartItems; // 直接在 Customer 中存储购物车项      void loadCartFromFile();      bool saveCartToFile() const;      // 购物车管理方法      bool addToCart(const Product &product, int quantity);      bool removeCartItem(const std::string &productName);      bool updateCartItemQuantity(const std::string &productName, int newQuantity, Store &store); // 需要 store 检查库存      void viewCart(Store &store, std::vector<User \*> &allUsers);                                 // 查看并管理购物车，结算也在这里处理      bool isCartEmpty() const;      void clearCartAndFile(); // 结算后清空购物车及文件  };  // 商家类  class Seller : public User  {  public:      Seller(std::string username, std::string password, double balance = 0.0);      Seller();      std::string getUserType() const override { return "seller"; }  };  // 管理员类  class Admin : public User  {  public:      Admin(std::string username, std::string password, double balance = 0.0);      Admin();      std::string getUserType() const override { return "admin"; }  };  #endif // USER\_H |

#### user.cpp

|  |
| --- |
| #include "user.h"  using namespace std;  const std::string CART\_DIR = "./server\_data/carts"; // 购物车文件存储目录  // 在网络服务端版本中注释掉这些全局变量声明，它们只在UI版本中使用  // extern const string CART\_FILE; // 用于 Customer 构造函数  // extern const string USER\_FILE; // 用于保存用户信息  // User 构造函数实现  User::User(std::string u\_name, std::string pwd, double bal, bool is\_customer, bool is\_seller, bool is\_admin)      : username(u\_name), password(pwd), balance(bal)  {      // userType 在派生类中设置  }  User::User() : username(""), password(""), balance(0.0), userType("unknown") {}  // --- Customer 类购物车相关方法实现 ---  std::string Customer::getCartFilePath() const  {      if (this->username.empty())      { // 防止用户名为空时生成无效路径          // cerr << "警告: Customer 用户名为空，无法生成购物车路径。" << endl;          return ""; // 或者抛出异常，或者返回一个默认的无效路径      }      return this->cartDirectoryPath + "/" + this->username + "\_cart.txt";  }  void Customer::loadCartFromFile()  {      shoppingCartItems.clear();      std::string cartPath = getCartFilePath();      if (cartPath.empty())          return; // 用户名为空，无法加载      std::ifstream file(cartPath);      if (!file.is\_open())      {          return; // 文件不存在或无法打开是正常情况（购物车为空）      }      std::string line;      while (getline(file, line))      {          std::stringstream ss(line);          std::string productId, productName, sellerUsernameStr;          int quantity;          double priceAtAddition;          if (getline(ss, productId, ',') &&              getline(ss, productName, ',') &&              (ss >> quantity) &&              (ss.ignore(1) && ss >> priceAtAddition) &&              (ss.ignore(1) && getline(ss, sellerUsernameStr)))          {              shoppingCartItems.emplace\_back(productId, productName, quantity, priceAtAddition, sellerUsernameStr);          }          else          {              std::cerr << "警告: 购物车文件 " << cartPath << " 中存在格式错误的行: " << line << std::endl;          }      }      file.close();  }  bool Customer::saveCartToFile() const  {      std::string cartPath = getCartFilePath();      if (cartPath.empty())      {          std::cerr << "错误: Customer 用户名为空，无法保存购物车。" << std::endl;          return false;      }      std::ofstream file(cartPath); // 会覆盖旧文件      if (!file.is\_open())      {          std::cerr << "错误: 无法打开购物车文件进行写入: " << cartPath << std::endl;          return false;      }      for (const auto &item : shoppingCartItems)      {          file << item.productId << ","               << item.productName << ","               << item.quantity << ","               << item.priceAtAddition << ","               << item.sellerUsername << std::endl;      }      file.close();      return true;  }  // Customer 构造函数实现  Customer::Customer(std::string uname, std::string pwd, const std::string &cartDir, double bal)      : User(uname, pwd, bal, true, false, false), cartDirectoryPath(cartDir)  {      this->userType = "customer";      if (!this->username.empty())      { // 只有当用户名有效时才加载购物车          loadCartFromFile();      }  }  // 默认构造函数  Customer::Customer(const std::string &cartDir)      : User(), cartDirectoryPath(cartDir)  {      this->userType = "customer";      // 注意: 此时 username 为空，loadCartFromFile 不会执行。      // 如果后续通过 setUsername 设置了用户名，需要一种机制来加载对应的购物车。      // 或者，强制 Customer 总是在有用户名的情况下创建。  }  bool Customer::addToCart(const Product &product, int quantity)  {      if (this->username.empty())      {          cout << "错误：无法为匿名用户添加购物车。" << endl;          return false;      }      if (quantity <= 0)      {          cout << "添加数量必须大于0。" << endl;          return false;      }      bool found = false;      for (auto &item : shoppingCartItems)      {          if (item.productId == product.getName())          { // 假设商品名称是唯一ID              item.quantity += quantity;              // 可选：更新 priceAtAddition 为最新价格或保持不变              found = true;              break;          }      }      if (!found)      {          shoppingCartItems.emplace\_back(product.getName(), product.getName(), quantity, product.getPrice(), product.getSellerUsername());      }      cout << "\"" << product.getName() << "\" 已成功加入购物车。" << endl;      return saveCartToFile();  }  bool Customer::removeCartItem(const std::string &productName)  {      if (this->username.empty())          return false;      auto initial\_size = shoppingCartItems.size();      shoppingCartItems.erase(std::remove\_if(shoppingCartItems.begin(), shoppingCartItems.end(),                                             [&](const CartItem &item)                                             { return item.productName == productName; }),                              shoppingCartItems.end());      if (shoppingCartItems.size() < initial\_size)      {          cout << "商品 \"" << productName << "\" 已从购物车移除。" << endl;          return saveCartToFile();      }      else      {          cout << "未在购物车中找到商品: " << productName << endl;          return false;      }  }  bool Customer::updateCartItemQuantity(const std::string &productName, int newQuantity, Store &store)  {      if (this->username.empty())          return false;      if (newQuantity < 0)      {          cout << "数量不能为负。" << endl;          return false;      }      bool updated = false;      for (auto &item : shoppingCartItems)      {          if (item.productName == productName)          {              Product \*p\_info = store.findProductByName(productName); // 检查最新库存              if (p\_info && newQuantity > p\_info->getQuantity())              {                  cout << "库存不足！无法将购物车中 \"" << productName << "\" 的数量修改为 " << newQuantity                       << "。当前库存: " << p\_info->getQuantity() << endl;                  return false;              }              item.quantity = newQuantity;              updated = true;              break;          }      }      if (!updated)      {          cout << "未在购物车中找到商品: " << productName << endl;          return false;      }      // 移除数量为0的商品      shoppingCartItems.erase(std::remove\_if(shoppingCartItems.begin(), shoppingCartItems.end(),                                             [](const CartItem &item)                                             { return item.quantity <= 0; }),                              shoppingCartItems.end());      cout << "购物车商品 \"" << productName << "\" 数量已更新。" << endl;      return saveCartToFile();  }  void Customer::clearCartAndFile()  {      if (this->username.empty())          return;      shoppingCartItems.clear();      std::string cartPath = getCartFilePath();      if (!cartPath.empty() && std::filesystem::exists(cartPath))      {          try          {              std::filesystem::remove(cartPath);          }          catch (const std::filesystem::filesystem\_error &err)          {              std::cerr << "错误: 删除购物车文件失败: " << err.what() << std::endl;          }      }  }  bool Customer::isCartEmpty() const  {      return shoppingCartItems.empty();  }  void Customer::viewCart(Store &store, std::vector<User \*> &allUsers)  {      if (this->username.empty())      {          cout << "错误：匿名用户无法查看购物车。" << endl;          return;      }      // 确保购物车是最新的（如果其他地方可能修改文件但未更新内存）      // loadCartFromFile(); // 通常在每次操作后保存，所以这里可能不需要，除非有外部修改购物车的可能      if (shoppingCartItems.empty())      {          cout << "\n您的购物车是空的。" << endl;          return;      }      double totalCartPriceBasedOnCurrent = 0;      cout << "\n--- 我的购物车 ---" << endl;      cout << std::fixed << std::setprecision(2); // 设置输出格式      for (size\_t i = 0; i < shoppingCartItems.size(); ++i)      {          const auto &item = shoppingCartItems[i];          Product \*p\_info = store.findProductByName(item.productId);          double currentItemPrice = p\_info ? p\_info->getPrice() : item.priceAtAddition;          cout << i + 1 << ". 商品: " << item.productName               << " | 数量: " << item.quantity               << " | 当前单价: ¥" << currentItemPrice               << " | 小计: ¥" << (item.quantity \* currentItemPrice) << endl;          if (p\_info)          {              if (p\_info->getQuantity() < item.quantity)              {                  cout << "   注意: " << item.productName << " 当前库存(" << p\_info->getQuantity() << ")不足购物车数量(" << item.quantity << ")!" << endl;              }              if (p\_info->getPrice() != item.priceAtAddition)              {                  cout << "   提示: " << item.productName << " 加入时价格为 ¥" << item.priceAtAddition << ", 当前价格已变为 ¥" << p\_info->getPrice() << endl;              }          }          else          {              cout << "   警告: 商品 " << item.productName << " 可能已从商店下架!" << endl;          }          totalCartPriceBasedOnCurrent += item.quantity \* currentItemPrice;      }      cout << "--------------------" << endl;      cout << "购物车总计 (按当前价格): ¥" << totalCartPriceBasedOnCurrent << endl;      cout << "--------------------" << endl;      int cartChoice = -1;      do      {          cout << "\n购物车操作:" << endl;          cout << "1. 修改商品数量" << endl;          cout << "2. 移除商品" << endl;          cout << "3. 生成订单并结算" << endl;          cout << "0. 返回商城" << endl;          cout << "请选择: ";          if (!(cin >> cartChoice))          {              cout << "无效输入。" << endl;              User::clearInputBuffer();              cartChoice = -1;              continue;          }          User::clearInputBuffer();          switch (cartChoice)          {          case 1:          {              string nameToUpdate;              int newQty;              cout << "输入要修改数量的商品名称: ";              getline(cin >> ws, nameToUpdate);              cout << "输入新的数量 (输入0将移除该商品): ";              if (!(cin >> newQty))              {                  cout << "无效数量输入。" << endl;                  User::clearInputBuffer();                  break;              }              User::clearInputBuffer();              if (updateCartItemQuantity(nameToUpdate, newQty, store))              {                  // 成功后重新加载并显示购物车 (递归调用或重新加载数据)                  // 为了避免无限递归，这里直接返回，让外层循环重新调用 viewCart                  cout << "购物车已更新。" << endl;              }              return viewCart(store, allUsers); // 重新显示更新后的购物车          }          case 2:          {              string nameToRemove;              cout << "输入要移除的商品名称: ";              getline(cin >> ws, nameToRemove);              if (removeCartItem(nameToRemove))              {                  cout << "购物车已更新。" << endl;              }              return viewCart(store, allUsers); // 重新显示更新后的购物车          }          case 3: // Generate Order and Checkout          {              if (shoppingCartItems.empty())              {                  cout << "购物车是空的，无法生成订单。" << endl;                  break;              }              Order currentOrder(this->getUsername()); // Create a new order object              currentOrder.setStatus("PENDING\_VALIDATION");              // 第一阶段：验证商品并构建订单（预检查）              bool preCheckOk = true;              for (const auto &cart\_item : shoppingCartItems)              {                  Product \*product = store.findProductByName(cart\_item.productId);                  if (!product)                  {                      cout << "错误: 购物车商品 \"" << cart\_item.productName << "\" 已不存在。" << endl;                      currentOrder.setStatus("CANCELLED\_PRODUCT\_NOT\_FOUND");                      preCheckOk = false;                      break;                  }                  if (product->getQuantity() < cart\_item.quantity)                  {                      cout << "错误: 购物车商品 \"" << cart\_item.productName << "\" 库存不足 (需要 "                           << cart\_item.quantity << ", 现有 " << product->getQuantity() << ")。" << endl;                      currentOrder.setStatus("CANCELLED\_INSUFFICIENT\_STOCK");                      preCheckOk = false;                      break;                  }                  currentOrder.addItemFromProduct(\*product, cart\_item.quantity);              }              if (!preCheckOk)              {                  currentOrder.displaySummary(); // Display order with cancelled status                  cout << "订单无法继续，请调整购物车后重试。" << endl;                  break; // Back to cart menu              }              // 第二阶段：验证客户余额（只是初步检查，队列处理时还会再次检查）              if (this->checkBalance() < currentOrder.getTotalAmount())              {                  cout << "错误: 您的余额不足以支付此订单！" << endl;                  currentOrder.setStatus("CANCELLED\_INSUFFICIENT\_FUNDS");                  // currentOrder.displaySummary();                  cout << "您的余额:   ¥" << this->checkBalance() << endl;                  break; // Back to cart menu              }              // 第三阶段：显示订单并请求确认              currentOrder.setStatus("PENDING\_CONFIRMATION");              currentOrder.displaySummary();              cout << "您的当前余额: ¥" << std::fixed << std::setprecision(2) << this->checkBalance() << endl;              cout << "支付后余额将为: ¥" << (this->checkBalance() - currentOrder.getTotalAmount()) << endl;              cout << "--------------------" << endl;              char confirmChoice;              cout << "确认支付并完成订单吗? (y/n): ";              cin >> confirmChoice;              User::clearInputBuffer();              if (tolower(confirmChoice) != 'y')              {                  currentOrder.setStatus("CANCELLED\_BY\_USER");                  currentOrder.displaySummary();                  cout << "订单已取消。" << endl;                  break; // Back to cart menu              }              // Stage 4: Process payment and update inventory (Actual "lock" and transaction)              cout << "\n正在处理订单，请稍候..." << endl;              currentOrder.setStatus("PROCESSING"); // 在网络服务端版本中，这部分逻辑由服务端处理，客户端不直接处理订单              // 这里只是一个占位符，实际的订单处理在服务端的 NetworkServer::handleOrderCreate 中              cout << "订单信息已准备完成，等待提交到服务端处理..." << endl; // 在网络版本中，订单处理逻辑在服务端完成              // 这里只显示订单准备完成的消息              cout << "订单信息准备完成，可以提交到服务端。" << endl;              cout << std::resetiosflags(std::ios\_base::fixed | std::ios\_base::floatfield);              return; // 退出viewCart，返回Customer\_page商店菜单              // if (!this->withdraw(currentOrder.getTotalAmount()))              // {              //     currentOrder.setStatus("FAILED\_PAYMENT\_WITHDRAWAL");              //     currentOrder.displaySummary();              //     cout << "错误: 支付失败。请检查余额或联系客服。" << endl;              //     // Potentially log this critical failure              //     break;              // }              // for (const auto &order\_item : currentOrder.getItems())              // {              //     Product \*product = store.findProductByName(order\_item.productId); // Re-fetch, though should be same              //     if (product)              //     { // Should always be true due to pre-checks              //         product->setQuantity(product->getQuantity() - order\_item.quantity);              //         User \*seller = User::findUser(allUsers, order\_item.sellerUsername);              //         if (seller)              //         {              //             if (!seller->deposit(order\_item.priceAtPurchase \* order\_item.quantity))              //             {              //                 cerr << "警告: 向商家 " << seller->getUsername() << " 转账失败 (商品: "              //                      << order\_item.productName << "). 订单号: " << currentOrder.getOrderId() << endl;              //                 // Log this for reconciliation              //             }              //         }              //         else if (!order\_item.sellerUsername.empty())              //         {              //             cerr << "警告: 未找到商品 \"" << order\_item.productName << "\" 的商家 \""              //                  << order\_item.sellerUsername << "\". 订单号: " << currentOrder.getOrderId() << endl;              //         }              //         else              //         {              //             cerr << "警告: 商品 \"" << order\_item.productName << "\" 没有商家信息. 订单号: " << currentOrder.getOrderId() << endl;              //         }              //     }              // }              // User::saveUsersToFile(allUsers, USER\_FILE);              // store.saveAllProducts(); // Or save only affected sellers              // currentOrder.setStatus("COMPLETED");              // cout << "\n订单处理完成！" << endl;              // currentOrder.displaySummary(); // Display final completed order              // // Optional: currentOrder.saveToFile(ORDER\_DIRECTORY\_BASE + "/" + this->getUsername());              // clearCartAndFile(); // Clear cart after successful order              // cout << std::resetiosflags(std::ios\_base::fixed | std::ios\_base::floatfield);              // return; // Exit viewCart, back to Customer\_page store menu          }          case 0:              // 返回商城，什么也不做，循环会结束              break;          default:              cout << "无效选项。" << endl;          }      } while (cartChoice != 0);      cout << std::resetiosflags(std::ios\_base::fixed | std::ios\_base::floatfield); // 重置输出格式  }  // Seller 构造函数实现  Seller::Seller(std::string uname, std::string pwd, double bal)      : User(uname, pwd, bal, false, true, false)  {      this->userType = "seller";  }  Seller::Seller() : User() { this->userType = "seller"; }  // Admin 构造函数实现  Admin::Admin(std::string uname, std::string pwd, double bal)      : User(uname, pwd, bal, false, false, true)  {      this->userType = "admin";  }  Admin::Admin() : User() { this->userType = "admin"; }  // --- 静态方法：用户注册实现 ---  User \*User::registerUser(std::vector<User \*> &users)  {      string newUsername, newPassword;      double initialBalance = 0.0;      int userTypeChoice;      cout << "请输入新用户名: ";      getline(cin >> ws, newUsername);      if (isUsernameExists(users, newUsername))      {          cout << "用户名已存在！" << endl;          return nullptr;      }      cout << "请输入密码: ";      getline(cin >> ws, newPassword);      cout << "选择用户类型 (1: 消费者, 2: 商家): ";      if (!(cin >> userTypeChoice) || (userTypeChoice != 1 && userTypeChoice != 2))      {          cout << "无效的用户类型选择。" << endl;          clearInputBuffer();          return nullptr;      }      clearInputBuffer();      User \*newUser = nullptr;      if (userTypeChoice == 1)      { // Customer          // 在服务端版本中使用默认的购物车目录          const std::string DEFAULT\_CART\_DIR = CART\_DIR;          newUser = new Customer(newUsername, newPassword, DEFAULT\_CART\_DIR, initialBalance);      }      else      { // Seller          newUser = new Seller(newUsername, newPassword, initialBalance);      }      if (newUser)      {          users.push\_back(newUser);          cout << (userTypeChoice == 1 ? "消费者" : "商家") << "用户 \"" << newUsername << "\" 注册成功！" << endl;      }      return newUser;  }  // 从文件加载用户信息  std::vector<User \*> User::loadUsersFromFile(const std::string &filename)  {      std::vector<User \*> loadedUsers;      std::ifstream file(filename);      if (!file.is\_open())      {          return loadedUsers;      }      std::string line;      while (getline(file, line))      {          std::stringstream ss(line);          std::string uname, pwd, type;          double bal;          if (              getline(ss, uname, ',') &&              getline(ss, pwd, ',') &&              getline(ss, type, ',') &&              (ss >> bal))          {              if (type == "customer")              {                  // 在服务端版本中使用默认的购物车目录                  const std::string DEFAULT\_CART\_DIR = CART\_DIR;                  loadedUsers.push\_back(new Customer(uname, pwd, DEFAULT\_CART\_DIR, bal));              }              else if (type == "seller")              {                  loadedUsers.push\_back(new Seller(uname, pwd, bal));              }              else if (type == "admin")              {                  loadedUsers.push\_back(new Admin(uname, pwd, bal));              }          }      }      file.close();      return loadedUsers;  }  // --- User 类其他成员函数实现 ---  // getUsername, getPassword, displayInfo, changePassword, deposit, withdraw, checkBalance,  // saveUsersToFile, findUser, isUsernameExists, login  // 这些方法的实现应该保持不变，除非它们需要与新的购物车逻辑交互（但看起来不需要）。  std::string User::getUsername() const { return username; }  std::string User::getPassword() const { return password; } // 密码不应这样直接返回，仅为示例  void User::setUsername(const std::string &newUsername) { username = newUsername; }  void User::setPassword(const std::string &newPassword) { password = newPassword; }  void User::setBalance(double newBalance)  {      if (newBalance >= 0)          balance = newBalance;  }  void User::displayInfo(int index) const  {      if (index != -1)      {          cout << "用户 " << index << ": ";      }      cout << "用户名: " << username << ", 类型: " << userType << ", 余额: " << balance << endl;  }  bool User::changePassword(const std::string &oldPassword, const std::string &newPassword)  {      if (this->password == oldPassword)      {          this->password = newPassword;          return true;      }      cout << "原密码错误！" << endl;      return false;  }  bool User::deposit(double amount)  {      if (amount <= 0)      {          cout << "充值金额必须为正！" << endl;          return false;      }      balance += amount;      return true;  }  bool User::withdraw(double amount)  {      if (amount <= 0)      {          cout << "消费金额必须为正！" << endl;          return false;      }      if (balance < amount)      {          cout << "余额不足！" << endl;          return false;      }      balance -= amount;      return true;  }  double User::checkBalance() const  {      return balance;  }  bool User::saveUsersToFile(const std::vector<User \*> &users, const std::string &filename)  {      ofstream file(filename);      if (!file.is\_open())      {          cerr << "错误: 无法打开用户文件进行写入: " << filename << endl;          return false;      }      for (const auto \*user : users)      {          if (user)          {              file << user->getUsername() << ","                   << user->getPassword() << ","                   << user->getUserType() << ","                   << user->checkBalance() << endl;          }      }      file.close();      return true;  }  User \*User::findUser(const std::vector<User \*> &users, const std::string &findUsername)  {      for (User \*user : users)      {          if (user->getUsername() == findUsername)          {              return user;          }      }      return nullptr;  }  bool User::isUsernameExists(const std::vector<User \*> &users, const std::string &checkUsername)  {      return findUser(users, checkUsername) != nullptr;  }  User \*User::userLogin(const std::vector<User \*> &users)  {      string loginUsername, loginPassword;      cout << "请输入用户名: ";      getline(cin >> ws, loginUsername);      cout << "请输入密码: ";      getline(cin >> ws, loginPassword);      User \*foundUser = findUser(users, loginUsername);      if (foundUser && foundUser->getPassword() == loginPassword)      {          cout << "登录成功！欢迎, " << loginUsername << "!" << endl;          // 如果是 Customer，并且之前用户名为空（例如通过默认构造后设置用户名），则加载其购物车          if (Customer \*cust = dynamic\_cast<Customer \*>(foundUser))          {              if (cust->shoppingCartItems.empty() && !cust->getUsername().empty())              {                             // 假设 shoppingCartItems 为空表示未加载                                            // cust->cartDirectoryPath 应该在构造时已设置                  cust->loadCartFromFile(); // 尝试加载购物车              }          }          return foundUser;      }      cout << "用户名或密码错误！" << endl;      return nullptr;  }  User \*User::login(const std::vector<User \*> &users, const std::string &username\_param, const std::string &password\_param)  {      User \*foundUser = findUser(users, username\_param);      if (foundUser && foundUser->getPassword() == password\_param)      {          return foundUser;      }      return nullptr;  }  void User::displayAllUsers(const std::vector<User \*> &users)  {      if (users.empty())      {          cout << "当前没有注册用户。" << endl;          return;      }      cout << "\n--- 所有用户信息 ---" << endl;      for (size\_t i = 0; i < users.size(); ++i)      {          users[i]->displayInfo(i + 1);      }      cout << "--------------------" << endl;  } |

#### store.h

|  |
| --- |
| #ifndef STORE\_H  #define STORE\_H  #include <string>  #include <vector>  #include <iostream>  #include <fstream>  #include <sstream>  #include <iomanip>  #include <limits>  #include <map>  #include <filesystem>  #include <mutex>  #include "../user/user.h"  #include <algorithm>  #include <set>  #include <direct.h> // Windows 系统特定的目录操作  // 前向声明 User  class User;  // --- Product 基类和派生类 ---  class Product  {  protected:      std::string name;      std::string description;      double originalPrice;      int quantity;      double discountRate;      std::string sellerUsername; // 添加商品所属商家  public:      Product(std::string name, std::string desc, double price, int qty, std::string seller = "")          : name(name), description(desc), originalPrice(price), quantity(qty),            discountRate(0.0), sellerUsername(seller) {}      virtual ~Product() = default;      virtual double getPrice() const { return originalPrice \* (1.0 - discountRate); }      virtual void display() const;      virtual std::string getType() const = 0;      virtual std::string getUserCategory() const { return ""; } // 默认返回空字符串      virtual void save(std::ofstream &ofs) const;      // Getters and Setters      std::string getName() const { return name; }      std::string getDescription() const { return description; }      double getOriginalPrice() const { return originalPrice; }      int getQuantity() const { return quantity; }      double getDiscountRate() const { return discountRate; }      std::string getSellerUsername() const { return sellerUsername; } // 获取商品所属商家      void setName(const std::string &newName) { name = newName; }      void setDescription(const std::string &newDesc) { description = newDesc; }      void setOriginalPrice(double newPrice)      {          if (newPrice >= 0)              originalPrice = newPrice;      }      void setQuantity(int newQuantity)      {          if (newQuantity >= 0)              quantity = newQuantity;      }      void setDiscountRate(double newRate)      {          if (newRate >= 0.0 && newRate <= 1.0)              discountRate = newRate;      }      void setSellerUsername(const std::string &seller) { sellerUsername = seller; } // 设置商品所属商家  };  class Book : public Product  {  private:      std::string author;      std::string isbn;  public:      Book(std::string name, std::string desc, double price, int qty,           std::string auth, std::string isbn\_num, std::string seller = "")          : Product(name, desc, price, qty, seller), author(auth), isbn(isbn\_num) {}      std::string getType() const override { return "Book"; }      std::string getUserCategory() const override { return "Book"; } // 固定类型为 "Book"      void display() const override;      void save(std::ofstream &ofs) const override;      // 新增的 getters      std::string getAuthor() const { return author; }      std::string getIsbn() const { return isbn; }  };  class Clothing : public Product  {  private:      std::string size;      std::string color;  public:      Clothing(std::string name, std::string desc, double price, int qty,               std::string sz, std::string clr, std::string seller = "")          : Product(name, desc, price, qty, seller), size(sz), color(clr) {}      std::string getType() const override { return "Clothing"; }      std::string getUserCategory() const override { return "Clothing"; } // 固定类型为 "Clothing"      void display() const override;      void save(std::ofstream &ofs) const override;      // 新增的 getters      std::string getSize() const { return size; }      std::string getColor() const { return color; }  };  class Food : public Product  {  private:      std::string expirationDate;  public:      Food(std::string name, std::string desc, double price, int qty,           std::string expDate, std::string seller = "")          : Product(name, desc, price, qty, seller), expirationDate(expDate) {}      std::string getType() const override { return "Food"; }      std::string getUserCategory() const override { return "Food"; } // 固定类型为 "Food"      void display() const override;      void save(std::ofstream &ofs) const override;      // 新增的 getters      std::string getExpirationDate() const { return expirationDate; }  };  // 自定义商品类  class GenericProduct : public Product  {  private:      std::string categoryTag; // 用户自定义的分类标签  public:      GenericProduct(std::string name, std::string desc, double price, int qty,                     std::string tag, std::string seller = "")          : Product(name, desc, price, qty, seller), categoryTag(tag) {}      std::string getType() const override { return "Generic"; }           // 固定类型为 "Generic"      std::string getUserCategory() const override { return categoryTag; } // 返回用户自定义标签      void display() const override;      void save(std::ofstream &ofs) const override;      // Getter for categoryTag      std::string getCategoryTag() const { return categoryTag; }      void setCategoryTag(const std::string &tag) { categoryTag = tag; }  };  // --- Store Class ---  class Store  {  private:      std::vector<Product \*> allProducts;                           // 存储所有商家的商品      std::map<std::string, std::vector<Product \*>> sellerProducts; // 每个商家的商品映射      std::string storeDirectory;                                   // 商品文件所在目录      // 库存锁定数据结构      std::map<std::string, int> lockedInventory; // 商品名称 -> 锁定数量      mutable std::mutex inventoryMutex;          // 保护库存锁定操作的互斥锁      // 辅助方法      std::string getSellerFilename(const std::string &username) const;      bool saveProductsForSeller(const std::string &sellerUsername);      bool ensureDirectoryExists(const std::string &path) const;  public:      // 构造函数接收目录名      Store(const std::string &directory);      ~Store();      Product \*findProductByName(const std::string &name, const std::string &sellerUsername = "");      const Product \*findProductByName(const std::string &name, const std::string &sellerUsername = "") const;      // 文件操作      bool loadAllProducts();      bool loadSellerProducts(const std::string &sellerUsername);      bool saveAllProducts();      // 显示功能      void displayAllProducts() const;                                     // 显示所有商品      void displaySellerProducts(const std::string &sellerUsername) const; // 显示指定商家的商品      // 搜索功能      std::vector<Product \*> searchProductsByName(const std::string &searchTerm, const std::string &sellerUsername = "") const;      // 商家商品管理功能      bool createBook(User \*currentUser, const std::string &name, const std::string &desc,                      double price, int qty, const std::string &author, const std::string &isbn);      bool createClothing(User \*currentUser, const std::string &name, const std::string &desc,                          double price, int qty, const std::string &size, const std::string &color);      bool createFood(User \*currentUser, const std::string &name, const std::string &desc,                      double price, int qty, const std::string &expDate);      bool createGenericProduct(User \*currentUser, const std::string &name, const std::string &desc,                                double price, int qty, const std::string &categoryTag);      bool manageProductPrice(User \*currentUser, const std::string &productName, double newPrice);      bool manageProductQuantity(User \*currentUser, const std::string &productName, int newQuantity);      bool manageProductDiscount(User \*currentUser, const std::string &productName, double newDiscount);      bool applyCategoryDiscount(User \*currentUser, const std::string &category, double discount); // 获取商品      const std::vector<Product \*> &getProducts() const { return allProducts; }      std::vector<Product \*> getSellerProducts(const std::string &sellerUsername) const;      // 获取商家商品的唯一分类      std::vector<std::string> getUniqueCategoriesForSeller(const std::string &sellerUsername) const;      // 库存锁定功能      bool lockInventory(const std::string &productName, int quantity);      bool unlockInventory(const std::string &productName, int quantity);      bool hasAvailableInventory(const std::string &productName, int quantity) const;  };  #endif // STORE\_H |

#### store.cpp

|  |
| --- |
| #include "store.h"  namespace fs = std::filesystem;  using namespace std;  // --- Product 方法实现 ---  void Product::display() const  {      std::cout << "  商户名称：" << sellerUsername << std::endl;      std::cout << "  商品名称: " << name << std::endl;      std::cout << "  描述: " << description << std::endl;      if (!getUserCategory().empty())      {          std::cout << "  分类标签: " << getUserCategory() << std::endl;      }      if (discountRate > 0)      {          std::cout << "  原价: ¥" << std::fixed << std::setprecision(2) << originalPrice << std::endl;          std::cout << "  折扣: " << (discountRate \* 100) << "%" << std::endl;          std::cout << "  现价: ¥" << std::fixed << std::setprecision(2) << getPrice() << std::endl;      }      else      {          std::cout << "  价格: ¥" << std::fixed << std::setprecision(2) << getPrice() << std::endl;      }      std::cout << "  库存: " << quantity << " 件" << std::endl;      if (!sellerUsername.empty())      {          std::cout << "  商家: " << sellerUsername << std::endl;      }  }  void Product::save(std::ofstream &ofs) const  {      ofs << getType() << "," << name << "," << description << ","          << originalPrice << "," << quantity << "," << discountRate << "," << sellerUsername;  }  void Book::display() const // 复用父类  {      Product::display();      std::cout << "  作者: " << author << std::endl;      std::cout << "  ISBN: " << isbn << std::endl;  }  void Book::save(std::ofstream &ofs) const  {      Product::save(ofs);      ofs << "," << author << "," << isbn;  }  void Clothing::display() const  {      Product::display();      std::cout << "  尺寸: " << size << std::endl;      std::cout << "  颜色: " << color << std::endl;  }  void Clothing::save(std::ofstream &ofs) const  {      Product::save(ofs);      ofs << "," << size << "," << color;  }  void Food::display() const  {      Product::display();      std::cout << "  保质期限: " << expirationDate << std::endl;  }  void Food::save(std::ofstream &ofs) const  {      Product::save(ofs);      ofs << "," << expirationDate;  }  void GenericProduct::display() const  {      Product::display();  }  void GenericProduct::save(std::ofstream &ofs) const  {      Product::save(ofs);      ofs << "," << categoryTag;  }  // --- Store 类实现 ---  // 构造函数  Store::Store(const string &directory) : storeDirectory(directory)  {      // 确保商家目录存在      string sellersDir = storeDirectory + "/sellers";      ensureDirectoryExists(sellersDir);      // 加载所有商品数据      if (!loadAllProducts())      {          cerr << "警告: 无法加载商店数据。商店可能为空或文件丢失/损坏。" << endl;      }      else      {          cout << "已从商店加载 " << allProducts.size() << " 件商品。" << endl;      }  }  // 析构函数  Store::~Store()  {      for (Product \*p : allProducts)      {          delete p;      }      allProducts.clear();      sellerProducts.clear(); // 清除映射  }  // 确保目录存在  bool Store::ensureDirectoryExists(const string &path) const  {      try      {          if (!fs::exists(path))          {              fs::create\_directories(path);              cout << "创建目录: " << path << endl;          }          return true;      }      catch (const fs::filesystem\_error &e)      {          cerr << "创建目录失败 " << path << ": " << e.what() << endl;          // 尝试使用传统方法创建目录          if (\_mkdir(path.c\_str()) == 0)          {              cout << "使用传统方法创建目录: " << path << endl;              return true;          }          return false;      }      catch (...)      {          cerr << "创建目录时发生未知错误: " << path << endl;          return false;      }  }  // 获取商家文件名  string Store::getSellerFilename(const string &username) const  {      return storeDirectory + "/sellers/" + username + ".txt";  }  // 加载所有商品  bool Store::loadAllProducts()  {      // 清空现有商品      for (Product \*p : allProducts)      {          delete p;      }      allProducts.clear();      sellerProducts.clear();      // 获取sellers目录下的所有文件      string sellersDir = storeDirectory + "/sellers";      try      {          // 如果目录不存在，创建目录并返回成功（表示商店为空）          if (!fs::exists(sellersDir))          {              ensureDirectoryExists(sellersDir);              return true;          }          // 遍历目录中的所有文件          for (const auto &entry : fs::directory\_iterator(sellersDir))          {              if (entry.is\_regular\_file())              {                  string filename = entry.path().string();                  string sellerUsername = entry.path().stem().string(); // 获取不带扩展名的文件名                  loadSellerProducts(sellerUsername);              }          }          return true;      }      catch (const fs::filesystem\_error &e)      {          cerr << "读取商家目录失败: " << e.what() << endl;          return false;      }      catch (...)      {          cerr << "读取商家数据时发生未知错误" << endl;          return false;      }  }  // 加载指定商家的商品  bool Store::loadSellerProducts(const string &sellerUsername)  {      string filename = getSellerFilename(sellerUsername);      ifstream file(filename);      if (!file.is\_open())      {          // 对于新商家，文件不存在是正常的          return true;      }      vector<Product \*> sellerProductsList;      string line;      while (getline(file, line))      {          stringstream ss(line);          string segment;          vector<string> seglist;          while (getline(ss, segment, ','))          {              seglist.push\_back(segment);          }          // 格式检查: 至少需要7个字段 (类型,名称,描述,价格,数量,折扣,商家)          if (seglist.size() < 7)          {              cerr << "无效的产品数据行: " << line << endl;              continue;          }          try          {              string type = seglist[0];              string name = seglist[1];              string desc = seglist[2];              double price = stod(seglist[3]);              int qty = stoi(seglist[4]);              double discount = stod(seglist[5]);              string seller = seglist[6]; // 第7个字段是商家名              // 确认商家名称匹配              if (seller != sellerUsername)              {                  cerr << "警告: 文件 " << filename << " 中的商品所属商家 \""                       << seller << "\" 与预期的 \"" << sellerUsername << "\" 不符" << endl;                  seller = sellerUsername; // 强制使用文件名指示的商家              }              Product \*newProduct = nullptr;              if (type == "Book" && seglist.size() >= 9)              {                  // Book需要额外的author和isbn字段                  newProduct = new Book(name, desc, price, qty, seglist[7], seglist[8], seller);              }              else if (type == "Clothing" && seglist.size() >= 9)              {                  // Clothing需要额外的size和color字段                  newProduct = new Clothing(name, desc, price, qty, seglist[7], seglist[8], seller);              }              else if (type == "Food" && seglist.size() >= 8)              {                  // Food需要额外的expirationDate字段                  newProduct = new Food(name, desc, price, qty, seglist[7], seller);              }              else if (type == "Generic" && seglist.size() >= 8)              {                  // GenericProduct需要额外的categoryTag字段                  newProduct = new GenericProduct(name, desc, price, qty, seglist[7], seller);              }              else              {                  cerr << "未知商品类型或缺少必要字段: " << line << endl;                  continue;              }              if (newProduct)              {                  newProduct->setDiscountRate(discount);                  allProducts.push\_back(newProduct);                  sellerProductsList.push\_back(newProduct);              }          }          catch (const exception &e)          {              cerr << "解析商品数据时出错: " << e.what() << " 行: " << line << endl;          }      }      file.close();      // 更新商家商品映射      sellerProducts[sellerUsername] = sellerProductsList;      cout << "已加载商家 \"" << sellerUsername << "\" 的 "           << sellerProductsList.size() << " 件商品" << endl;      return true;  }  // 保存所有商品  bool Store::saveAllProducts()  {      // 对于每个商家，保存其商品      set<string> processedSellers; // 用于跟踪已处理的商家      for (const auto &product : allProducts)      {          string seller = product->getSellerUsername();          if (!seller.empty() && processedSellers.find(seller) == processedSellers.end())          {              saveProductsForSeller(seller);              processedSellers.insert(seller);          }      }      return true;  }  // 保存指定商家的商品  bool Store::saveProductsForSeller(const string &sellerUsername)  {      if (sellerUsername.empty())      {          cerr << "错误: 尝试保存无效商家的商品" << endl;          return false;      }      vector<Product \*> products;      // 获取该商家的所有商品      for (Product \*p : allProducts)      {          if (p->getSellerUsername() == sellerUsername)          {              products.push\_back(p);          }      }      string filename = getSellerFilename(sellerUsername);      ofstream file(filename);      if (!file.is\_open())      {          cerr << "错误: 无法打开文件保存商品: " << filename << endl;          return false;      }      for (const Product \*p : products)      {          p->save(file);          file << endl;      }      file.close();      // cout << "商家 \"" << sellerUsername << "\" 的 " << products.size()      //      << " 件商品已保存至 " << filename << endl;      return true;  }  // 查找商品  Product \*Store::findProductByName(const string &name, const string &sellerUsername)  {      if (!sellerUsername.empty())      {          // 只在指定商家的商品中查找          auto it = sellerProducts.find(sellerUsername);          if (it != sellerProducts.end())          {              for (Product \*p : it->second)              {                  if (p->getName() == name)                  {                      return p;                  }              }          }          return nullptr;      }      else      {          // 在所有商品中查找          for (Product \*p : allProducts)          {              if (p->getName() == name)              {                  return p;              }          }          return nullptr;      }  }  // 查找商品（const版本）  const Product \*Store::findProductByName(const string &name, const string &sellerUsername) const  {      if (!sellerUsername.empty())      {          // 只在指定商家的商品中查找          auto it = sellerProducts.find(sellerUsername);          if (it != sellerProducts.end())          {              for (const Product \*p : it->second)              {                  if (p->getName() == name)                  {                      return p;                  }              }          }          return nullptr;      }      else      {          // 在所有商品中查找          for (const Product \*p : allProducts)          {              if (p->getName() == name)              {                  return p;              }          }          return nullptr;      }  }  // 显示所有商品  void Store::displayAllProducts() const  {      if (allProducts.empty())      {          cout << "\n--- 商品列表为空 ---" << endl;          return;      }      cout << "\n--- 所有商品列表 ---" << endl;      int count = 0;      for (const Product \*p : allProducts)      {          cout << "--------------------------" << endl;          p->display();          count++;      }      cout << "--------------------------" << endl;      cout << "共显示 " << count << " 件商品" << endl;  }  // 显示指定商家的商品  void Store::displaySellerProducts(const string &sellerUsername) const  {      auto it = sellerProducts.find(sellerUsername);      if (it == sellerProducts.end() || it->second.empty())      {          cout << "\n--- 商家 \"" << sellerUsername << "\" 没有商品 ---" << endl;          return;      }      cout << "\n--- 商家 \"" << sellerUsername << "\" 的商品列表 ---" << endl;      int count = 0;      for (const Product \*p : it->second)      {          cout << "--------------------------" << endl;          p->display();          count++;      }      cout << "--------------------------" << endl;      cout << "共显示 " << count << " 件商品" << endl;  }  // 按名称搜索商品  vector<Product \*> Store::searchProductsByName(const string &searchTerm, const string &sellerUsername) const  {      vector<Product \*> results;      const vector<Product \*> &productsToSearch = sellerUsername.empty() ? allProducts : (sellerProducts.count(sellerUsername) ? sellerProducts.at(sellerUsername) : vector<Product \*>());      for (Product \*p : productsToSearch)      {          // 简单的包含搜索 (不区分大小写)          string productName = p->getName();          string searchTermLower = searchTerm;          // 将两个字符串转为小写进行比较          transform(productName.begin(), productName.end(), productName.begin(), ::tolower);          transform(searchTermLower.begin(), searchTermLower.end(), searchTermLower.begin(), ::tolower);          if (productName.find(searchTermLower) != string::npos)          {              results.push\_back(p);          }      }      return results;  }  // 获取商家的商品  vector<Product \*> Store::getSellerProducts(const string &sellerUsername) const  {      auto it = sellerProducts.find(sellerUsername);      return (it != sellerProducts.end()) ? it->second : vector<Product \*>();  }  // 创建图书  bool Store::createBook(User \*currentUser, const string &name, const string &desc,                         double price, int qty, const string &author, const string &isbn)  {      if (!currentUser)      {          cerr << "错误: 用户未登录" << endl;          return false;      }      if (currentUser->getUserType() != "seller")      {          cerr << "错误: 只有商家可以添加商品" << endl;          return false;      }      string sellerUsername = currentUser->getUsername();      // 检查该商家是否已有同名商品      if (findProductByName(name, sellerUsername) != nullptr)      {          cerr << "错误: 您已有名为 \"" << name << "\" 的商品" << endl;          return false;      }      if (price < 0 || qty < 0)      {          cerr << "错误: 价格和数量不能为负数" << endl;          return false;      }      try      {          Book \*newBook = new Book(name, desc, price, qty, author, isbn, sellerUsername);          allProducts.push\_back(newBook);          // 更新商家商品映射          sellerProducts[sellerUsername].push\_back(newBook);          // 保存商家的商品          if (saveProductsForSeller(sellerUsername))          {              cout << "商品 \"" << name << "\" 添加成功！" << endl;              return true;          }          else          {              cout << "商品 \"" << name << "\" 添加失败！" << endl;              return false;          }      }      catch (const bad\_alloc &e)      {          cerr << "错误: 无法分配内存: " << e.what() << endl;          return false;      }      catch (const exception &e)      {          cerr << "错误: " << e.what() << endl;          return false;      }      catch (...)      {          cerr << "错误: 添加商品时发生未知错误" << endl;          return false;      }  }  // 创建服装  bool Store::createClothing(User \*currentUser, const string &name, const string &desc,                             double price, int qty, const string &size, const string &color)  {      if (!currentUser)      {          cerr << "错误: 用户未登录" << endl;          return false;      }      if (currentUser->getUserType() != "seller")      {          cerr << "错误: 只有商家可以添加商品" << endl;          return false;      }      string sellerUsername = currentUser->getUsername();      // 检查该商家是否已有同名商品      if (findProductByName(name, sellerUsername) != nullptr)      {          cerr << "错误: 您已有名为 \"" << name << "\" 的商品" << endl;          return false;      }      if (price < 0 || qty < 0)      {          cerr << "错误: 价格和数量不能为负数" << endl;          return false;      }      try      {          Clothing \*newClothing = new Clothing(name, desc, price, qty, size, color, sellerUsername);          allProducts.push\_back(newClothing);          // 更新商家商品映射          sellerProducts[sellerUsername].push\_back(newClothing);          // 保存商家的商品          if (saveProductsForSeller(sellerUsername))          {              cout << "商品 \"" << name << "\" 添加成功！" << endl;              return true;          }          else          {              cout << "商品 \"" << name << "\" 添加失败！" << endl;              return false;          }      }      catch (const bad\_alloc &e)      {          cerr << "错误: 无法分配内存: " << e.what() << endl;          return false;      }      catch (const exception &e)      {          cerr << "错误: " << e.what() << endl;          return false;      }      catch (...)      {          cerr << "错误: 添加商品时发生未知错误" << endl;          return false;      }  }  // 创建食品  bool Store::createFood(User \*currentUser, const string &name, const string &desc,                         double price, int qty, const string &expDate)  {      if (!currentUser)      {          cerr << "错误: 用户未登录" << endl;          return false;      }      if (currentUser->getUserType() != "seller")      {          cerr << "错误: 只有商家可以添加商品" << endl;          return false;      }      string sellerUsername = currentUser->getUsername();      // 检查该商家是否已有同名商品      if (findProductByName(name, sellerUsername) != nullptr)      {          cerr << "错误: 您已有名为 \"" << name << "\" 的商品" << endl;          return false;      }      if (price < 0 || qty < 0)      {          cerr << "错误: 价格和数量不能为负数" << endl;          return false;      }      try      {          Food \*newFood = new Food(name, desc, price, qty, expDate, sellerUsername);          allProducts.push\_back(newFood);          // 更新商家商品映射          sellerProducts[sellerUsername].push\_back(newFood);          // 保存商家的商品          if (saveProductsForSeller(sellerUsername))          {              cout << "商品 \"" << name << "\" 添加成功！" << endl;              return true;          }          else          {              cout << "商品 \"" << name << "\" 添加失败！" << endl;              return false;          }      }      catch (const bad\_alloc &e)      {          cerr << "错误: 无法分配内存: " << e.what() << endl;          return false;      }      catch (const exception &e)      {          cerr << "错误: " << e.what() << endl;          return false;      }      catch (...)      {          cerr << "错误: 添加商品时发生未知错误" << endl;          return false;      }  }  bool Store::createGenericProduct(User \*currentUser, const string &name, const string &desc,                                   double price, int qty, const string &categoryTag)  {      if (!currentUser)      {          cerr << "错误: 用户未登录" << endl;          return false;      }      if (currentUser->getUserType() != "seller")      {          cerr << "错误: 只有商家可以添加商品" << endl;          return false;      }      string sellerUsername = currentUser->getUsername();      if (findProductByName(name, sellerUsername) != nullptr)      {          cerr << "错误: 您已有名为 \"" << name << "\" 的商品" << endl;          return false;      }      if (price < 0 || qty < 0)      {          cerr << "错误: 价格和数量不能为负数" << endl;          return false;      }      if (categoryTag.empty())      {          cerr << "错误: 通用商品必须指定分类标签" << endl;          return false;      }      try      {          GenericProduct \*newGenericProduct = new GenericProduct(name, desc, price, qty, categoryTag, sellerUsername);          allProducts.push\_back(newGenericProduct);          sellerProducts[sellerUsername].push\_back(newGenericProduct);          if (saveProductsForSeller(sellerUsername))          {              cout << "通用商品 \"" << name << "\" (分类: " << categoryTag << ") 添加成功！" << endl;              return true;          }          else          {              // 如果保存失败，应该从内存中移除，以保持一致性              allProducts.pop\_back();              sellerProducts[sellerUsername].pop\_back();              delete newGenericProduct;              cout << "通用商品 \"" << name << "\" 添加失败（保存错误）！" << endl;              return false;          }      }      catch (const bad\_alloc &e)      {          cerr << "错误: 无法分配内存: " << e.what() << endl;          return false;      }      catch (const exception &e)      {          cerr << "错误: " << e.what() << endl;          return false;      }      catch (...)      {          cerr << "错误: 添加通用商品时发生未知错误" << endl;          return false;      }  }  // 修改商品价格  bool Store::manageProductPrice(User \*currentUser, const string &productName, double newPrice)  {      if (!currentUser)      {          cerr << "错误: 用户未登录" << endl;          return false;      }      if (currentUser->getUserType() != "seller")      {          cerr << "错误: 只有商家可以修改商品价格" << endl;          return false;      }      string sellerUsername = currentUser->getUsername();      Product \*product = findProductByName(productName, sellerUsername);      if (!product)      {          cerr << "错误: 未找到您的商品 \"" << productName << "\"" << endl;          return false;      }      if (newPrice < 0)      {          cerr << "错误: 价格不能为负数" << endl;          return false;      }      product->setOriginalPrice(newPrice);      // 保存商家的商品      if (saveProductsForSeller(sellerUsername))      {          cout << "价格保存成功！" << endl;          return true;      }      else      {          cout << "价格保存失败！" << endl;          return false;      }  }  // 修改商品库存  bool Store::manageProductQuantity(User \*currentUser, const string &productName, int newQuantity)  {      if (!currentUser)      {          cerr << "错误: 用户未登录" << endl;          return false;      }      if (currentUser->getUserType() != "seller")      {          cerr << "错误: 只有商家可以修改商品库存" << endl;          return false;      }      string sellerUsername = currentUser->getUsername();      Product \*product = findProductByName(productName, sellerUsername);      if (!product)      {          cerr << "错误: 未找到您的商品 \"" << productName << "\"" << endl;          return false;      }      if (newQuantity < 0)      {          cerr << "错误: 库存不能为负数" << endl;          return false;      }      product->setQuantity(newQuantity);      if (saveProductsForSeller(sellerUsername))      {          cout << "库存修改成功！" << endl;          return true;      }      else      {          cout << "库存修改失败！" << endl;          return false;      }  }  // 修改商品折扣  bool Store::manageProductDiscount(User \*currentUser, const string &productName, double newDiscount)  {      if (!currentUser)      {          cerr << "错误: 用户未登录" << endl;          return false;      }      if (currentUser->getUserType() != "seller")      {          cerr << "错误: 只有商家可以修改商品折扣" << endl;          return false;      }      string sellerUsername = currentUser->getUsername();      Product \*product = findProductByName(productName, sellerUsername);      if (!product)      {          cerr << "错误: 未找到您的商品 \"" << productName << "\"" << endl;          return false;      }      if (newDiscount < 0 || newDiscount > 100)      {          cerr << "错误: 折扣必须在0到100之间" << endl;          return false;      }      product->setDiscountRate(newDiscount);      return saveProductsForSeller(sellerUsername);  }  std::vector<std::string> Store::getUniqueCategoriesForSeller(const std::string &sellerUsername) const  {      std::set<std::string> uniqueCategories;      auto it = sellerProducts.find(sellerUsername);      if (it != sellerProducts.end())      {          for (const Product \*p : it->second)          {              if (p)              {                                                // Ensure product pointer is not null                  std::string category = p->getUserCategory(); // This gets "Book", "Clothing", "Food", or custom tag                  if (!category.empty())                  { // Ensure not to add empty strings                      uniqueCategories.insert(category);                  }              }          }      }      return std::vector<std::string>(uniqueCategories.begin(), uniqueCategories.end());  }  // 应用分类折扣  bool Store::applyCategoryDiscount(User \*currentUser, const string &category, double discount)  {      if (!currentUser)      {          cerr << "错误: 用户未登录" << endl;          return false;      }      if (currentUser->getUserType() != "seller")      {          cerr << "错误: 只有商家可以应用折扣" << endl;          return false;      }      if (discount < 0.0 || discount > 1.0)      {          cerr << "错误: 折扣必须在0到100之间" << endl;          return false;      }      string sellerUsername = currentUser->getUsername();      bool changed = false; // 只应用折扣到该商家的商品      auto it = sellerProducts.find(sellerUsername);      if (it != sellerProducts.end())      {          for (Product \*p : it->second)          {              // 修复类别折扣匹配问题：使用getUserCategory()而不是getType()来匹配              std::string productCategory = p->getUserCategory();              if (!productCategory.empty())              {                  // 使用用户定义的类别进行匹配                  if (productCategory == category)                  {                      p->setDiscountRate(discount);                      changed = true;                  }              }              else              {                  // 处理标准类型以及自定义类型                  if (p->getType() == category)                  {                      p->setDiscountRate(discount);                      changed = true;                  }                  // 处理特殊情况："其他"选择时应用于所有非标准类型商品                  else if (category == "其他" && p->getType() != "Book" && p->getType() != "Clothing" && p->getType() != "Food")                  {                      p->setDiscountRate(discount);                      changed = true;                  }              }          }      }      if (changed)      {          return saveProductsForSeller(sellerUsername);      }      cerr << "错误: 未找到分类为 \"" << category << "\" 的商品" << endl;      return false;  }  // 库存锁定功能实现  bool Store::lockInventory(const std::string &productName, int quantity)  {      std::lock\_guard<std::mutex> lock(inventoryMutex);      // 查找商品      Product \*product = findProductByName(productName);      if (!product)      {          std::cerr << "商品不存在: " << productName << std::endl;          return false;      }      // 检查可用库存（实际库存 - 已锁定库存）      int currentLocked = 0;      auto it = lockedInventory.find(productName);      if (it != lockedInventory.end())      {          currentLocked = it->second;      }      int availableInventory = product->getQuantity() - currentLocked;      if (availableInventory < quantity)      {          std::cerr << "库存不足，无法锁定。商品: " << productName                    << ", 总库存: " << product->getQuantity()                    << ", 已锁定: " << currentLocked                    << ", 可用: " << availableInventory                    << ", 请求锁定: " << quantity << std::endl;          return false;      }      // 锁定库存      lockedInventory[productName] = currentLocked + quantity;      std::cout << "库存锁定成功: " << productName                << ", 锁定数量: " << quantity                << ", 总锁定: " << lockedInventory[productName] << std::endl;      return true;  }  bool Store::unlockInventory(const std::string &productName, int quantity)  {      std::lock\_guard<std::mutex> lock(inventoryMutex);      auto it = lockedInventory.find(productName);      if (it == lockedInventory.end())      {          std::cerr << "没有找到锁定的库存: " << productName << std::endl;          return false;      }      if (it->second < quantity)      {          std::cerr << "解锁数量超过锁定数量。商品: " << productName                    << ", 锁定数量: " << it->second                    << ", 请求解锁: " << quantity << std::endl;          return false;      }      // 解锁库存      it->second -= quantity;      // 如果锁定数量为0，从map中移除      if (it->second <= 0)      {          lockedInventory.erase(it);      }      std::cout << "库存解锁成功: " << productName                << ", 解锁数量: " << quantity << std::endl;      return true;  }  bool Store::hasAvailableInventory(const std::string &productName, int quantity) const  {      std::lock\_guard<std::mutex> lock(inventoryMutex);      const Product \*product = findProductByName(productName);      if (!product)      {          return false;      }      int currentLocked = 0;      auto it = lockedInventory.find(productName);      if (it != lockedInventory.end())      {          currentLocked = it->second;      }      int availableInventory = product->getQuantity() - currentLocked;      return availableInventory >= quantity;  } |

#### order.h

|  |
| --- |
| #ifndef ORDER\_H  #define ORDER\_H  #include <string>  #include <vector>  #include <ctime>  #include <iomanip>  // For std::setprecision  #include <iostream> // For display methods  // Forward declarations  class Product; // From store.h  class User;    // From user.h  class Store;   // From store.h  class OrderItem  {  public:      std::string productId;      std::string productName;      int quantity;      double priceAtPurchase;      std::string sellerUsername;      OrderItem(std::string pid, std::string pName, int qty, double price, std::string sUsername)          : productId(std::move(pid)), productName(std::move(pName)), quantity(qty),            priceAtPurchase(price), sellerUsername(std::move(sUsername)) {}      void display() const;      std::string toStringForSave() const;  };  class Order  {  private:      std::string orderId;      std::string customerUsername;      std::vector<OrderItem> items;      double totalAmount;      bool isProcessed; // Indicates if the order has been processed      time\_t orderTimestamp;      std::string status; // e.g., "PENDING\_CONFIRMATION", "COMPLETED", "CANCELLED\_INSUFFICIENT\_STOCK", "CANCELLED\_INSUFFICIENT\_FUNDS", "CANCELLED\_BY\_USER"      static std::string generateOrderId();  public:      Order(std::string custUsername);      // 新增默认构造函数      Order() : customerUsername(""),                totalAmount(0.0),                orderTimestamp(std::time(nullptr)), status("UNINITIALIZED"),                isProcessed(false)      {          orderId = "EMPTY-" + std::to\_string(orderTimestamp);      }      bool addItemFromProduct(const Product &product, int quantity); // Adds item and updates total      void addItem(const OrderItem &item);                           // Adds an existing OrderItem and updates total      void calculateTotalAmount();                                   // Recalculates total based on items      // Getters      std::string getOrderId() const { return orderId; }      std::string getCustomerUsername() const { return customerUsername; }      const std::vector<OrderItem> &getItems() const { return items; }      double getTotalAmount() const { return totalAmount; }      std::string getStatus() const { return status; }      time\_t getTimestamp() const { return orderTimestamp; } // Setters      void setStatus(const std::string &newStatus) { status = newStatus; }      void setProcessed(bool processed) { isProcessed = processed; }      // 添加必要的 setter 方法      void setTimestamp(time\_t timestamp) { orderTimestamp = timestamp; }      void setOrderId(const std::string &id) { orderId = id; }      void setTotalAmount(double amount) { totalAmount = amount; }      bool getProcessed() const { return isProcessed; }      void displaySummary() const;      std::string toStringForSaveHeader() const;      // Persistence (Optional - for saving orders to files)      // bool saveToFile(const std::string& directoryPath) const;      // static Order loadFromFile(const std::string& filePath);  };  #endif // ORDER\_H |

#### order.cpp

|  |
| --- |
| #include "order.h"  #include "../store/store.h"  #include "../user/user.h" // For Product and User definitions  #include <sstream>  #include <chrono>    // For generating unique ID  #include <fstream>   // For file operations (if saving orders)  #include <algorithm> // For std::remove\_if (if needed for item management)  // --- OrderItem Implementation ---  void OrderItem::display() const  {      std::cout << "    - 商品名称: " << productName << " (ID: " << productId << ")" << std::endl;      std::cout << "      购买数量: " << quantity << ", 商品单价: ¥" << priceAtPurchase                << ", 商品总价: ¥" << (quantity \* priceAtPurchase)                << ", 商家: " << sellerUsername << std::endl;  }  // --- Order Implementation ---  std::string Order::generateOrderId()  {      auto now = std::chrono::system\_clock::now();      auto now\_c = std::chrono::system\_clock::to\_time\_t(now); // Convert to time\_t for tm structure      std::tm now\_tm = \*std::localtime(&now\_c);               // Get local time structure      // Get milliseconds part      auto ms = std::chrono::duration\_cast<std::chrono::milliseconds>(now.time\_since\_epoch()) % 1000;      std::stringstream ss;      ss << "ORD-";      ss << std::put\_time(&now\_tm, "%Y%m%d-%H%M%S");                // Format: YYYYMMDD-HHMMSS      ss << "-" << std::setfill('0') << std::setw(3) << ms.count(); // Append milliseconds, zero-padded      return ss.str();  }  Order::Order(std::string custUsername)      : customerUsername(std::move(custUsername)), totalAmount(0.0), status("PENDING\_CREATION")  {      orderId = generateOrderId();      orderTimestamp = std::time(nullptr);  }  bool Order::addItemFromProduct(const Product &product, int quantity)  {      if (quantity <= 0)          return false;      // Note: Stock check should happen before calling this, or this method could also check      // For now, assumes stock check is done externally before adding to order object      items.emplace\_back(product.getName(), product.getName(), quantity, product.getPrice(), product.getSellerUsername());      calculateTotalAmount(); // Recalculate total every time an item is added      return true;  }  void Order::addItem(const OrderItem &item)  {      items.push\_back(item);      calculateTotalAmount(); // Recalculate total every time an item is added  }  void Order::calculateTotalAmount()  {      totalAmount = 0;      for (const auto &item : items)      {          totalAmount += item.priceAtPurchase \* item.quantity;      }  }  void Order::displaySummary() const  {      std::cout << "\n--- 订单汇总 ---" << std::endl;      std::cout << "订单 ID: " << orderId << std::endl;      std::cout << "买家: " << customerUsername << std::endl;      std::cout << "日期: " << std::asctime(std::localtime(&orderTimestamp)); // ctime adds newline      std::cout << "状态: " << status << std::endl;      std::cout << "下单物品:" << std::endl;      std::cout << std::fixed << std::setprecision(2);      for (const auto &item : items)      {          item.display();      }      std::cout << "---------------------" << std::endl;      std::cout << "总花费: ¥" << totalAmount << std::endl;      std::cout << "---------------------" << std::endl;      std::cout << std::resetiosflags(std::ios\_base::fixed | std::ios\_base::floatfield);  }  // 转为CSV格式字符串  std::string OrderItem::toStringForSave() const  {      std::stringstream ss;      ss << std::fixed << std::setprecision(2);      // Simple CSV-like, ensure no commas in your string fields or use a different delimiter/quoting      ss << productId << "," << productName << "," << quantity << "," << priceAtPurchase << "," << sellerUsername;      return ss.str();  }  std::string Order::toStringForSaveHeader() const  {      std::stringstream ss;      ss << std::fixed << std::setprecision(2);      ss << "OrderID:" << orderId << std::endl;      ss << "CustomerUsername:" << customerUsername << std::endl;      ss << "Timestamp:" << orderTimestamp << std::endl; // Save as raw time\_t      ss << "Status:" << status << std::endl;      ss << "TotalAmount:" << totalAmount << std::endl;      ss << "ItemsCount:" << items.size() << std::endl; // Useful for loading      return ss.str();  } |

#### ordermanager.h

|  |
| --- |
| #ifndef ORDER\_MANAGER\_H  #define ORDER\_MANAGER\_H  #include "../order/order.h"  #include "../store/store.h"  #include "../user/user.h"  #include <deque>  #include <string>  #include <vector>  #include <thread>  #include <mutex>  #include <condition\_variable>  #include <atomic>  #include <memory> // 为shared\_ptr添加  class OrderManager  {  private:      std::deque<std::shared\_ptr<Order>> orderQueue;      std::string completedOrdersDirectory;      std::string pendingOrdersFile;      // 多线程支持      std::thread processingThread;      mutable std::mutex queueMutex;      std::condition\_variable queueCondVar;      std::atomic<bool> stopProcessing;      // Helper to save a single order to its own file      bool saveOrderToFile(const Order &order) const;      // 处理线程的主函数      void processingLoop(Store &store, std::vector<User \*> &allUsers);      // 内部处理订单的方法      void processNextOrderInternal(std::shared\_ptr<Order> currentOrder, Store &store, std::vector<User \*> &allUsers);  public:      OrderManager(const std::string &ordersDir);      ~OrderManager();      // 启动和停止处理线程      void startProcessingThread(Store &store, std::vector<User \*> &allUsers);      void stopProcessingThread();      // 提交订单到队列      std::shared\_ptr<Order> submitOrderRequest(const Order &orderRequest);      // 处理队列中的订单      void processNextOrder(Store &store, std::vector<User \*> &allUsers);      void processAllPendingOrders(Store &store, std::vector<User \*> &allUsers);      // 添加这个声明！      size\_t getPendingOrderCount() const;      // 显示待处理订单      void displayPendingOrders() const;      // 获取指定用户的所有订单（包括待处理和已完成的）      void getUserOrders(const std::string &username,                         std::vector<std::shared\_ptr<Order>> &pendingOrders,                         std::vector<std::shared\_ptr<Order>> &completedOrders);  };  #endif // ORDER\_MANAGER\_H |

#### ordermanager.cpp

|  |
| --- |
| #include "ordermanager.h"  #include "../store/store.h"  #include "../order/order.h"  #include "../user/user.h"  #include <iostream>  #include <fstream>  #include <filesystem>  #include <iomanip>  #include <chrono>  #include <thread>  #include <mutex>  #include <algorithm>  // 声明外部变量 - 注释掉，在服务端版本中不需要  // extern const std::string USER\_FILE;  extern bool startProcessing; // 用于控制订单处理线程的启动状态  using namespace std;  // 构造函数，初始化订单目录  OrderManager::OrderManager(const std::string &ordersDir)      : completedOrdersDirectory(ordersDir), stopProcessing(false)  {      // 确保订单目录存在      if (!std::filesystem::exists(completedOrdersDirectory))      {          try          {              std::filesystem::create\_directories(completedOrdersDirectory);              cout << "已创建订单目录：" << completedOrdersDirectory << endl;          }          catch (const std::filesystem::filesystem\_error &e)          {              cerr << "创建订单目录失败: " << e.what() << endl;          }      }      // 设置待处理订单文件路径      pendingOrdersFile = completedOrdersDirectory + "/pending\_orders.txt";  }  // 析构函数 - 停止处理线程  OrderManager::~OrderManager()  {      stopProcessingThread();  }  // 启动处理线程  void OrderManager::startProcessingThread(Store &store, std::vector<User \*> &allUsers)  {      // 如果线程已经在运行，先停止      stopProcessingThread();      // 重置停止标志      stopProcessing = false;      // 启动处理线程      processingThread = std::thread(&OrderManager::processingLoop, this, std::ref(store), std::ref(allUsers));      cout << "订单处理线程已启动" << endl;  }  // 停止处理线程  void OrderManager::stopProcessingThread()  {      if (processingThread.joinable())      {          {              std::lock\_guard<std::mutex> lock(queueMutex);              stopProcessing = true; // 设置停止标志          }          // 通知条件变量，唤醒等待的线程          queueCondVar.notify\_one();          // 等待线程结束          processingThread.join();          cout << "订单处理线程已停止" << endl;      }  }  // 处理线程的主循环函数  void OrderManager::processingLoop(Store &store, std::vector<User \*> &allUsers)  {      cout << "订单处理线程开始运行..." << endl;      while (!stopProcessing)      {          std::shared\_ptr<Order> orderToProcess;          bool hasOrder = false;          {              // 获取队列互斥锁              std::unique\_lock<std::mutex> lock(queueMutex);              // 等待条件变量：直到队列非空或收到停止信号              queueCondVar.wait(lock, [this]                                { return !orderQueue.empty() || stopProcessing; });              // 如果收到停止信号且队列为空，则退出              if (stopProcessing && orderQueue.empty())                  break;              // 如果队列非空，取出队首订单              if (!orderQueue.empty())              {                  orderToProcess = orderQueue.front();                  orderQueue.pop\_front();                  hasOrder = true;              }          }          // 如果有订单需要处理，则处理它          if (hasOrder)          {              cout << "\n线程正在处理订单 ID: " << orderToProcess->getOrderId()                   << "，客户: " << orderToProcess->getCustomerUsername() << endl;              // 处理订单              processNextOrderInternal(orderToProcess, store, allUsers);          }      }      cout << "订单处理线程已终止" << endl;  }  // 内部处理订单的方法（被处理线程调用）  void OrderManager::processNextOrderInternal(std::shared\_ptr<Order> currentOrder, Store &store, std::vector<User \*> &allUsers)  {      cout << "正在处理订单 ID: " << currentOrder->getOrderId()           << "，客户: " << currentOrder->getCustomerUsername() << endl;      currentOrder->setStatus("PROCESSING");      // 找到订单对应的客户      Customer \*customer = dynamic\_cast<Customer \*>(User::findUser(allUsers, currentOrder->getCustomerUsername()));      if (!customer)      {          cerr << "错误: 找不到订单对应的客户 " << currentOrder->getCustomerUsername() << endl;          currentOrder->setStatus("FAILED\_CUSTOMER\_NOT\_FOUND");          saveOrderToFile(\*currentOrder);   // 注意这里需要解引用          currentOrder->setProcessed(true); // 标记处理完成          return;      }      // 第一阶段：重新验证商品（库存可能在队列等待期间发生变化）      for (const auto &item : currentOrder->getItems())      {          Product \*product = store.findProductByName(item.productId);          if (!product)          {              cerr << "错误: 商品 \"" << item.productName << "\" 不存在或已下架。订单取消。" << endl;              currentOrder->setStatus("FAILED\_PRODUCT\_NOT\_FOUND");              saveOrderToFile(\*currentOrder);              currentOrder->setProcessed(true);              return;          }          if (product->getQuantity() < item.quantity)          {              cerr << "错误: 商品 \"" << item.productName << "\" 库存不足 (需要 "                   << item.quantity << ", 现有 " << product->getQuantity() << ")。订单取消。" << endl;              currentOrder->setStatus("FAILED\_INSUFFICIENT\_STOCK");              saveOrderToFile(\*currentOrder);              currentOrder->setProcessed(true);              return;          }      }      // 第二阶段：验证客户余额      if (customer->checkBalance() < currentOrder->getTotalAmount())      {          cerr << "错误: 客户 " << customer->getUsername() << " 余额不足。订单取消。" << endl;          currentOrder->setStatus("FAILED\_INSUFFICIENT\_FUNDS");          saveOrderToFile(\*currentOrder);          currentOrder->setProcessed(true);          return;      }      // 第三阶段：处理支付和更新库存      if (!customer->withdraw(currentOrder->getTotalAmount()))      {          cerr << "严重错误: 客户 " << customer->getUsername()               << " 扣款失败，尽管余额检查已通过。订单: " << currentOrder->getOrderId() << endl;          currentOrder->setStatus("FAILED\_PAYMENT\_ERROR");          saveOrderToFile(\*currentOrder);          currentOrder->setProcessed(true);          return;      }      bool allSellersPaid = true;      for (const auto &item : currentOrder->getItems())      {          Product \*product = store.findProductByName(item.productId);          // 更新商品库存          product->setQuantity(product->getQuantity() - item.quantity);          // 向卖家转账          User \*seller = User::findUser(allUsers, item.sellerUsername);          if (seller)          {              double itemAmount = item.priceAtPurchase \* item.quantity;              if (!seller->deposit(itemAmount))              {                  cerr << "警告: 向商家 " << item.sellerUsername                       << " 支付商品 " << item.productName                       << " 的款项 ¥" << fixed << setprecision(2) << itemAmount                       << " 失败。订单 ID: " << currentOrder->getOrderId() << endl;                  allSellersPaid = false;              }              else              {                  cout << "已向商家 " << item.sellerUsername                       << " 支付 ¥" << fixed << setprecision(2) << itemAmount << endl;              }          }          else if (!item.sellerUsername.empty())          {              cerr << "警告: 找不到商品 \"" << item.productName                   << "\" 的商家 \"" << item.sellerUsername                   << "\"。订单 ID: " << currentOrder->getOrderId() << endl;              allSellersPaid = false;          }      } // 保存所有更改（用户和商店）      // 在服务端版本中，用户数据的保存由网络服务端统一管理      // User::saveUsersToFile(allUsers, userFilePath);      store.saveAllProducts();      // 设置最终订单状态      if (allSellersPaid)      {          currentOrder->setStatus("COMPLETED");      }      else      {          currentOrder->setStatus("COMPLETED\_WITH\_PAYMENT\_ISSUES");      }      cout << "订单 ID: " << currentOrder->getOrderId()           << " 处理完成。最终状态: " << currentOrder->getStatus() << endl;      currentOrder->displaySummary();      saveOrderToFile(\*currentOrder);      // 标记订单已处理完成（关键步骤！）      currentOrder->setProcessed(true);  }  // 将订单保存到文件  bool OrderManager::saveOrderToFile(const Order &order) const  {      if (completedOrdersDirectory.empty())      {          std::cerr << "错误: 未设置订单目录。" << std::endl;          return false;      }      // 创建基于订单ID的文件名      std::string filePath = completedOrdersDirectory + "/" + order.getOrderId() + ".txt";      std::ofstream outFile(filePath);      if (!outFile)      {          std::cerr << "错误: 无法打开文件保存订单: " << filePath << std::endl;          return false;      }      // 保存订单头部信息      outFile << order.toStringForSaveHeader();      // 保存订单项      for (const auto &item : order.getItems())      {          outFile << item.toStringForSave() << std::endl;      }      outFile.close();      std::cout << "订单 " << order.getOrderId() << " 已保存到 " << filePath << std::endl;      return true;  }  // 提交订单到处理队列 (线程安全版本)  std::shared\_ptr<Order> OrderManager::submitOrderRequest(const Order &orderRequest)  {      // 创建订单的共享指针副本      auto orderPtr = std::make\_shared<Order>(orderRequest);      // 设置订单状态为等待队列处理      orderPtr->setStatus("PENDING\_IN\_QUEUE");      orderPtr->setProcessed(false);      // 锁定队列      std::lock\_guard<std::mutex> lock(queueMutex);      // 将订单加入队列      orderQueue.push\_back(orderPtr);      // 通知处理线程有新订单      queueCondVar.notify\_one();      cout << "订单 ID: " << orderPtr->getOrderId()           << " 已提交到处理队列，当前队列长度: " << orderQueue.size() << endl;      // 返回订单共享指针      return orderPtr;  }  // 处理队列中的一个订单 (公共方法版，委托给内部方法)  void OrderManager::processNextOrder(Store &store, std::vector<User \*> &allUsers)  {      std::shared\_ptr<Order> orderToProcess;      bool hasOrder = false;      {          std::lock\_guard<std::mutex> lock(queueMutex);          if (!orderQueue.empty())          {              orderToProcess = orderQueue.front();              orderQueue.pop\_front();              hasOrder = true;          }      }      if (hasOrder)      {          processNextOrderInternal(orderToProcess, store, allUsers);      }  }  // 处理所有待处理订单 (线程安全版本)  void OrderManager::processAllPendingOrders(Store &store, std::vector<User \*> &allUsers)  {      std::vector<std::shared\_ptr<Order>> ordersToProcess;      // 首先，从队列中安全地取出所有订单      {          std::lock\_guard<std::mutex> lock(queueMutex);          while (!orderQueue.empty())          {              ordersToProcess.push\_back(orderQueue.front());              orderQueue.pop\_front();          }      }      // 如果没有订单，直接返回      if (ordersToProcess.empty())      {          cout << "当前没有待处理订单。" << endl;          return;      }      cout << "\n--- 开始处理 " << ordersToProcess.size() << " 个待处理订单 ---" << endl;      // 处理获取的订单      for (auto &order : ordersToProcess)      {          processNextOrderInternal(order, store, allUsers);      }      cout << "--- 成功处理 " << ordersToProcess.size() << " 个订单 ---" << endl;  }  // 获取待处理订单数量 (线程安全版本)  size\_t OrderManager::getPendingOrderCount() const  {      std::lock\_guard<std::mutex> lock(queueMutex);      return orderQueue.size();  }  // 显示待处理订单 (线程安全版本)  void OrderManager::displayPendingOrders() const  {      std::lock\_guard<std::mutex> lock(queueMutex);      if (orderQueue.empty())      {          cout << "当前队列中没有待处理订单。" << endl;          return;      }      cout << "\n--- 待处理订单队列 ---" << endl;      int i = 1;      for (const auto &order : orderQueue)      {          cout << i++ << ". 订单 ID: " << order->getOrderId()               << ", 客户: " << order->getCustomerUsername()               << ", 金额: ¥" << fixed << setprecision(2) << order->getTotalAmount()               << ", 状态: " << order->getStatus() << endl;      }      cout << "-----------------------------" << endl;  }  // 获取指定用户的所有订单  void OrderManager::getUserOrders(const std::string &username,                                   std::vector<std::shared\_ptr<Order>> &pendingOrders,                                   std::vector<std::shared\_ptr<Order>> &completedOrders)  {      // 清空传入的向量      pendingOrders.clear();      completedOrders.clear();      // 获取队列中的待处理订单      {          std::lock\_guard<std::mutex> lock(queueMutex);          for (const auto &order : orderQueue)          {              if (order->getCustomerUsername() == username)              {                  pendingOrders.push\_back(order);              }          }      }      // 获取已完成的订单      try      {          // 检查订单目录是否存在          if (!std::filesystem::exists(completedOrdersDirectory))          {              std::cerr << "订单目录不存在: " << completedOrdersDirectory << std::endl;              return;          }          // 遍历订单目录中的所有文件          for (const auto &entry : std::filesystem::directory\_iterator(completedOrdersDirectory))          {              if (entry.is\_regular\_file() && entry.path().extension() == ".txt")              {                  // 从文件名中提取订单ID                  std::string filename = entry.path().filename().string();                  if (filename.find("ORD-") == 0) // 确保是订单文件                  {                      // 尝试读取订单文件                      std::ifstream file(entry.path());                      if (file.is\_open())                      {                          std::string line;                          bool isUserOrder = false;                          std::string orderId, status;                          double totalAmount = 0.0;                          time\_t timestamp = 0;                          int itemsCount = 0; // 修改变量名                          std::string productId;                          std::string productName;                          int quantity;                          double priceAtPurchase;                          std::string sellerUsername; // 读取订单头部信息                          for (int i = 0; i < 6 && std::getline(file, line); i++)                          {                              switch (i)                              {                              case 0:                                  orderId = line.substr(8); // 识别从传参开始到末尾的字符串                                  break;                              case 1:                                  if (line.substr(17) == username)                                  {                                      isUserOrder = true;                                  }                                  else                                  {                                      isUserOrder = false;                                  }                                  break;                              case 2:                                  timestamp = std::stoll(line.substr(10));                                  break;                              case 3:                                  status = line.substr(7);                                  break;                              case 4:                                  totalAmount = std::stod(line.substr(12));                                  break;                              case 5:                                  itemsCount = std::stoi(line.substr(11)); // 修复：使用正确的变量名和函数                                  break;                              }                          }                          // 如果不是用户的订单，跳过商品项读取                          if (!isUserOrder)                          {                              // 跳过商品项                              for (int i = 0; i < itemsCount; i++)                              {                                  std::getline(file, line);                              }                              continue; // 继续处理下一个订单文件                          } // 如果是用户的订单，则创建一个新的订单对象并添加到相应列表                          std::vector<OrderItem> items;                          // 读取订单中的指定数量的商品项                          for (int i = 0; i < itemsCount && std::getline(file, line); i++)                          {                              std::istringstream iss(line);                              std::string part;                              std::vector<std::string> parts;                              // 解析逗号分隔的值                              while (std::getline(iss, part, ','))                              {                                  parts.push\_back(part);                              }                              // 确保解析了足够的部分                              if (parts.size() >= 5)                              {                                  productId = parts[0];                                  productName = parts[1];                                  quantity = std::stoi(parts[2]);                                  priceAtPurchase = std::stod(parts[3]);                                  sellerUsername = parts[4];                                  // 创建OrderItem并添加到items向量                                  OrderItem item(productId, productName, quantity, priceAtPurchase, sellerUsername);                                  items.push\_back(item);                              }                          }                          if (isUserOrder)                          {                              auto order = std::make\_shared<Order>(username);                              order->setStatus(status);                              // 如果有订单ID，设置它（否则会生成新的ID）                              if (!orderId.empty())                              {                                  order->setOrderId(orderId);                              }                              if (totalAmount > 0)                              {                                  // 设置总金额                                  order->setTotalAmount(totalAmount); // 先计算一次                              }                              // 如果有时间戳，设置它                              if (timestamp > 0)                              {                                  // 我们已经添加了 setTimestamp 方法                                  order->setTimestamp(timestamp);                              }                              // 添加所有解析出的订单项到订单中                              for (const auto &item : items)                              {                                  order->addItem(item);                              }                              // 添加到已完成订单列表                              // 根据状态判断是已完成还是失败/取消                              if (status.find("COMPLETED") == 0 ||                                  status.find("FAILED") == 0 ||                                  status.find("CANCELLED") == 0)                              {                                  completedOrders.push\_back(order);                              }                          }                      }                  }              }          }      }      catch (const std::exception &e)      {          std::cerr << "读取订单文件时出错: " << e.what() << std::endl;      }      // 按时间戳排序（可选）      // 使用显式的标准库排序算法      ::std::sort(pendingOrders.begin(), pendingOrders.end(),                  [](const std::shared\_ptr<Order> &a, const std::shared\_ptr<Order> &b)                  {                      return a->getTimestamp() > b->getTimestamp(); // 从新到旧                  });      ::std::sort(completedOrders.begin(), completedOrders.end(),                  [](const std::shared\_ptr<Order> &a, const std::shared\_ptr<Order> &b)                  {                      return a->getTimestamp() > b->getTimestamp(); // 从新到旧                  });  } |

#### protocol.h

|  |
| --- |
| #ifndef PROTOCOL\_H  #define PROTOCOL\_H  #include <string>  #include <vector>  #include <map>  // 网络协议定义  namespace Protocol  {      // 消息类型枚举      enum class MessageType : int      {          // 用户相关          USER\_LOGIN = 1000,          USER\_REGISTER = 1001,          USER\_LOGOUT = 1002,          USER\_GET\_INFO = 1003,          USER\_UPDATE\_BALANCE = 1004,          USER\_CHANGE\_PASSWORD = 1005, // 商品相关          PRODUCT\_GET\_ALL = 2000,          PRODUCT\_SEARCH = 2001,          PRODUCT\_GET\_BY\_ID = 2002,          PRODUCT\_ADD = 2003,          PRODUCT\_UPDATE = 2004,          PRODUCT\_DELETE = 2005,          PRODUCT\_UPDATE\_STOCK = 2006,          PRODUCT\_MANAGE\_PRICE = 2007,          PRODUCT\_MANAGE\_QUANTITY = 2008,          PRODUCT\_MANAGE\_DISCOUNT = 2009,          PRODUCT\_APPLY\_CATEGORY\_DISCOUNT = 2010,          // 购物车相关          CART\_GET = 3000,          CART\_ADD\_ITEM = 3001,          CART\_UPDATE\_ITEM = 3002,          CART\_REMOVE\_ITEM = 3003,          CART\_CLEAR = 3004, // 订单相关          ORDER\_CREATE = 4000,          ORDER\_DIRECT\_PURCHASE = 4001,          ORDER\_GET\_BY\_USER = 4002,          ORDER\_GET\_BY\_ID = 4003,          ORDER\_UPDATE\_STATUS = 4004,          ORDER\_GET\_ALL = 4005,          // 库存锁定相关          INVENTORY\_LOCK = 4100,          INVENTORY\_UNLOCK = 4101,          INVENTORY\_RESERVE = 4102,          // 响应          RESPONSE\_SUCCESS = 5000,          RESPONSE\_ERROR = 5001,          RESPONSE\_DATA = 5002,          // 心跳          HEARTBEAT = 6000      };      // 用户类型      enum class UserType : int      {          CUSTOMER = 1,          SELLER = 2,          ADMIN = 3      };      // 订单状态      enum class OrderStatus : int      {          PENDING = 1,          COMPLETED = 2,          CANCELLED\_STOCK = 3,          CANCELLED\_FUNDS = 4,          CANCELLED\_USER = 5      };      // 网络消息结构      struct Message      {          MessageType type;          std::string sessionId;                   // 会话ID          std::map<std::string, std::string> data; // 消息数据          Message() : type(MessageType::HEARTBEAT) {}          Message(MessageType t, const std::string &sid = "") : type(t), sessionId(sid) {}          // 序列化为字符串          std::string serialize() const;          // 从字符串反序列化          static Message deserialize(const std::string &str);          // 设置数据字段          void setData(const std::string &key, const std::string &value)          {              data[key] = value;          }          // 获取数据字段          std::string getData(const std::string &key) const          {              auto it = data.find(key);              return (it != data.end()) ? it->second : "";          }      };      // 用户数据结构      struct UserData      {          std::string username;          std::string password;          UserType userType;          double balance;          std::string serialize() const;          static UserData deserialize(const std::string &str);      }; // 商品数据结构      struct ProductData      {          std::string id;          std::string name;          std::string description;          std::string type;     // book, clothing, food, generic          double price;         // 折扣后的实际价格          double originalPrice; // 原价          int quantity;          double discountRate;          std::string sellerUsername;          std::map<std::string, std::string> attributes; // 额外属性          std::string serialize() const;          static ProductData deserialize(const std::string &str);      };      // 购物车项数据结构      struct CartItemData      {          std::string productId;          std::string productName;          int quantity;          double priceAtAddition;          std::string sellerUsername;          std::string serialize() const;          static CartItemData deserialize(const std::string &str);      };      // 订单数据结构      struct OrderData      {          std::string orderId;          std::string customerUsername;          std::vector<CartItemData> items;          double totalAmount;          OrderStatus status;          long long timestamp;          std::string serialize() const;          static OrderData deserialize(const std::string &str);      };  } // namespace Protocol  #endif // PROTOCOL\_H |

#### protocol.cpp

|  |
| --- |
| #include "protocol.h"  #include <sstream>  #include <iostream>  namespace Protocol  {      // Message序列化实现      std::string Message::serialize() const      {          std::ostringstream oss;          oss << static\_cast<int>(type) << "|" << sessionId << "|";          // 序列化数据字段          oss << data.size() << "|";          for (const auto &pair : data)          {              oss << pair.first << "=" << pair.second << "|";          }          return oss.str();      }      Message Message::deserialize(const std::string &str)      {          Message msg;          size\_t pos = 0;          std::string token;          try          {              // 解析消息类型              size\_t next\_pos = str.find('|', pos);              if (next\_pos != std::string::npos)              {                  token = str.substr(pos, next\_pos - pos);                  msg.type = static\_cast<MessageType>(std::stoi(token));                  pos = next\_pos + 1;              }              // 解析会话ID              next\_pos = str.find('|', pos);              if (next\_pos != std::string::npos)              {                  msg.sessionId = str.substr(pos, next\_pos - pos);                  pos = next\_pos + 1;              }              // 解析数据字段数量              int dataCount = 0;              next\_pos = str.find('|', pos);              if (next\_pos != std::string::npos)              {                  token = str.substr(pos, next\_pos - pos);                  dataCount = std::stoi(token);                  pos = next\_pos + 1;              }              // 解析数据字段 - 改进的逻辑              int parsedCount = 0;              while (parsedCount < dataCount && pos < str.length())              {                  // 查找下一个 key=value 对                  next\_pos = str.find('|', pos);                  if (next\_pos == std::string::npos)                  {                      // 最后一个字段                      token = str.substr(pos);                  }                  else                  {                      token = str.substr(pos, next\_pos - pos);                  }                  // 解析 key=value                  size\_t eq\_pos = token.find('=');                  if (eq\_pos != std::string::npos)                  {                      std::string key = token.substr(0, eq\_pos);                      std::string value = token.substr(eq\_pos + 1); // 对于包含内部分隔符的值（如订单数据），需要特殊处理                      if (key.find("order\_") == 0)                      {                          // 这是订单数据，可能包含内部的 | 分隔符                          // 需要查找到下一个 key= 模式来确定真正的结束位置                          size\_t search\_pos = next\_pos + 1;                          while (search\_pos < str.length())                          {                              size\_t temp\_pos = str.find('|', search\_pos);                              if (temp\_pos == std::string::npos)                              {                                  // 没有更多的 | 分隔符，包含剩余所有内容                                  if (search\_pos < str.length())                                  {                                      value += "|" + str.substr(search\_pos);                                  }                                  next\_pos = std::string::npos; // 标记为最后一个字段                                  break;                              }                              std::string temp\_token = str.substr(search\_pos, temp\_pos - search\_pos);                              if (temp\_token.find('=') != std::string::npos &&                                  (temp\_token.find("order\_") == 0 || temp\_token == "count" || temp\_token.find("user\_") == 0))                              {                                  // 找到了下一个有效的 key=value 对                                  break;                              }                              // 继续包含到值中                              value += "|" + temp\_token;                              next\_pos = temp\_pos;                              search\_pos = temp\_pos + 1;                          }                      }                      msg.data[key] = value;                  }                  parsedCount++;                  if (next\_pos != std::string::npos)                  {                      pos = next\_pos + 1;                  }                  else                  {                      break;                  }              }          }          catch (const std::exception &e)          {              std::cerr << "消息反序列化错误: " << e.what() << std::endl;          }          return msg;      }      // UserData序列化实现      std::string UserData::serialize() const      {          std::ostringstream oss;          oss << username << "," << password << "," << static\_cast<int>(userType) << "," << balance;          return oss.str();      }      UserData UserData::deserialize(const std::string &str)      {          UserData user;          std::istringstream iss(str);          std::string token;          try          {              if (std::getline(iss, token, ','))                  user.username = token;              if (std::getline(iss, token, ','))                  user.password = token;              if (std::getline(iss, token, ','))                  user.userType = static\_cast<UserType>(std::stoi(token));              if (std::getline(iss, token, ','))                  user.balance = std::stod(token);          }          catch (const std::exception &e)          {              std::cerr << "用户数据反序列化错误: " << e.what() << std::endl;          }          return user;      } // ProductData序列化实现      std::string ProductData::serialize() const      {          std::ostringstream oss;          oss << id << "," << name << "," << description << "," << type << ","              << price << "," << originalPrice << "," << quantity << "," << discountRate << "," << sellerUsername;          // 序列化额外属性          oss << "," << attributes.size();          for (const auto &attr : attributes)          {              oss << "," << attr.first << "=" << attr.second;          }          return oss.str();      }      ProductData ProductData::deserialize(const std::string &str)      {          ProductData product;          std::istringstream iss(str);          std::string token;          try          {              if (std::getline(iss, token, ','))                  product.id = token;              if (std::getline(iss, token, ','))                  product.name = token;              if (std::getline(iss, token, ','))                  product.description = token;              if (std::getline(iss, token, ','))                  product.type = token;              if (std::getline(iss, token, ','))                  product.price = std::stod(token);              if (std::getline(iss, token, ','))                  product.originalPrice = std::stod(token);              if (std::getline(iss, token, ','))                  product.quantity = std::stoi(token);              if (std::getline(iss, token, ','))                  product.discountRate = std::stod(token);              if (std::getline(iss, token, ','))                  product.sellerUsername = token;              // 解析额外属性              int attrCount = 0;              if (std::getline(iss, token, ','))              {                  attrCount = std::stoi(token);              }              for (int i = 0; i < attrCount; ++i)              {                  if (std::getline(iss, token, ','))                  {                      size\_t pos = token.find('=');                      if (pos != std::string::npos)                      {                          std::string key = token.substr(0, pos);                          std::string value = token.substr(pos + 1);                          product.attributes[key] = value;                      }                  }              }          }          catch (const std::exception &e)          {              std::cerr << "商品数据反序列化错误: " << e.what() << std::endl;          }          return product;      }      // CartItemData序列化实现      std::string CartItemData::serialize() const      {          std::ostringstream oss;          oss << productId << ";" << productName << ";" << quantity << ";"              << priceAtAddition << ";" << sellerUsername; // Changed delimiter to ;          return oss.str();      }      CartItemData CartItemData::deserialize(const std::string &str)      {          CartItemData item;          std::istringstream iss(str);          std::string token;          try          {              if (std::getline(iss, token, ';')) // Changed delimiter to ;                  item.productId = token;              if (std::getline(iss, token, ';')) // Changed delimiter to ;                  item.productName = token;              if (std::getline(iss, token, ';')) // Changed delimiter to ;                  item.quantity = std::stoi(token);              if (std::getline(iss, token, ';')) // Changed delimiter to ;                  item.priceAtAddition = std::stod(token);              if (std::getline(iss, token, ';')) // Changed delimiter to ;                  item.sellerUsername = token;          }          catch (const std::exception &e)          {              std::cerr << "购物车项数据反序列化错误: " << e.what() << std::endl;          }          return item;      }      // OrderData序列化实现      std::string OrderData::serialize() const      {          std::ostringstream oss;          oss << orderId << "," << customerUsername << "," << totalAmount << ","              << static\_cast<int>(status) << "," << timestamp << "," << items.size();          for (const auto &item : items)          {              oss << "|" << item.serialize(); // 使用不同的分隔符避免冲突          }          return oss.str();      }      OrderData OrderData::deserialize(const std::string &str)      {          OrderData order;          std::istringstream iss(str);          std::string token;          try          {              if (std::getline(iss, token, ','))                  order.orderId = token;              if (std::getline(iss, token, ','))                  order.customerUsername = token;              if (std::getline(iss, token, ','))                  order.totalAmount = std::stod(token);              if (std::getline(iss, token, ','))                  order.status = static\_cast<OrderStatus>(std::stoi(token));              if (std::getline(iss, token, ','))                  order.timestamp = std::stoll(token);              int itemCount = 0;              if (std::getline(iss, token, '|'))              {                  itemCount = std::stoi(token);              }              if (itemCount > 0)              {                  std::string allItemsStringSegment;                  std::getline(iss, allItemsStringSegment);                  if (!allItemsStringSegment.empty())                  {                      std::istringstream itemStream(allItemsStringSegment);                      std::string singleItemSerialized;                      for (int i = 0; i < itemCount; ++i)                      {                          if (std::getline(itemStream, singleItemSerialized, '|'))                          {                              if (!singleItemSerialized.empty())                              {                                  order.items.push\_back(CartItemData::deserialize(singleItemSerialized));                              }                              else                              {                                  std::cerr << "Warning: Order " << order.orderId << ": Encountered an empty string segment for item at index " << i                                            << " (expected " << itemCount << " total items)." << std::endl;                              }                          }                          else                          {                              std::cerr << "Error: Order " << order.orderId << ": Expected " << itemCount                                        << " items, but could only read " << i                                        << " segments. Data may be malformed or truncated." << std::endl;                              break;                          }                      }                  }                  else                  {                      std::cerr << "Error: Order " << order.orderId << ": Expected item data starting with '|' for "                                << itemCount << " items, but found: \"" << (allItemsStringSegment.length() > 20 ? allItemsStringSegment.substr(0, 20) + "..." : allItemsStringSegment) << "\"" << std::endl;                  }              }              // 如果 itemCount 为 0，不需要读取任何订单项数据，这是正常情况          }          catch (const std::exception &e)          {              std::cerr << "订单数据反序列化错误: " << e.what() << std::endl;          }          return order;      }  } // namespace Protocol |

#### server.h

|  |
| --- |
| #ifndef NETWORK\_SERVER\_H  #define NETWORK\_SERVER\_H  #include "../network/protocol.h"  #include <winsock2.h>  #include <ws2tcpip.h>  #include <string>  #include <vector>  #include <map>  #include <thread>  #include <mutex>  #include <atomic>  #include <memory>  #pragma comment(lib, "ws2\_32.lib")  // 前向声明  class User;  class Store;  class OrderManager;  class Product;  // 客户端会话类  class ClientSession : public std::enable\_shared\_from\_this<ClientSession>  {  private:      SOCKET clientSocket;      std::string sessionId;      std::string username;      Protocol::UserType userType;      std::thread clientThread;      std::atomic<bool> isActive;  public:      ClientSession(SOCKET socket, const std::string &sid);      ~ClientSession();      SOCKET getSocket() const { return clientSocket; }      std::string getSessionId() const { return sessionId; }      std::string getUsername() const { return username; }      Protocol::UserType getUserType() const { return userType; }      bool isSessionActive() const { return isActive.load(); }      void setUsername(const std::string &user) { username = user; }      void setUserType(Protocol::UserType type) { userType = type; }      bool sendMessage(const Protocol::Message &message);      std::string receiveRawData();      void startSession(class NetworkServer \*server);      void stopSession();  private:      void sessionLoop(class NetworkServer \*server);      bool sendRawData(const std::string &data);  };  class NetworkServer  {  private:      SOCKET serverSocket;      int port;      std::atomic<bool> isRunning;      std::thread acceptThread;      // 客户端会话管理      std::vector<std::shared\_ptr<ClientSession>> sessions;      std::mutex sessionsMutex;      // 业务逻辑组件      std::vector<User \*> users;      std::unique\_ptr<Store> store;      std::unique\_ptr<OrderManager> orderManager;      // 数据文件路径      std::string userFile;      std::string storeDir;      std::string orderDir;      // 内部方法      void acceptLoop();      std::string generateSessionId();      std::shared\_ptr<ClientSession> findSession(const std::string &sessionId);      void removeSession(const std::string &sessionId);      std::shared\_ptr<ClientSession> findSessionByUsername(const std::string &username);      // 数据转换方法      Protocol::UserData convertToUserData(const User \*user);      Protocol::ProductData convertToProductData(const Product \*product);      std::vector<Protocol::ProductData> convertToProductDataList(const std::vector<Product \*> &products);  public:      NetworkServer(int port = 8888);      ~NetworkServer();      // 服务器控制      bool start();      void stop();      bool isServerRunning() const { return isRunning.load(); }      // 客户端会话处理      void handleClientMessage(std::shared\_ptr<ClientSession> session, const Protocol::Message &message);      // 用户管理处理      void handleUserLogin(std::shared\_ptr<ClientSession> session, const Protocol::Message &message);      void handleUserRegister(std::shared\_ptr<ClientSession> session, const Protocol::Message &message);      void handleUserLogout(std::shared\_ptr<ClientSession> session, const Protocol::Message &message);      void handleUserGetInfo(std::shared\_ptr<ClientSession> session, const Protocol::Message &message);      void handleUserUpdateBalance(std::shared\_ptr<ClientSession> session, const Protocol::Message &message);      void handleUserChangePassword(std::shared\_ptr<ClientSession> session, const Protocol::Message &message);      // 商品管理处理      void handleProductGetAll(std::shared\_ptr<ClientSession> session, const Protocol::Message &message);      void handleProductSearch(std::shared\_ptr<ClientSession> session, const Protocol::Message &message);      void handleProductGetById(std::shared\_ptr<ClientSession> session, const Protocol::Message &message);      void handleProductAdd(std::shared\_ptr<ClientSession> session, const Protocol::Message &message);      void handleProductUpdate(std::shared\_ptr<ClientSession> session, const Protocol::Message &message);      void handleProductDelete(std::shared\_ptr<ClientSession> session, const Protocol::Message &message);      void handleProductUpdateStock(std::shared\_ptr<ClientSession> session, const Protocol::Message &message);      void handleProductManagePrice(std::shared\_ptr<ClientSession> session, const Protocol::Message &message);      void handleProductManageQuantity(std::shared\_ptr<ClientSession> session, const Protocol::Message &message);      void handleProductManageDiscount(std::shared\_ptr<ClientSession> session, const Protocol::Message &message);      void handleProductApplyCategoryDiscount(std::shared\_ptr<ClientSession> session, const Protocol::Message &message);      // 购物车管理处理      void handleCartGet(std::shared\_ptr<ClientSession> session, const Protocol::Message &message);      void handleCartAddItem(std::shared\_ptr<ClientSession> session, const Protocol::Message &message);      void handleCartUpdateItem(std::shared\_ptr<ClientSession> session, const Protocol::Message &message);      void handleCartRemoveItem(std::shared\_ptr<ClientSession> session, const Protocol::Message &message);      void handleCartClear(std::shared\_ptr<ClientSession> session, const Protocol::Message &message);      // 库存锁定处理      void handleInventoryLock(std::shared\_ptr<ClientSession> session, const Protocol::Message &message);      void handleInventoryUnlock(std::shared\_ptr<ClientSession> session, const Protocol::Message &message);      // 订单管理处理      void handleOrderCreate(std::shared\_ptr<ClientSession> session, const Protocol::Message &message);      void handleDirectPurchase(std::shared\_ptr<ClientSession> session, const Protocol::Message &message);      void handleOrderGetByUser(std::shared\_ptr<ClientSession> session, const Protocol::Message &message);      void handleOrderGetById(std::shared\_ptr<ClientSession> session, const Protocol::Message &message);      void handleOrderUpdateStatus(std::shared\_ptr<ClientSession> session, const Protocol::Message &message);      void handleOrderGetAll(std::shared\_ptr<ClientSession> session, const Protocol::Message &message);      // 响应发送辅助方法      void sendSuccessResponse(std::shared\_ptr<ClientSession> session, const std::map<std::string, std::string> &data = {});      void sendErrorResponse(std::shared\_ptr<ClientSession> session, const std::string &error);      void sendDataResponse(std::shared\_ptr<ClientSession> session, const std::map<std::string, std::string> &data);      // 数据持久化      void saveUserData();      void loadUserData();      void initializeStore();      void initializeOrderManager();  };  #endif // NETWORK\_SERVER\_H |

#### server.cpp

|  |
| --- |
| #include "server.h"  #include "../user/user.h"  #include "../store/store.h"  #include "../order/ordermanager.h"  #include <iostream>  #include <sstream>  #include <random>  #include <algorithm>  #include <chrono>  // ClientSession 实现  ClientSession::ClientSession(SOCKET socket, const std::string &sid)      : clientSocket(socket), sessionId(sid), userType(Protocol::UserType::CUSTOMER), isActive(true)  {  }  ClientSession::~ClientSession()  {      stopSession();  }  void ClientSession::startSession(NetworkServer \*server)  {      clientThread = std::thread(&ClientSession::sessionLoop, this, server);  }  void ClientSession::stopSession()  {      isActive.store(false);      if (clientSocket != INVALID\_SOCKET)      {          closesocket(clientSocket);          clientSocket = INVALID\_SOCKET;      }      if (clientThread.joinable())      {          clientThread.join();      }  }  void ClientSession::sessionLoop(NetworkServer \*server)  {      std::cout << "客户端会话开始: " << sessionId << std::endl;      while (isActive.load())      {          std::string rawData = receiveRawData();          if (rawData.empty())          {              break;          }          Protocol::Message message = Protocol::Message::deserialize(rawData);          message.sessionId = sessionId; // 确保会话ID正确          // 处理消息          server->handleClientMessage(shared\_from\_this(), message);      }      std::cout << "客户端会话结束: " << sessionId << std::endl;  }  bool ClientSession::sendRawData(const std::string &data)  {      if (clientSocket == INVALID\_SOCKET)      {          return false;      }      // 发送数据长度      uint32\_t dataLength = htonl(static\_cast<uint32\_t>(data.length()));      if (send(clientSocket, reinterpret\_cast<const char \*>(&dataLength), sizeof(dataLength), 0) == SOCKET\_ERROR)      {          return false;      }      // 发送数据内容      int totalSent = 0;      int dataSize = static\_cast<int>(data.length());      while (totalSent < dataSize)      {          int sent = send(clientSocket, data.c\_str() + totalSent, dataSize - totalSent, 0);          if (sent == SOCKET\_ERROR)          {              return false;          }          totalSent += sent;      }      return true;  }  std::string ClientSession::receiveRawData()  {      if (clientSocket == INVALID\_SOCKET)      {          return "";      }      // 接收数据长度      uint32\_t dataLength;      int received = recv(clientSocket, reinterpret\_cast<char \*>(&dataLength), sizeof(dataLength), 0);      if (received <= 0)      {          isActive.store(false);          return "";      }      dataLength = ntohl(dataLength);      // 接收数据内容      std::string data;      data.resize(dataLength);      int totalReceived = 0;      while (totalReceived < static\_cast<int>(dataLength))      {          int received = recv(clientSocket, &data[totalReceived], dataLength - totalReceived, 0);          if (received <= 0)          {              isActive.store(false);              return "";          }          totalReceived += received;      }      return data;  }  bool ClientSession::sendMessage(const Protocol::Message &message)  {      std::string serialized = message.serialize();      return sendRawData(serialized);  }  // NetworkServer 实现  NetworkServer::NetworkServer(int port)      : port(port), serverSocket(INVALID\_SOCKET), isRunning(false),        userFile("./server\_data/users.txt"),        storeDir("./server\_data/store"),        orderDir("./server\_data/orders")  {      // 初始化Winsock      WSADATA wsaData;      int result = WSAStartup(MAKEWORD(2, 2), &wsaData);      if (result != 0)      {          std::cerr << "WSAStartup failed: " << result << std::endl;      }  }  NetworkServer::~NetworkServer()  {      stop();      WSACleanup();  }  bool NetworkServer::start()  {      if (isRunning.load())      {          return true;      }      // 创建socket      serverSocket = socket(AF\_INET, SOCK\_STREAM, IPPROTO\_TCP);      if (serverSocket == INVALID\_SOCKET)      {          std::cerr << "Socket creation failed: " << WSAGetLastError() << std::endl;          return false;      }      // 设置socket选项      int opt = 1;      setsockopt(serverSocket, SOL\_SOCKET, SO\_REUSEADDR, reinterpret\_cast<const char \*>(&opt), sizeof(opt));      // 绑定地址      sockaddr\_in serverAddr;      serverAddr.sin\_family = AF\_INET;      serverAddr.sin\_addr.s\_addr = INADDR\_ANY;      serverAddr.sin\_port = htons(port);      if (bind(serverSocket, reinterpret\_cast<sockaddr \*>(&serverAddr), sizeof(serverAddr)) == SOCKET\_ERROR)      {          std::cerr << "Bind failed: " << WSAGetLastError() << std::endl;          closesocket(serverSocket);          return false;      }      // 开始监听      if (listen(serverSocket, SOMAXCONN) == SOCKET\_ERROR)      {          std::cerr << "Listen failed: " << WSAGetLastError() << std::endl;          closesocket(serverSocket);          return false;      }      // 初始化业务组件      loadUserData();      initializeStore();      initializeOrderManager();      isRunning.store(true);      // 启动接受连接的线程      acceptThread = std::thread(&NetworkServer::acceptLoop, this);      return true;  }  void NetworkServer::stop()  {      if (!isRunning.load())      {          return;      }      isRunning.store(false);      // 关闭服务器socket      if (serverSocket != INVALID\_SOCKET)      {          closesocket(serverSocket);          serverSocket = INVALID\_SOCKET;      }      // 等待接受线程结束      if (acceptThread.joinable())      {          acceptThread.join();      }      // 停止所有客户端会话      {          std::lock\_guard<std::mutex> lock(sessionsMutex);          for (auto &session : sessions)          {              session->stopSession();          }          sessions.clear();      }      // 保存数据      saveUserData();      std::cout << "服务器已停止" << std::endl;  }  void NetworkServer::acceptLoop()  {      while (isRunning.load())      {          SOCKET clientSocket = accept(serverSocket, nullptr, nullptr);          if (clientSocket == INVALID\_SOCKET)          {              if (isRunning.load())              {                  std::cerr << "Accept failed: " << WSAGetLastError() << std::endl;              }              break;          }          // 生成会话ID          std::string sessionId = generateSessionId();          // 创建客户端会话          auto session = std::make\_shared<ClientSession>(clientSocket, sessionId);          {              std::lock\_guard<std::mutex> lock(sessionsMutex);              sessions.push\_back(session);          }          // 启动会话          session->startSession(this);          std::cout << "新客户端连接，会话ID: " << sessionId << std::endl;      }  }  std::string NetworkServer::generateSessionId()  {      static std::random\_device rd;      static std::mt19937 gen(rd());      static std::uniform\_int\_distribution<> dis(100000, 999999);      auto now = std::chrono::system\_clock::now();      auto timestamp = std::chrono::duration\_cast<std::chrono::milliseconds>(now.time\_since\_epoch()).count();      return "SESSION\_" + std::to\_string(timestamp) + "\_" + std::to\_string(dis(gen));  }  std::shared\_ptr<ClientSession> NetworkServer::findSession(const std::string &sessionId)  {      std::lock\_guard<std::mutex> lock(sessionsMutex);      auto it = std::find\_if(sessions.begin(), sessions.end(),                             [&sessionId](const std::shared\_ptr<ClientSession> &session)                             {                                 return session->getSessionId() == sessionId;                             });      return (it != sessions.end()) ? \*it : nullptr;  }  void NetworkServer::removeSession(const std::string &sessionId)  {      std::lock\_guard<std::mutex> lock(sessionsMutex);      sessions.erase(          std::remove\_if(sessions.begin(), sessions.end(),                         [&sessionId](const std::shared\_ptr<ClientSession> &session)                         {                             return session->getSessionId() == sessionId || !session->isSessionActive();                         }),          sessions.end());  }  // 添加查找指定用户名会话的方法  std::shared\_ptr<ClientSession> NetworkServer::findSessionByUsername(const std::string &username)  {      std::lock\_guard<std::mutex> lock(sessionsMutex);      auto it = std::find\_if(sessions.begin(), sessions.end(),                             [&username](const std::shared\_ptr<ClientSession> &session)                             {                                 return session->getUsername() == username && session->isSessionActive();                             });      return (it != sessions.end()) ? \*it : nullptr;  }  // 处理客户端消息  void NetworkServer::handleClientMessage(std::shared\_ptr<ClientSession> session, const Protocol::Message &message)  {      switch (message.type)      {      case Protocol::MessageType::USER\_LOGIN:          handleUserLogin(session, message);          break;      case Protocol::MessageType::USER\_REGISTER:          handleUserRegister(session, message);          break;      case Protocol::MessageType::USER\_LOGOUT:          handleUserLogout(session, message);          break;      case Protocol::MessageType::USER\_GET\_INFO:          handleUserGetInfo(session, message);          break;      case Protocol::MessageType::USER\_UPDATE\_BALANCE:          handleUserUpdateBalance(session, message);          break;      case Protocol::MessageType::USER\_CHANGE\_PASSWORD:          handleUserChangePassword(session, message);          break;      case Protocol::MessageType::PRODUCT\_GET\_ALL:          handleProductGetAll(session, message);          break;      case Protocol::MessageType::PRODUCT\_SEARCH:          handleProductSearch(session, message);          break;      case Protocol::MessageType::CART\_GET:          handleCartGet(session, message);          break;      case Protocol::MessageType::CART\_ADD\_ITEM:          handleCartAddItem(session, message);          break;      case Protocol::MessageType::CART\_REMOVE\_ITEM:          handleCartRemoveItem(session, message);          break;      case Protocol::MessageType::CART\_UPDATE\_ITEM:          handleCartUpdateItem(session, message);          break;      case Protocol::MessageType::CART\_CLEAR:          handleCartClear(session, message);          break;      case Protocol::MessageType::PRODUCT\_ADD:          handleProductAdd(session, message);          break;      case Protocol::MessageType::PRODUCT\_MANAGE\_PRICE:          handleProductManagePrice(session, message);          break;      case Protocol::MessageType::PRODUCT\_MANAGE\_QUANTITY:          handleProductManageQuantity(session, message);          break;      case Protocol::MessageType::PRODUCT\_MANAGE\_DISCOUNT:          handleProductManageDiscount(session, message);          break;      case Protocol::MessageType::PRODUCT\_APPLY\_CATEGORY\_DISCOUNT:          handleProductApplyCategoryDiscount(session, message);          break;      case Protocol::MessageType::ORDER\_CREATE:          handleOrderCreate(session, message);          break;      case Protocol::MessageType::ORDER\_DIRECT\_PURCHASE:          handleDirectPurchase(session, message);          break;      case Protocol::MessageType::ORDER\_GET\_BY\_USER:          handleOrderGetByUser(session, message);          break;      case Protocol::MessageType::INVENTORY\_LOCK:          handleInventoryLock(session, message);          break;      case Protocol::MessageType::INVENTORY\_UNLOCK:          handleInventoryUnlock(session, message);          break;      default:          sendErrorResponse(session, "不支持的消息类型");          break;      }  }  // 用户登录处理  void NetworkServer::handleUserLogin(std::shared\_ptr<ClientSession> session, const Protocol::Message &message)  {      std::string username = message.getData("username");      std::string password = message.getData("password");      // 查找用户      User \*user = User::findUser(users, username);      if (!user || user->getPassword() != password)      {          sendErrorResponse(session, "用户名或密码错误");          return;      } // 检查该用户是否已经在其他客户端登录      std::shared\_ptr<ClientSession> existingSession = findSessionByUsername(username);      if (existingSession && existingSession->getSessionId() != session->getSessionId())      {          sendErrorResponse(session, "该用户已经登录");          std::cout << "用户 " << username << " 尝试重复登录，已拒绝" << std::endl;          return;      }      // 设置会话用户信息      session->setUsername(username);      std::string userTypeStr = user->getUserType();      if (userTypeStr == "customer")      {          session->setUserType(Protocol::UserType::CUSTOMER);      }      else if (userTypeStr == "seller")      {          session->setUserType(Protocol::UserType::SELLER);      }      else if (userTypeStr == "admin")      {          session->setUserType(Protocol::UserType::ADMIN);      }      // 准备响应数据      std::map<std::string, std::string> responseData;      responseData["sessionId"] = session->getSessionId();      responseData["userData"] = convertToUserData(user).serialize();      sendSuccessResponse(session, responseData);      std::cout << "用户登录成功: " << username << std::endl;  }  // 用户注册处理  void NetworkServer::handleUserRegister(std::shared\_ptr<ClientSession> session, const Protocol::Message &message)  {      std::string username = message.getData("username");      std::string password = message.getData("password");      int userTypeInt = std::stoi(message.getData("userType"));      // 检查用户名是否已存在      if (User::isUsernameExists(users, username))      {          sendErrorResponse(session, "用户名已存在");          return;      }      // 创建新用户      User \*newUser = nullptr;      switch (static\_cast<Protocol::UserType>(userTypeInt))      {      case Protocol::UserType::CUSTOMER:          newUser = new Customer("./server\_data/carts");          break;      case Protocol::UserType::SELLER:          newUser = new Seller();          break;      case Protocol::UserType::ADMIN:          newUser = new Admin();          break;      }      if (newUser)      {          newUser->setUsername(username);          newUser->setPassword(password);          newUser->setBalance(0.0);          users.push\_back(newUser);          saveUserData();          sendSuccessResponse(session);          std::cout << "新用户注册成功: " << username << std::endl;      }      else      {          sendErrorResponse(session, "创建用户失败");      }  }  // 用户登出处理  void NetworkServer::handleUserLogout(std::shared\_ptr<ClientSession> session, const Protocol::Message &message)  {      session->setUsername("");      session->setUserType(Protocol::UserType::CUSTOMER);      sendSuccessResponse(session);      std::cout << "用户登出成功" << std::endl;  }  // 获取用户信息处理  void NetworkServer::handleUserGetInfo(std::shared\_ptr<ClientSession> session, const Protocol::Message &message)  {      std::string username = session->getUsername();      if (username.empty())      {          sendErrorResponse(session, "用户未登录");          return;      }      // 查找用户      User \*user = User::findUser(users, username);      if (!user)      {          sendErrorResponse(session, "用户不存在");          return;      }      // 准备响应数据      std::map<std::string, std::string> responseData;      responseData["userData"] = convertToUserData(user).serialize();      sendDataResponse(session, responseData);      std::cout << "用户 " << username << " 获取用户信息成功" << std::endl;  }  // 用户余额更新处理  void NetworkServer::handleUserUpdateBalance(std::shared\_ptr<ClientSession> session, const Protocol::Message &message)  {      std::string username = session->getUsername();      if (username.empty())      {          sendErrorResponse(session, "用户未登录");          return;      }      // 获取充值金额      std::string amountStr = message.getData("amount");      double amount = 0.0;      try      {          amount = std::stod(amountStr);      }      catch (const std::exception &e)      {          sendErrorResponse(session, "无效的充值金额");          return;      }      if (amount <= 0)      {          sendErrorResponse(session, "充值金额必须大于0");          return;      }      // 查找用户      User \*user = User::findUser(users, username);      if (!user)      {          sendErrorResponse(session, "用户不存在");          return;      }      // 执行充值      if (user->deposit(amount))      {          // 保存用户数据          saveUserData();          // 返回成功响应，包含新的余额          std::map<std::string, std::string> responseData;          responseData["newBalance"] = std::to\_string(user->checkBalance());          sendSuccessResponse(session, responseData);          std::cout << "用户 " << username << " 充值成功，金额: " << amount                    << "，当前余额: " << user->checkBalance() << std::endl;      }      else      {          sendErrorResponse(session, "充值失败");      }  }  // 用户密码修改处理  void NetworkServer::handleUserChangePassword(std::shared\_ptr<ClientSession> session, const Protocol::Message &message)  {      std::string username = session->getUsername();      if (username.empty())      {          sendErrorResponse(session, "用户未登录");          return;      }      // 获取旧密码和新密码      std::string oldPassword = message.getData("oldPassword");      std::string newPassword = message.getData("newPassword");      if (oldPassword.empty() || newPassword.empty())      {          sendErrorResponse(session, "密码不能为空");          return;      }      // 查找用户      User \*user = User::findUser(users, username);      if (!user)      {          sendErrorResponse(session, "用户不存在");          return;      }      // 执行密码修改      if (user->changePassword(oldPassword, newPassword))      {          // 保存用户数据          saveUserData();          sendSuccessResponse(session);          std::cout << "用户 " << username << " 密码修改成功" << std::endl;      }      else      {          sendErrorResponse(session, "原密码错误");      }  }  // 商品获取处理  void NetworkServer::handleProductGetAll(std::shared\_ptr<ClientSession> session, const Protocol::Message &message)  {      std::vector<Product \*> allProducts = store->getProducts();      std::vector<Protocol::ProductData> productDataList = convertToProductDataList(allProducts);      std::map<std::string, std::string> responseData;      responseData["count"] = std::to\_string(productDataList.size());      for (size\_t i = 0; i < productDataList.size(); ++i)      {          responseData["product\_" + std::to\_string(i)] = productDataList[i].serialize();      }      sendDataResponse(session, responseData);  }  // 商品搜索处理  void NetworkServer::handleProductSearch(std::shared\_ptr<ClientSession> session, const Protocol::Message &message)  {      std::string keyword = message.getData("keyword");      if (keyword.empty())      {          sendErrorResponse(session, "搜索关键词不能为空");          return;      }      // 使用Store的搜索功能      std::vector<Product \*> searchResults = store->searchProductsByName(keyword);      std::vector<Protocol::ProductData> productDataList = convertToProductDataList(searchResults);      std::map<std::string, std::string> responseData;      responseData["count"] = std::to\_string(productDataList.size());      for (size\_t i = 0; i < productDataList.size(); ++i)      {          responseData["product\_" + std::to\_string(i)] = productDataList[i].serialize();      }      sendDataResponse(session, responseData);      std::cout << "商品搜索完成，关键词: \"" << keyword << "\"，找到 " << productDataList.size() << " 个结果" << std::endl;  }  // 购物车获取处理  void NetworkServer::handleCartGet(std::shared\_ptr<ClientSession> session, const Protocol::Message &message)  {      std::string username = session->getUsername();      if (username.empty())      {          sendErrorResponse(session, "用户未登录");          return;      }      // 找到用户      User \*user = User::findUser(users, username);      Customer \*customer = dynamic\_cast<Customer \*>(user);      if (!customer)      {          sendErrorResponse(session, "不是消费者用户");          return;      }      // 获取购物车数据      std::vector<Protocol::CartItemData> cartData;      for (const auto &item : customer->shoppingCartItems)      {          Protocol::CartItemData cartItem;          cartItem.productId = item.productId;          cartItem.productName = item.productName;          cartItem.quantity = item.quantity;          cartItem.priceAtAddition = item.priceAtAddition;          cartItem.sellerUsername = item.sellerUsername;          cartData.push\_back(cartItem);      }      std::map<std::string, std::string> responseData;      responseData["count"] = std::to\_string(cartData.size());      for (size\_t i = 0; i < cartData.size(); ++i)      {          responseData["item\_" + std::to\_string(i)] = cartData[i].serialize();      }      sendDataResponse(session, responseData);  }  // 购物车添加商品处理  void NetworkServer::handleCartAddItem(std::shared\_ptr<ClientSession> session, const Protocol::Message &message)  {      std::string username = session->getUsername();      if (username.empty())      {          sendErrorResponse(session, "用户未登录");          return;      }      // 找到用户      User \*user = User::findUser(users, username);      Customer \*customer = dynamic\_cast<Customer \*>(user);      if (!customer)      {          sendErrorResponse(session, "不是消费者用户");          return;      }      std::string productId = message.getData("productId");      int quantity = std::stoi(message.getData("quantity"));      // 查找商品      Product \*product = store->findProductByName(productId);      if (!product)      {          sendErrorResponse(session, "商品不存在");          return;      }      // 检查库存      if (product->getQuantity() < quantity)      {          sendErrorResponse(session, "库存不足");          return;      } // 添加到购物车 - 使用正确的方法名和参数      if (customer->addToCart(\*product, quantity))      {          // 添加成功，需要将商品信息保存到购物车文件          customer->saveCartToFile();          sendSuccessResponse(session);          std::cout << "商品已添加到购物车: " << productId << ", 数量: " << quantity << std::endl;      }      else      {          sendErrorResponse(session, "添加商品到购物车失败");      }  }  // 购物车移除商品处理  void NetworkServer::handleCartRemoveItem(std::shared\_ptr<ClientSession> session, const Protocol::Message &message)  {      std::string username = session->getUsername();      if (username.empty())      {          sendErrorResponse(session, "用户未登录");          return;      }      // 找到用户      User \*user = User::findUser(users, username);      Customer \*customer = dynamic\_cast<Customer \*>(user);      if (!customer)      {          sendErrorResponse(session, "不是消费者用户");          return;      }      std::string productId = message.getData("productId"); // 从购物车移除商品 - 使用正确的方法名      if (customer->removeCartItem(productId))      {          customer->saveCartToFile();          sendSuccessResponse(session);          std::cout << "商品已从购物车移除: " << productId << std::endl;      }      else      {          sendErrorResponse(session, "移除商品失败");      }  }  // 购物车清空处理  void NetworkServer::handleCartClear(std::shared\_ptr<ClientSession> session, const Protocol::Message &message)  {      std::string username = session->getUsername();      if (username.empty())      {          sendErrorResponse(session, "用户未登录");          return;      }      // 找到用户      User \*user = User::findUser(users, username);      Customer \*customer = dynamic\_cast<Customer \*>(user);      if (!customer)      {          sendErrorResponse(session, "不是消费者用户");          return;      }      // 清空购物车 - 使用正确的方法名      customer->clearCartAndFile();      sendSuccessResponse(session);      std::cout << "购物车已清空，用户: " << username << std::endl;  }  // 购物车商品数量更新处理  void NetworkServer::handleCartUpdateItem(std::shared\_ptr<ClientSession> session, const Protocol::Message &message)  {      std::string username = session->getUsername();      if (username.empty())      {          sendErrorResponse(session, "用户未登录");          return;      }      // 找到用户      User \*user = User::findUser(users, username);      Customer \*customer = dynamic\_cast<Customer \*>(user);      if (!customer)      {          sendErrorResponse(session, "不是消费者用户");          return;      }      std::string productId = message.getData("productId");      int newQuantity = 0;      try      {          newQuantity = std::stoi(message.getData("quantity"));      }      catch (const std::exception &e)      {          sendErrorResponse(session, "无效的数量");          return;      }      // 检查数量是否有效      if (newQuantity < 0)      {          sendErrorResponse(session, "数量不能为负数");          return;      }      // 如果新数量为0，则移除商品      if (newQuantity == 0)      {          if (customer->removeCartItem(productId))          {              customer->saveCartToFile();              sendSuccessResponse(session);              std::cout << "商品已从购物车移除: " << productId << std::endl;          }          else          {              sendErrorResponse(session, "移除商品失败");          }          return;      }      // 查找商品以检查库存      Product \*product = store->findProductByName(productId);      if (!product)      {          sendErrorResponse(session, "商品不存在");          return;      }      // 检查库存      if (product->getQuantity() < newQuantity)      {          sendErrorResponse(session, "库存不足");          return;      }      // 更新购物车商品数量      if (customer->updateCartItemQuantity(productId, newQuantity, \*store))      {          customer->saveCartToFile();          sendSuccessResponse(session);          std::cout << "购物车商品数量已更新: " << productId << ", 新数量: " << newQuantity << std::endl;      }      else      {          sendErrorResponse(session, "更新购物车商品数量失败");      }  }  // 添加商品处理  void NetworkServer::handleProductAdd(std::shared\_ptr<ClientSession> session, const Protocol::Message &message)  {      std::string username = session->getUsername();      if (username.empty())      {          sendErrorResponse(session, "用户未登录");          return;      }      // 找到用户      User \*user = User::findUser(users, username);      Seller \*seller = dynamic\_cast<Seller \*>(user);      if (!seller)      {          sendErrorResponse(session, "不是商家用户");          return;      }      // 解析商品数据      std::string productDataStr = message.getData("productData");      Protocol::ProductData productData = Protocol::ProductData::deserialize(productDataStr);    // 根据商品类型创建具体的商品对象，使用从客户端发送的attributes数据      bool success = false;      if (productData.type == "Book")      {          // 对于图书，从attributes中获取作者和ISBN          std::string author = "Unknown Author";          std::string isbn = "N/A";          auto authorIt = productData.attributes.find("author");          auto isbnIt = productData.attributes.find("isbn");          if (authorIt != productData.attributes.end()) {              author = authorIt->second;          }          if (isbnIt != productData.attributes.end()) {              isbn = isbnIt->second;          }            success = store->createBook(seller, productData.name, productData.description,                                      productData.price, productData.quantity,                                      author, isbn);      }      else if (productData.type == "Clothing")      {          // 对于服装，从attributes中获取尺码和颜色          std::string size = "M";          std::string color = "Default";          auto sizeIt = productData.attributes.find("size");          auto colorIt = productData.attributes.find("color");          if (sizeIt != productData.attributes.end()) {              size = sizeIt->second;          }          if (colorIt != productData.attributes.end()) {              color = colorIt->second;          }            success = store->createClothing(seller, productData.name, productData.description,                                          productData.price, productData.quantity,                                          size, color);      }      else if (productData.type == "Food")      {          // 对于食品，从attributes中获取过期日期          std::string expirationDate = "2025-12-31";          auto expIt = productData.attributes.find("expirationDate");          if (expIt != productData.attributes.end()) {              expirationDate = expIt->second;          }            success = store->createFood(seller, productData.name, productData.description,                                      productData.price, productData.quantity,                                      expirationDate);      }      else      {          // 对于通用商品，从attributes中获取类别标签          std::string categoryTag = productData.type;          auto categoryIt = productData.attributes.find("categoryTag");          if (categoryIt != productData.attributes.end()) {              categoryTag = categoryIt->second;          }            // 通用商品类型始终设为"Generic"          success = store->createGenericProduct(seller, productData.name, productData.description,                                                productData.price, productData.quantity,                                                categoryTag);      }      // 检查创建结果      if (success)      {          sendSuccessResponse(session);          std::cout << "商品添加成功: " << productData.name << "，商家: " << seller->getUsername() << std::endl;      }      else      {          sendErrorResponse(session, "添加商品失败");      }  }  // 订单创建处理  void NetworkServer::handleOrderCreate(std::shared\_ptr<ClientSession> session, const Protocol::Message &message)  {      std::string username = session->getUsername();      if (username.empty())      {          sendErrorResponse(session, "用户未登录");          return;      }      // 找到用户      User \*user = User::findUser(users, username);      Customer \*customer = dynamic\_cast<Customer \*>(user);      if (!customer)      {          sendErrorResponse(session, "不是消费者用户");          return;      }      // 检查购物车是否为空      if (customer->shoppingCartItems.empty())      {          sendErrorResponse(session, "购物车为空");          return;      }      // 创建订单      Order newOrder(customer->getUsername());      for (const auto &cartItem : customer->shoppingCartItems)      { // 将CartItem转换为OrderItem          OrderItem orderItem(cartItem.productId, cartItem.productName,                              cartItem.quantity, cartItem.priceAtAddition,                              cartItem.sellerUsername);          newOrder.addItem(orderItem);      }      // 提交订单给订单管理器      auto submittedOrder = orderManager->submitOrderRequest(newOrder);      if (submittedOrder)      {          // 立即处理订单（同步处理）          orderManager->processNextOrder(\*store, users);          // 清空购物车 - 使用正确的方法名          customer->clearCartAndFile();          std::map<std::string, std::string> responseData;          responseData["orderId"] = submittedOrder->getOrderId();          sendSuccessResponse(session, responseData);          std::cout << "订单创建成功: " << submittedOrder->getOrderId() << std::endl;      }      else      {          sendErrorResponse(session, "订单创建失败");      }  }  // 直接购买处理  void NetworkServer::handleDirectPurchase(std::shared\_ptr<ClientSession> session, const Protocol::Message &message)  {      std::string username = session->getUsername();      if (username.empty())      {          sendErrorResponse(session, "用户未登录");          return;      }      // 找到用户      User \*user = User::findUser(users, username);      Customer \*customer = dynamic\_cast<Customer \*>(user);      if (!customer)      {          sendErrorResponse(session, "不是消费者用户");          return;      }      // 获取请求参数      std::string productId = message.getData("productId");      int quantity = std::stoi(message.getData("quantity"));      // 查找商品      Product \*product = store->findProductByName(productId);      if (!product)      {          sendErrorResponse(session, "商品不存在");          return;      }      // 检查库存      if (product->getQuantity() < quantity)      {          sendErrorResponse(session, "库存不足");          return;      }      // 创建直接购买订单（不涉及购物车）      Order newOrder(customer->getUsername());      OrderItem orderItem(product->getName(), product->getName(),                          quantity, product->getPrice(),                          product->getSellerUsername());      newOrder.addItem(orderItem);      // 提交订单给订单管理器      auto submittedOrder = orderManager->submitOrderRequest(newOrder);      if (submittedOrder)      {          // 立即处理订单（同步处理）          orderManager->processNextOrder(\*store, users);          std::map<std::string, std::string> responseData;          responseData["orderId"] = submittedOrder->getOrderId();          sendSuccessResponse(session, responseData);          std::cout << "直接购买订单创建成功: " << submittedOrder->getOrderId() << std::endl;      }      else      {          sendErrorResponse(session, "直接购买订单创建失败");      }  }  // 获取用户订单处理  void NetworkServer::handleOrderGetByUser(std::shared\_ptr<ClientSession> session, const Protocol::Message &message)  {      std::string username = session->getUsername();      if (username.empty())      {          sendErrorResponse(session, "用户未登录");          return;      }      // 获取用户的所有订单      std::vector<std::shared\_ptr<Order>> pendingOrders, completedOrders;      orderManager->getUserOrders(username, pendingOrders, completedOrders);      // 合并所有订单      std::vector<std::shared\_ptr<Order>> allOrders;      allOrders.insert(allOrders.end(), pendingOrders.begin(), pendingOrders.end());      allOrders.insert(allOrders.end(), completedOrders.begin(), completedOrders.end());      // 转换为协议数据      std::vector<Protocol::OrderData> orderDataList;      for (const auto &order : allOrders)      {          Protocol::OrderData orderData;          orderData.orderId = order->getOrderId();          orderData.customerUsername = order->getCustomerUsername();          orderData.totalAmount = order->getTotalAmount();          orderData.timestamp = order->getTimestamp();          // 转换订单状态          std::string status = order->getStatus();          if (status == "PENDING")          {              orderData.status = Protocol::OrderStatus::PENDING;          }          else if (status == "COMPLETED")          {              orderData.status = Protocol::OrderStatus::COMPLETED;          }          else if (status == "CANCELLED\_STOCK")          {              orderData.status = Protocol::OrderStatus::CANCELLED\_STOCK;          }          else if (status == "CANCELLED\_FUNDS")          {              orderData.status = Protocol::OrderStatus::CANCELLED\_FUNDS;          }          else if (status == "CANCELLED\_USER")          {              orderData.status = Protocol::OrderStatus::CANCELLED\_USER;          }          else          {              orderData.status = Protocol::OrderStatus::PENDING;          }          // 转换订单项          for (const auto &item : order->getItems())          {              Protocol::CartItemData cartItem;              cartItem.productId = item.productId;              cartItem.productName = item.productName;              cartItem.quantity = item.quantity;              cartItem.priceAtAddition = item.priceAtPurchase;              cartItem.sellerUsername = item.sellerUsername;              orderData.items.push\_back(cartItem);          }          orderDataList.push\_back(orderData);      }      // 发送响应      std::map<std::string, std::string> responseData;      responseData["count"] = std::to\_string(orderDataList.size());      for (size\_t i = 0; i < orderDataList.size(); ++i)      {          responseData["order\_" + std::to\_string(i)] = orderDataList[i].serialize();      }      sendDataResponse(session, responseData);  }  // 商家管理处理函数  void NetworkServer::handleProductManagePrice(std::shared\_ptr<ClientSession> session, const Protocol::Message &message)  {      std::string username = session->getUsername();      if (username.empty())      {          sendErrorResponse(session, "用户未登录");          return;      }      // 找到用户      User \*user = User::findUser(users, username);      Seller \*seller = dynamic\_cast<Seller \*>(user);      if (!seller)      {          sendErrorResponse(session, "不是商家用户");          return;      }      // 获取参数      std::string productName = message.getData("productName");      double newPrice = 0.0;      try      {          newPrice = std::stod(message.getData("newPrice"));      }      catch (const std::exception &e)      {          sendErrorResponse(session, "无效的价格");          return;      }      // 调用Store方法修改价格      if (store->manageProductPrice(seller, productName, newPrice))      {          sendSuccessResponse(session);          std::cout << "商家 " << username << " 修改商品 " << productName << " 价格为 " << newPrice << std::endl;      }      else      {          sendErrorResponse(session, "修改价格失败");      }  }  void NetworkServer::handleProductManageQuantity(std::shared\_ptr<ClientSession> session, const Protocol::Message &message)  {      std::string username = session->getUsername();      if (username.empty())      {          sendErrorResponse(session, "用户未登录");          return;      }      // 找到用户      User \*user = User::findUser(users, username);      Seller \*seller = dynamic\_cast<Seller \*>(user);      if (!seller)      {          sendErrorResponse(session, "不是商家用户");          return;      }      // 获取参数      std::string productName = message.getData("productName");      int newQuantity = 0;      try      {          newQuantity = std::stoi(message.getData("newQuantity"));      }      catch (const std::exception &e)      {          sendErrorResponse(session, "无效的数量");          return;      }      // 调用Store方法修改库存      if (store->manageProductQuantity(seller, productName, newQuantity))      {          sendSuccessResponse(session);          std::cout << "商家 " << username << " 修改商品 " << productName << " 库存为 " << newQuantity << std::endl;      }      else      {          sendErrorResponse(session, "修改库存失败");      }  }  void NetworkServer::handleProductManageDiscount(std::shared\_ptr<ClientSession> session, const Protocol::Message &message)  {      std::string username = session->getUsername();      if (username.empty())      {          sendErrorResponse(session, "用户未登录");          return;      }      // 找到用户      User \*user = User::findUser(users, username);      Seller \*seller = dynamic\_cast<Seller \*>(user);      if (!seller)      {          sendErrorResponse(session, "不是商家用户");          return;      }      // 获取参数      std::string productName = message.getData("productName");      double newDiscount = 0.0;      try      {          newDiscount = std::stod(message.getData("newDiscount"));      }      catch (const std::exception &e)      {          sendErrorResponse(session, "无效的折扣");          return;      }      // 调用Store方法修改折扣      if (store->manageProductDiscount(seller, productName, newDiscount))      {          sendSuccessResponse(session);          std::cout << "商家 " << username << " 修改商品 " << productName << " 折扣为 " << (newDiscount \* 100) << "%" << std::endl;      }      else      {          sendErrorResponse(session, "修改折扣失败");      }  }  void NetworkServer::handleProductApplyCategoryDiscount(std::shared\_ptr<ClientSession> session, const Protocol::Message &message)  {      std::string username = session->getUsername();      if (username.empty())      {          sendErrorResponse(session, "用户未登录");          return;      }      // 找到用户      User \*user = User::findUser(users, username);      Seller \*seller = dynamic\_cast<Seller \*>(user);      if (!seller)      {          sendErrorResponse(session, "不是商家用户");          return;      }      // 获取参数      std::string category = message.getData("category");      double discount = 0.0;      try      {          discount = std::stod(message.getData("discount"));      }      catch (const std::exception &e)      {          sendErrorResponse(session, "无效的折扣");          return;      }      // 调用Store方法应用类别折扣      if (store->applyCategoryDiscount(seller, category, discount))      {          sendSuccessResponse(session);          std::cout << "商家 " << username << " 为类别 " << category << " 应用了 " << (discount \* 100) << "% 折扣" << std::endl;      }      else      {          sendErrorResponse(session, "应用类别折扣失败");      }  }  // 库存锁定处理  void NetworkServer::handleInventoryLock(std::shared\_ptr<ClientSession> session, const Protocol::Message &message)  {      std::string username = session->getUsername();      if (username.empty())      {          sendErrorResponse(session, "用户未登录");          return;      }      // 检查是否有单个商品的锁定请求      std::string productId = message.getData("productId");      if (!productId.empty())      {          // 单个商品锁定          int quantity = 0;          try          {              quantity = std::stoi(message.getData("quantity"));          }          catch (const std::exception &e)          {              sendErrorResponse(session, "无效的数量");              return;          }          // 锁定库存          if (store->lockInventory(productId, quantity))          {              sendSuccessResponse(session);              std::cout << "用户 " << username << " 锁定库存成功: " << productId << ", 数量: " << quantity << std::endl;          }          else          {              sendErrorResponse(session, "库存锁定失败");          }      }      else      {          // 多个商品锁定（购物车）          std::string itemCountStr = message.getData("itemCount");          if (itemCountStr.empty())          {              sendErrorResponse(session, "缺少商品数量信息");              return;          }          int itemCount = 0;          try          {              itemCount = std::stoi(itemCountStr);          }          catch (const std::exception &e)          {              sendErrorResponse(session, "无效的商品数量");              return;          }          // 收集所有商品和数量          std::vector<std::pair<std::string, int>> items;          for (int i = 0; i < itemCount; ++i)          {              std::string prodId = message.getData("productId\_" + std::to\_string(i));              std::string quantityStr = message.getData("quantity\_" + std::to\_string(i));              if (prodId.empty() || quantityStr.empty())              {                  sendErrorResponse(session, "商品信息不完整");                  return;              }              int qty = 0;              try              {                  qty = std::stoi(quantityStr);              }              catch (const std::exception &e)              {                  sendErrorResponse(session, "无效的商品数量");                  return;              }              items.push\_back({prodId, qty});          }          // 锁定所有商品的库存          bool allLocked = true;          for (const auto &item : items)          {              if (!store->lockInventory(item.first, item.second))              {                  allLocked = false;                  // 回滚已锁定的库存                  for (int j = 0; j < &item - &items[0]; ++j)                  {                      store->unlockInventory(items[j].first, items[j].second);                  }                  break;              }          }          if (allLocked)          {              sendSuccessResponse(session);              std::cout << "用户 " << username << " 购物车库存锁定成功，商品数量: " << itemCount << std::endl;          }          else          {              sendErrorResponse(session, "购物车库存锁定失败");          }      }  }  void NetworkServer::handleInventoryUnlock(std::shared\_ptr<ClientSession> session, const Protocol::Message &message)  {      std::string username = session->getUsername();      if (username.empty())      {          sendErrorResponse(session, "用户未登录");          return;      }      // 检查是否有单个商品的解锁请求      std::string productId = message.getData("productId");      if (!productId.empty())      {          // 单个商品解锁          int quantity = 0;          try          {              quantity = std::stoi(message.getData("quantity"));          }          catch (const std::exception &e)          {              sendErrorResponse(session, "无效的数量");              return;          }          // 解锁库存          if (store->unlockInventory(productId, quantity))          {              sendSuccessResponse(session);              std::cout << "用户 " << username << " 解锁库存成功: " << productId << ", 数量: " << quantity << std::endl;          }          else          {              sendErrorResponse(session, "库存解锁失败");          }      }      else      {          // 多个商品解锁（购物车）          std::string itemCountStr = message.getData("itemCount");          if (itemCountStr.empty())          {              sendErrorResponse(session, "缺少商品数量信息");              return;          }          int itemCount = 0;          try          {              itemCount = std::stoi(itemCountStr);          }          catch (const std::exception &e)          {              sendErrorResponse(session, "无效的商品数量");              return;          }          // 收集所有商品和数量并解锁          for (int i = 0; i < itemCount; ++i)          {              std::string prodId = message.getData("productId\_" + std::to\_string(i));              std::string quantityStr = message.getData("quantity\_" + std::to\_string(i));              if (!prodId.empty() && !quantityStr.empty())              {                  try                  {                      int qty = std::stoi(quantityStr);                      store->unlockInventory(prodId, qty);                  }                  catch (const std::exception &e)                  {                      // 继续处理其他商品，即使某个商品解锁失败                      std::cerr << "解锁商品 " << prodId << " 失败: " << e.what() << std::endl;                  }              }          }          sendSuccessResponse(session);          std::cout << "用户 " << username << " 购物车库存解锁完成，商品数量: " << itemCount << std::endl;      }  }  // 响应发送辅助方法  void NetworkServer::sendSuccessResponse(std::shared\_ptr<ClientSession> session, const std::map<std::string, std::string> &data)  {      Protocol::Message response(Protocol::MessageType::RESPONSE\_SUCCESS, session->getSessionId());      for (const auto &pair : data)      {          response.setData(pair.first, pair.second);      }      session->sendMessage(response);  }  void NetworkServer::sendErrorResponse(std::shared\_ptr<ClientSession> session, const std::string &error)  {      Protocol::Message response(Protocol::MessageType::RESPONSE\_ERROR, session->getSessionId());      response.setData("error", error);      session->sendMessage(response);  }  void NetworkServer::sendDataResponse(std::shared\_ptr<ClientSession> session, const std::map<std::string, std::string> &data)  {      Protocol::Message response(Protocol::MessageType::RESPONSE\_DATA, session->getSessionId());      for (const auto &pair : data)      {          response.setData(pair.first, pair.second);      }      session->sendMessage(response);  }  // 数据转换方法  Protocol::UserData NetworkServer::convertToUserData(const User \*user)  {      Protocol::UserData userData;      userData.username = user->getUsername();      userData.password = user->getPassword();      userData.balance = user->checkBalance();      std::string userType = user->getUserType();      if (userType == "customer")      {          userData.userType = Protocol::UserType::CUSTOMER;      }      else if (userType == "seller")      {          userData.userType = Protocol::UserType::SELLER;      }      else if (userType == "admin")      {          userData.userType = Protocol::UserType::ADMIN;      }      return userData;  }  Protocol::ProductData NetworkServer::convertToProductData(const Product \*product)  {      Protocol::ProductData productData;      productData.id = product->getName(); // 使用名称作为ID      productData.name = product->getName();      productData.description = product->getDescription();      // 修复Generic产品显示问题：使用getUserCategory()显示实际类别名称      std::string userCategory = product->getUserCategory();      if (!userCategory.empty())      {          productData.type = userCategory; // 显示实际类别名称（如自定义标签）      }      else      {          productData.type = product->getType(); // 回退到基本类型      }      // 修复价格显示：发送折扣后的实际价格      productData.price = product->getPrice();                 // 使用 getPrice() 获取折扣后价格      productData.originalPrice = product->getOriginalPrice(); // 添加原价字段      productData.quantity = product->getQuantity();      productData.discountRate = product->getDiscountRate();      productData.sellerUsername = product->getSellerUsername();      return productData;  }  std::vector<Protocol::ProductData> NetworkServer::convertToProductDataList(const std::vector<Product \*> &products)  {      std::vector<Protocol::ProductData> result;      for (const Product \*product : products)      {          result.push\_back(convertToProductData(product));      }      return result;  }  // 数据持久化方法  void NetworkServer::loadUserData()  {      users = User::loadUsersFromFile(userFile);      std::cout << "已加载 " << users.size() << " 个用户数据" << std::endl;  }  void NetworkServer::saveUserData()  {      User::saveUsersToFile(users, userFile);      std::cout << "用户数据已保存" << std::endl;  }  void NetworkServer::initializeStore()  {      store = std::make\_unique<Store>(storeDir);      store->loadAllProducts();      std::cout << "商店数据已初始化" << std::endl;  }  void NetworkServer::initializeOrderManager()  {      orderManager = std::make\_unique<OrderManager>(orderDir);      // 注意：这里不启动处理线程，因为服务端会同步处理订单      std::cout << "订单管理器已初始化" << std::endl;  } |

#### client.h

|  |
| --- |
| #ifndef NETWORK\_CLIENT\_H  #define NETWORK\_CLIENT\_H  #include "../network/protocol.h"  #include <winsock2.h>  #include <ws2tcpip.h>  #include <string>  #include <functional>  #include <thread>  #include <mutex>  #include <condition\_variable>  #include <queue>  #include <atomic>  #pragma comment(lib, "ws2\_32.lib")  class NetworkClient  {  private:      SOCKET clientSocket;      std::string serverAddress;      int serverPort;      std::string sessionId;      // 消息处理      std::thread receiveThread;      std::atomic<bool> isConnected;      std::atomic<bool> shouldStop;      // 消息队列      std::queue<Protocol::Message> responseQueue;      mutable std::mutex queueMutex;      std::condition\_variable queueCondition;      // 回调函数      std::function<void(const Protocol::Message &)> messageCallback;      // 内部方法      void receiveLoop();      bool sendRawData(const std::string &data);      std::string receiveRawData();  public:      NetworkClient(const std::string &address = "127.0.0.1", int port = 8888);      ~NetworkClient();      // 连接管理      bool connect();      void disconnect();      bool isConnectionActive() const { return isConnected.load(); }      // 消息发送      bool sendMessage(const Protocol::Message &message); // 消息接收      Protocol::Message waitForResponse(Protocol::MessageType expectedType, int timeoutMs = 2000);      Protocol::Message waitForResponseBlocking(Protocol::MessageType expectedType); // 无超时限制版本      bool hasResponse() const;      Protocol::Message getNextResponse();      // 设置消息回调      void setMessageCallback(std::function<void(const Protocol::Message &)> callback);      // 会话管理      void setSessionId(const std::string &sid) { sessionId = sid; }      std::string getSessionId() const { return sessionId; } // 用户操作      bool login(const std::string &username, const std::string &password, Protocol::UserData &userData);      bool loginWithError(const std::string &username, const std::string &password, Protocol::UserData &userData, std::string &errorMessage);      bool registerUser(const std::string &username, const std::string &password, Protocol::UserType userType);      bool logout();      bool getUserInfo(Protocol::UserData &userData);      bool updateBalance(double amount);      bool changePassword(const std::string &oldPassword, const std::string &newPassword);      // 商品操作      bool getAllProducts(std::vector<Protocol::ProductData> &products);      bool searchProducts(const std::string &keyword, std::vector<Protocol::ProductData> &products);      bool getProductById(const std::string &productId, Protocol::ProductData &product);      bool addProduct(const Protocol::ProductData &product);      bool updateProduct(const Protocol::ProductData &product);      bool deleteProduct(const std::string &productId);      bool updateProductStock(const std::string &productId, int newStock);      // 购物车操作      bool getCart(std::vector<Protocol::CartItemData> &cartItems);      bool addToCart(const std::string &productId, int quantity);      bool updateCartItem(const std::string &productId, int newQuantity);      bool removeFromCart(const std::string &productId);      bool clearCart(); // 订单操作      bool createOrder(const std::vector<Protocol::CartItemData> &items, std::string &orderId);      bool createDirectOrder(const std::string &productId, int quantity, std::string &orderId);      bool getUserOrders(std::vector<Protocol::OrderData> &orders);      bool getOrderById(const std::string &orderId, Protocol::OrderData &order);      bool updateOrderStatus(const std::string &orderId, Protocol::OrderStatus status);      bool getAllOrders(std::vector<Protocol::OrderData> &orders); // 管理员功能      // 库存锁定操作      bool lockInventory(const std::string &productId, int quantity);      bool unlockInventory(const std::string &productId, int quantity);      bool lockCartInventory(const std::vector<Protocol::CartItemData> &items);      bool unlockCartInventory(const std::vector<Protocol::CartItemData> &items);      // 商家管理操作      bool manageProductPrice(const std::string &productName, double newPrice);      bool manageProductQuantity(const std::string &productName, int newQuantity);      bool manageProductDiscount(const std::string &productName, double newDiscount);      bool applyCategoryDiscount(const std::string &category, double discount);  };  #endif // NETWORK\_CLIENT\_H |

#### client.cpp

|  |
| --- |
| #include "client.h"  #include <iostream>  #include <chrono>  NetworkClient::NetworkClient(const std::string &address, int port)      : serverAddress(address), serverPort(port), clientSocket(INVALID\_SOCKET),        isConnected(false), shouldStop(false)  {      // 初始化Winsock      WSADATA wsaData;      int result = WSAStartup(MAKEWORD(2, 2), &wsaData);      if (result != 0)      {          std::cerr << "WSAStartup failed: " << result << std::endl;      }  }  NetworkClient::~NetworkClient()  {      disconnect();      WSACleanup();  }  bool NetworkClient::connect()  {      if (isConnected.load())      {          return true;      }      // 创建socket      clientSocket = socket(AF\_INET, SOCK\_STREAM, IPPROTO\_TCP);      if (clientSocket == INVALID\_SOCKET)      {          std::cerr << "Socket creation failed: " << WSAGetLastError() << std::endl;          return false;      }      // 设置服务器地址      sockaddr\_in serverAddr;      serverAddr.sin\_family = AF\_INET;      serverAddr.sin\_port = htons(serverPort);      inet\_pton(AF\_INET, serverAddress.c\_str(), &serverAddr.sin\_addr);      // 连接服务器      if (::connect(clientSocket, (sockaddr \*)&serverAddr, sizeof(serverAddr)) == SOCKET\_ERROR)      {          std::cerr << "Connect failed: " << WSAGetLastError() << std::endl;          closesocket(clientSocket);          clientSocket = INVALID\_SOCKET;          return false;      }      isConnected.store(true);      shouldStop.store(false);      // 启动接收线程      receiveThread = std::thread(&NetworkClient::receiveLoop, this);      std::cout << "已连接到服务器 " << serverAddress << ":" << serverPort << std::endl;      return true;  }  void NetworkClient::disconnect()  {      if (!isConnected.load())      {          return;      }      shouldStop.store(true);      isConnected.store(false);      if (clientSocket != INVALID\_SOCKET)      {          closesocket(clientSocket);          clientSocket = INVALID\_SOCKET;      }      if (receiveThread.joinable())      {          receiveThread.join();      }      // 清理消息队列      std::lock\_guard<std::mutex> lock(queueMutex);      while (!responseQueue.empty())      {          responseQueue.pop();      }      std::cout << "已断开连接" << std::endl;  }  void NetworkClient::receiveLoop()  {      while (!shouldStop.load() && isConnected.load())      {          std::string rawData = receiveRawData();          if (rawData.empty())          {              break;          }          Protocol::Message message = Protocol::Message::deserialize(rawData);          // 如果设置了回调函数，调用回调          if (messageCallback)          {              messageCallback(message);          }          // 将消息放入队列          {              std::lock\_guard<std::mutex> lock(queueMutex);              responseQueue.push(message);          }          queueCondition.notify\_one();      }  }  bool NetworkClient::sendRawData(const std::string &data)  {      if (!isConnected.load())      {          return false;      }      // 发送数据长度      uint32\_t dataLength = htonl(static\_cast<uint32\_t>(data.length()));      if (send(clientSocket, reinterpret\_cast<const char \*>(&dataLength), sizeof(dataLength), 0) == SOCKET\_ERROR)      {          std::cerr << "Send length failed: " << WSAGetLastError() << std::endl;          return false;      }      // 发送数据内容      int totalSent = 0;      int dataSize = static\_cast<int>(data.length());      while (totalSent < dataSize)      {          int sent = send(clientSocket, data.c\_str() + totalSent, dataSize - totalSent, 0);          if (sent == SOCKET\_ERROR)          {              std::cerr << "Send data failed: " << WSAGetLastError() << std::endl;              return false;          }          totalSent += sent;      }      return true;  }  std::string NetworkClient::receiveRawData()  {      if (!isConnected.load())      {          return "";      }      // 接收数据长度      uint32\_t dataLength;      int received = recv(clientSocket, reinterpret\_cast<char \*>(&dataLength), sizeof(dataLength), 0);      if (received <= 0)      {          if (received == 0)          {              std::cout << "服务器关闭了连接" << std::endl;          }          else          {              std::cerr << "Receive length failed: " << WSAGetLastError() << std::endl;          }          isConnected.store(false);          return "";      }      dataLength = ntohl(dataLength);      // 接收数据内容      std::string data;      data.resize(dataLength);      int totalReceived = 0;      while (totalReceived < static\_cast<int>(dataLength))      {          int received = recv(clientSocket, &data[totalReceived], dataLength - totalReceived, 0);          if (received <= 0)          {              std::cerr << "Receive data failed: " << WSAGetLastError() << std::endl;              isConnected.store(false);              return "";          }          totalReceived += received;      }      return data;  }  bool NetworkClient::sendMessage(const Protocol::Message &message)  {      std::string serialized = message.serialize();      return sendRawData(serialized);  }  Protocol::Message NetworkClient::waitForResponse(Protocol::MessageType expectedType, int timeoutMs)  {      auto start = std::chrono::steady\_clock::now();      while (true)      {          {              std::unique\_lock<std::mutex> lock(queueMutex);              // 减少等待时间，避免阻塞UI              if (queueCondition.wait\_for(lock, std::chrono::milliseconds(50), [this]                                          { return !responseQueue.empty(); }))              {                  Protocol::Message msg = responseQueue.front();                  if (msg.type == expectedType || expectedType == Protocol::MessageType::RESPONSE\_DATA)                  {                      responseQueue.pop();                      return msg;                  }                  // 如果不是期望的消息类型，但是是错误消息，也应该返回                  if (msg.type == Protocol::MessageType::RESPONSE\_ERROR)                  {                      responseQueue.pop();                      return msg;                  }              }          }          auto now = std::chrono::steady\_clock::now();          if (std::chrono::duration\_cast<std::chrono::milliseconds>(now - start).count() > timeoutMs)          {              break;          }          // 如果连接断开，立即返回          if (!isConnected.load())          {              break;          }      }      return Protocol::Message(); // 超时返回空消息  }  bool NetworkClient::hasResponse() const  {      std::lock\_guard<std::mutex> lock(queueMutex);      return !responseQueue.empty();  }  Protocol::Message NetworkClient::getNextResponse()  {      std::lock\_guard<std::mutex> lock(queueMutex);      if (!responseQueue.empty())      {          Protocol::Message msg = responseQueue.front();          responseQueue.pop();          return msg;      }      return Protocol::Message();  }  void NetworkClient::setMessageCallback(std::function<void(const Protocol::Message &)> callback)  {      messageCallback = callback;  }  // 用户操作实现  bool NetworkClient::login(const std::string &username, const std::string &password, Protocol::UserData &userData)  {      std::string errorMessage;      return loginWithError(username, password, userData, errorMessage);  }  bool NetworkClient::loginWithError(const std::string &username, const std::string &password, Protocol::UserData &userData, std::string &errorMessage)  {      Protocol::Message request(Protocol::MessageType::USER\_LOGIN);      request.setData("username", username);      request.setData("password", password);      if (!sendMessage(request))      {          errorMessage = "无法发送登录请求";          return false;      }      Protocol::Message response = waitForResponse(Protocol::MessageType::RESPONSE\_SUCCESS);      if (response.type == Protocol::MessageType::RESPONSE\_SUCCESS)      {          sessionId = response.getData("sessionId");          userData = Protocol::UserData::deserialize(response.getData("userData"));          return true;      }      else if (response.type == Protocol::MessageType::RESPONSE\_ERROR)      {          errorMessage = response.getData("error");          return false;      }      errorMessage = "登录超时或网络错误";      return false;  }  bool NetworkClient::registerUser(const std::string &username, const std::string &password, Protocol::UserType userType)  {      Protocol::Message request(Protocol::MessageType::USER\_REGISTER);      request.setData("username", username);      request.setData("password", password);      request.setData("userType", std::to\_string(static\_cast<int>(userType)));      if (!sendMessage(request))      {          return false;      }      Protocol::Message response = waitForResponse(Protocol::MessageType::RESPONSE\_SUCCESS);      return response.type == Protocol::MessageType::RESPONSE\_SUCCESS;  }  bool NetworkClient::logout()  {      Protocol::Message request(Protocol::MessageType::USER\_LOGOUT, sessionId);      if (!sendMessage(request))      {          return false;      }      Protocol::Message response = waitForResponse(Protocol::MessageType::RESPONSE\_SUCCESS);      if (response.type == Protocol::MessageType::RESPONSE\_SUCCESS)      {          sessionId.clear();          return true;      }      return false;  }  // 商品操作实现  bool NetworkClient::getAllProducts(std::vector<Protocol::ProductData> &products)  {      Protocol::Message request(Protocol::MessageType::PRODUCT\_GET\_ALL, sessionId);      if (!sendMessage(request))      {          return false;      }      Protocol::Message response = waitForResponse(Protocol::MessageType::RESPONSE\_DATA);      if (response.type == Protocol::MessageType::RESPONSE\_DATA)      {          products.clear();          int count = std::stoi(response.getData("count"));          for (int i = 0; i < count; ++i)          {              std::string productData = response.getData("product\_" + std::to\_string(i));              products.push\_back(Protocol::ProductData::deserialize(productData));          }          return true;      }      return false;  }  // 购物车操作实现  bool NetworkClient::getCart(std::vector<Protocol::CartItemData> &cartItems)  {      Protocol::Message request(Protocol::MessageType::CART\_GET, sessionId);      if (!sendMessage(request))      {          return false;      }      Protocol::Message response = waitForResponse(Protocol::MessageType::RESPONSE\_DATA);      if (response.type == Protocol::MessageType::RESPONSE\_DATA)      {          cartItems.clear();          int count = std::stoi(response.getData("count"));          for (int i = 0; i < count; ++i)          {              std::string itemData = response.getData("item\_" + std::to\_string(i));              cartItems.push\_back(Protocol::CartItemData::deserialize(itemData));          }          return true;      }      return false;  }  bool NetworkClient::addToCart(const std::string &productId, int quantity)  {      Protocol::Message request(Protocol::MessageType::CART\_ADD\_ITEM, sessionId);      request.setData("productId", productId);      request.setData("quantity", std::to\_string(quantity));      if (!sendMessage(request))      {          return false;      }      Protocol::Message response = waitForResponse(Protocol::MessageType::RESPONSE\_SUCCESS);      return response.type == Protocol::MessageType::RESPONSE\_SUCCESS;  }  bool NetworkClient::removeFromCart(const std::string &productId)  {      Protocol::Message request(Protocol::MessageType::CART\_REMOVE\_ITEM, sessionId);      request.setData("productId", productId);      if (!sendMessage(request))      {          return false;      }      Protocol::Message response = waitForResponse(Protocol::MessageType::RESPONSE\_SUCCESS);      return response.type == Protocol::MessageType::RESPONSE\_SUCCESS;  }  bool NetworkClient::clearCart()  {      Protocol::Message request(Protocol::MessageType::CART\_CLEAR, sessionId);      if (!sendMessage(request))      {          return false;      }      Protocol::Message response = waitForResponse(Protocol::MessageType::RESPONSE\_SUCCESS);      return response.type == Protocol::MessageType::RESPONSE\_SUCCESS;  }  bool NetworkClient::addProduct(const Protocol::ProductData &product)  {      Protocol::Message request(Protocol::MessageType::PRODUCT\_ADD, sessionId);      request.setData("productData", product.serialize());      if (!sendMessage(request))      {          return false;      }      Protocol::Message response = waitForResponse(Protocol::MessageType::RESPONSE\_SUCCESS);      return response.type == Protocol::MessageType::RESPONSE\_SUCCESS;  }  // 订单操作实现  bool NetworkClient::createOrder(const std::vector<Protocol::CartItemData> &items, std::string &orderId)  {      Protocol::Message request(Protocol::MessageType::ORDER\_CREATE, sessionId);      request.setData("itemCount", std::to\_string(items.size()));      for (size\_t i = 0; i < items.size(); ++i)      {          request.setData("item\_" + std::to\_string(i), items[i].serialize());      }      if (!sendMessage(request))      {          return false;      }      Protocol::Message response = waitForResponse(Protocol::MessageType::RESPONSE\_SUCCESS);      if (response.type == Protocol::MessageType::RESPONSE\_SUCCESS)      {          orderId = response.getData("orderId");          return true;      }      return false;  }  bool NetworkClient::createDirectOrder(const std::string &productId, int quantity, std::string &orderId)  {      Protocol::Message request(Protocol::MessageType::ORDER\_DIRECT\_PURCHASE, sessionId);      request.setData("productId", productId);      request.setData("quantity", std::to\_string(quantity));      if (!sendMessage(request))      {          return false;      }      Protocol::Message response = waitForResponse(Protocol::MessageType::RESPONSE\_SUCCESS);      if (response.type == Protocol::MessageType::RESPONSE\_SUCCESS)      {          orderId = response.getData("orderId");          return true;      }      return false;  }  bool NetworkClient::getUserOrders(std::vector<Protocol::OrderData> &orders)  {      Protocol::Message request(Protocol::MessageType::ORDER\_GET\_BY\_USER, sessionId);      if (!sendMessage(request))      {          return false;      }      Protocol::Message response = waitForResponse(Protocol::MessageType::RESPONSE\_DATA);      if (response.type == Protocol::MessageType::RESPONSE\_DATA)      {          orders.clear();          int count = std::stoi(response.getData("count"));          for (int i = 0; i < count; ++i)          {              std::string orderData = response.getData("order\_" + std::to\_string(i));              // std::cerr << "Received order data: " << orderData << std::endl;              orders.push\_back(Protocol::OrderData::deserialize(orderData));          }          return true;      }      return false;  }  // 用户信息更新实现  bool NetworkClient::getUserInfo(Protocol::UserData &userData)  {      Protocol::Message request(Protocol::MessageType::USER\_GET\_INFO, sessionId);      if (!sendMessage(request))      {          return false;      }      Protocol::Message response = waitForResponse(Protocol::MessageType::RESPONSE\_DATA);      if (response.type == Protocol::MessageType::RESPONSE\_DATA)      {          userData = Protocol::UserData::deserialize(response.getData("userData"));          return true;      }      return false;  }  bool NetworkClient::changePassword(const std::string &oldPassword, const std::string &newPassword)  {      Protocol::Message request(Protocol::MessageType::USER\_CHANGE\_PASSWORD, sessionId);      request.setData("oldPassword", oldPassword);      request.setData("newPassword", newPassword);      if (!sendMessage(request))      {          return false;      }      Protocol::Message response = waitForResponse(Protocol::MessageType::RESPONSE\_SUCCESS);      return response.type == Protocol::MessageType::RESPONSE\_SUCCESS;  }  bool NetworkClient::updateBalance(double amount)  {      Protocol::Message request(Protocol::MessageType::USER\_UPDATE\_BALANCE, sessionId);      request.setData("amount", std::to\_string(amount));      if (!sendMessage(request))      {          return false;      }      Protocol::Message response = waitForResponse(Protocol::MessageType::RESPONSE\_SUCCESS);      return response.type == Protocol::MessageType::RESPONSE\_SUCCESS;  }  // 商品搜索实现  bool NetworkClient::searchProducts(const std::string &keyword, std::vector<Protocol::ProductData> &products)  {      Protocol::Message request(Protocol::MessageType::PRODUCT\_SEARCH, sessionId);      request.setData("keyword", keyword);      if (!sendMessage(request))      {          return false;      }      Protocol::Message response = waitForResponse(Protocol::MessageType::RESPONSE\_DATA);      if (response.type == Protocol::MessageType::RESPONSE\_DATA)      {          products.clear();          int count = std::stoi(response.getData("count"));          for (int i = 0; i < count; ++i)          {              std::string productData = response.getData("product\_" + std::to\_string(i));              // std::cerr << "Received product data: " << productData << std::endl;              products.push\_back(Protocol::ProductData::deserialize(productData));          }          return true;      }      return false;  }  // 购物车商品更新实现  bool NetworkClient::updateCartItem(const std::string &productId, int newQuantity)  {      Protocol::Message request(Protocol::MessageType::CART\_UPDATE\_ITEM, sessionId);      request.setData("productId", productId);      request.setData("quantity", std::to\_string(newQuantity));      if (!sendMessage(request))      {          return false;      }      Protocol::Message response = waitForResponse(Protocol::MessageType::RESPONSE\_SUCCESS);      return response.type == Protocol::MessageType::RESPONSE\_SUCCESS;  }  // 商家管理操作实现  bool NetworkClient::manageProductPrice(const std::string &productName, double newPrice)  {      Protocol::Message request(Protocol::MessageType::PRODUCT\_MANAGE\_PRICE, sessionId);      request.setData("productName", productName);      request.setData("newPrice", std::to\_string(newPrice));      if (!sendMessage(request))      {          return false;      }      Protocol::Message response = waitForResponse(Protocol::MessageType::RESPONSE\_SUCCESS);      return response.type == Protocol::MessageType::RESPONSE\_SUCCESS;  }  bool NetworkClient::manageProductQuantity(const std::string &productName, int newQuantity)  {      Protocol::Message request(Protocol::MessageType::PRODUCT\_MANAGE\_QUANTITY, sessionId);      request.setData("productName", productName);      request.setData("newQuantity", std::to\_string(newQuantity));      if (!sendMessage(request))      {          return false;      }      Protocol::Message response = waitForResponse(Protocol::MessageType::RESPONSE\_SUCCESS);      return response.type == Protocol::MessageType::RESPONSE\_SUCCESS;  }  bool NetworkClient::manageProductDiscount(const std::string &productName, double newDiscount)  {      Protocol::Message request(Protocol::MessageType::PRODUCT\_MANAGE\_DISCOUNT, sessionId);      request.setData("productName", productName);      request.setData("newDiscount", std::to\_string(newDiscount));      if (!sendMessage(request))      {          return false;      }      Protocol::Message response = waitForResponse(Protocol::MessageType::RESPONSE\_SUCCESS);      return response.type == Protocol::MessageType::RESPONSE\_SUCCESS;  }  bool NetworkClient::applyCategoryDiscount(const std::string &category, double discount)  {      Protocol::Message request(Protocol::MessageType::PRODUCT\_APPLY\_CATEGORY\_DISCOUNT, sessionId);      request.setData("category", category);      request.setData("discount", std::to\_string(discount));      if (!sendMessage(request))      {          return false;      }      Protocol::Message response = waitForResponse(Protocol::MessageType::RESPONSE\_SUCCESS);      return response.type == Protocol::MessageType::RESPONSE\_SUCCESS;  }  // 库存锁定操作实现  bool NetworkClient::lockInventory(const std::string &productId, int quantity)  {      Protocol::Message request(Protocol::MessageType::INVENTORY\_LOCK, sessionId);      request.setData("productId", productId);      request.setData("quantity", std::to\_string(quantity));      if (!sendMessage(request))      {          return false;      }      Protocol::Message response = waitForResponse(Protocol::MessageType::RESPONSE\_SUCCESS);      return response.type == Protocol::MessageType::RESPONSE\_SUCCESS;  }  bool NetworkClient::unlockInventory(const std::string &productId, int quantity)  {      Protocol::Message request(Protocol::MessageType::INVENTORY\_UNLOCK, sessionId);      request.setData("productId", productId);      request.setData("quantity", std::to\_string(quantity));      if (!sendMessage(request))      {          return false;      }      Protocol::Message response = waitForResponse(Protocol::MessageType::RESPONSE\_SUCCESS);      return response.type == Protocol::MessageType::RESPONSE\_SUCCESS;  }  bool NetworkClient::lockCartInventory(const std::vector<Protocol::CartItemData> &items)  {      Protocol::Message request(Protocol::MessageType::INVENTORY\_LOCK, sessionId);      request.setData("itemCount", std::to\_string(items.size()));      for (size\_t i = 0; i < items.size(); ++i)      {          request.setData("productId\_" + std::to\_string(i), items[i].productId);          request.setData("quantity\_" + std::to\_string(i), std::to\_string(items[i].quantity));      }      if (!sendMessage(request))      {          return false;      }      Protocol::Message response = waitForResponse(Protocol::MessageType::RESPONSE\_SUCCESS);      return response.type == Protocol::MessageType::RESPONSE\_SUCCESS;  }  bool NetworkClient::unlockCartInventory(const std::vector<Protocol::CartItemData> &items)  {      Protocol::Message request(Protocol::MessageType::INVENTORY\_UNLOCK, sessionId);      request.setData("itemCount", std::to\_string(items.size()));      for (size\_t i = 0; i < items.size(); ++i)      {          request.setData("productId\_" + std::to\_string(i), items[i].productId);          request.setData("quantity\_" + std::to\_string(i), std::to\_string(items[i].quantity));      }      if (!sendMessage(request))      {          return false;      }      Protocol::Message response = waitForResponse(Protocol::MessageType::RESPONSE\_SUCCESS);      return response.type == Protocol::MessageType::RESPONSE\_SUCCESS;  } |

#### client\_ui.h

|  |
| --- |
| #ifndef CLIENT\_UI\_H  #define CLIENT\_UI\_H  #include "../imgui/imgui.h" // 必须首先包含 ImGui  #include "../network/client.h"  #include "../network/protocol.h"  #include <string>  #include <vector>  #include <functional>  #include <memory>  #include <cstdarg> // 添加支持可变参数函数  #include "../imgui/backends/imgui\_impl\_glfw.h"  #include "../imgui/backends/imgui\_impl\_opengl3.h"  #include <GLFW/glfw3.h>  #include <iostream>  #include <algorithm>  #include <ctime>  #include <sstream>  #include <iomanip>  #include <cmath> // 为了 std::min, std::max  // 前向声明 - 确保正确的类型声明  typedef struct GLFWwindow GLFWwindow;  // 客户端UI系统主类  class ClientUI  {  private:    // GLFW窗口    GLFWwindow \*window = nullptr;    // 窗口尺寸    int width = 800;    int height = 600;    int originalWidth = 800;  // 原始窗口宽度    int originalHeight = 600; // 原始窗口高度    float uiScale = 1.0f;     // UI缩放比例    // UI状态    bool shouldClose = false;    int currentPage = 0; // 0=主菜单, 1=用户菜单, 2=商城    Protocol::UserType userType = Protocol::UserType::CUSTOMER;    // 商品滚动位置    float scrollY = 0.0f;    // 登录输入缓冲区    char usernameBuffer[128] = {0};    char passwordBuffer[128] = {0};    // 用户信息    Protocol::UserData currentUser;    bool isLoggedIn = false; // 商品相关    char searchBuffer[128] = {0};    char lastSearchBuffer[128] = {0}; // 用于跟踪搜索框内容变化    std::vector<Protocol::ProductData> allProducts;    std::vector<Protocol::ProductData> searchResults;    // 购买商品    char buyProductName[128] = {0};    int buyQuantity = 1; // 商品管理 (商家)    char newProductName[128] = {0};    char newProductDesc[256] = {0};    float newProductPrice = 0.0f;    int newProductQuantity = 1;    float newProductDiscount = 0.0f;    int newProductType = 0;          // 商品类型索引 (0=图书, 1=服装, 2=食品, 3=其他)    char newProductAttr1[128] = {0}; // 第一个属性字段    char newProductAttr2[128] = {0}; // 第二个属性字段    // 购物车    std::vector<Protocol::CartItemData> cartItems;    bool showCartWindow = false; // 订单    std::vector<Protocol::OrderData> userOrders;    bool showOrderWindow = false;    Protocol::OrderData selectedOrder;    // 网络客户端    std::unique\_ptr<NetworkClient> networkClient;    // 状态信息    std::string statusMessage;    bool showErrorMessage = false;    std::string errorMessage;    // 确认对话框状态    bool showPurchaseConfirmDialog = false;    bool showDirectPurchaseConfirmDialog = false;    bool showCartCheckoutConfirmDialog = false;    Protocol::ProductData productToPurchase;    int quantityToPurchase = 1;    bool inventoryLocked = false; // 标记直接购买时库存是否已锁定// 成功消息弹窗状态    bool showAddToCartSuccessPopup = false;    std::string addToCartSuccessMessage;    bool showDirectPurchaseSuccessPopup = false;    std::string directPurchaseSuccessMessage; // 密码修改弹窗状态    bool showPasswordChangeDialog = false;    bool showPasswordChangeSuccessDialog = false;    bool showPasswordChangeFailDialog = false;    // 充值弹窗状态    bool showRechargeDialog = false;    bool showRechargeSuccessDialog = false;    bool showRechargeFailDialog = false;    float rechargeAmount = 0.0f;    std::string rechargeMessage;    // 主题管理    bool isDarkTheme = true;    ImVec4 primaryColor;    ImVec4 secondaryColor;    ImVec4 backgroundColor;    // 渲染方法    void renderMainMenu();    void renderUserMenu();    void renderStore();    void renderUserRegister();    void renderUserLogin();    void renderStoreCustomer();    void renderStoreSeller();    void renderStoreAdmin();    void renderProductList(const std::vector<Protocol::ProductData> &products, const std::string &title);    void renderCartWindow();    void renderOrderWindow();    void renderOrderDetails(const Protocol::OrderData &order);    void renderAddProductWindow();    void renderStatusMessages(); // 对话框和弹窗方法    void renderPurchaseConfirmDialog();    void renderDirectPurchaseConfirmDialog();    void renderCartCheckoutConfirmDialog();    void renderAddToCartSuccessPopup();    void renderDirectPurchaseSuccessPopup();    void renderPasswordChangeDialog();    void renderPasswordChangeSuccessDialog();    void renderPasswordChangeFailDialog();    void renderRechargeDialog();    void renderRechargeSuccessDialog();    void renderRechargeFailDialog();    // UI优化相关方法    void setupModernTheme();    void setupDarkTheme();    void setupLightTheme();    bool renderModernButton(const char \*label, ImVec2 size, bool primary = false);    void renderCard(const char \*title, const char \*content, bool collapsible = false);    void renderSearchBox();    void renderProductCard(const Protocol::ProductData &product);    void renderStatusBar();    // 窗口响应式方法    void updateWindowSize();    void updateUIScale();    ImVec2 scaleSize(const ImVec2 &size);    float scaleFloat(float value);    static void framebufferSizeCallback(GLFWwindow \*window, int width, int height);  public:    ClientUI(int width = 800, int height = 600);    ~ClientUI();    // 初始化UI系统    bool init();    // 连接到服务器    bool connectToServer(const std::string &address = "127.0.0.1", int port = 8888); // 断开服务器连接    void disconnectFromServer();    // 主循环    void mainLoop();    // 是否应该关闭    bool isClosing() const { return shouldClose; }    // 用户操作    bool login(const std::string &username, const std::string &password);    bool registerUser(const std::string &username, const std::string &password, Protocol::UserType userType);    void logout();    // 商品操作    bool addProduct(const Protocol::ProductData &product);    void refreshProducts();    void refreshCart();    void refreshOrders();    void refreshUserOrders();    void performSearch();    void addProductToCart(const Protocol::ProductData &product);    void updateCartQuantity(const std::string &productId, int newQuantity);    void removeFromCart(const std::string &productId);    void checkout();    void checkoutWithInventoryLock();    void addToCartByName(const std::string &productName, int quantity);    void purchaseProduct(const std::string &productName, int quantity);    void clearMessages();    // 状态和错误消息方法    void setStatus(const std::string &message);    void setError(const std::string &error);    void setStatus(const char \*format, ...);    void setError(const char \*format, ...); // 获取当前用户信息    const Protocol::UserData &getCurrentUser() const { return currentUser; }    bool getIsLoggedIn() const { return isLoggedIn; }  };  #endif // CLIENT\_UI\_H |

#### client\_ui.cpp

|  |
| --- |
| #include "client\_ui.h"  // ClientUI构造函数  ClientUI::ClientUI(int width, int height)  : width(width), height(height),  originalWidth(width), originalHeight(height),  isDarkTheme(true),  primaryColor(ImVec4(0.26f, 0.59f, 0.98f, 1.00f)),  secondaryColor(ImVec4(0.60f, 0.60f, 0.60f, 1.00f)),  backgroundColor(ImVec4(0.06f, 0.06f, 0.06f, 1.00f))  {  networkClient = std::make\_unique<NetworkClient>();  }  // ClientUI析构函数  ClientUI::~ClientUI()  {  // 清理ImGui  ImGui\_ImplOpenGL3\_Shutdown();  ImGui\_ImplGlfw\_Shutdown();  ImGui::DestroyContext(); // 清理GLFW  if (this->window)  {  glfwDestroyWindow(this->window);  this->window = nullptr;  }  glfwTerminate();  }  // 初始化UI  bool ClientUI::init()  {  // 初始化GLFW  if (!glfwInit())  {  std::cerr << "GLFW初始化失败" << std::endl;  return false;  } // 创建窗口  this->window = glfwCreateWindow(width, height, "电子商城系统 - 客户端", nullptr, nullptr);  if (!this->window)  {  std::cerr << "创建窗口失败" << std::endl;  glfwTerminate();  return false;  }  // 设置当前上下文  glfwMakeContextCurrent(this->window);  glfwSwapInterval(1); // 启用垂直同步  // 设置窗口大小回调函数  glfwSetWindowUserPointer(this->window, this);  glfwSetFramebufferSizeCallback(this->window, framebufferSizeCallback); // 初始化ImGui  IMGUI\_CHECKVERSION();  ImGui::CreateContext();  ImGuiIO &io = ImGui::GetIO();  io.ConfigFlags |= ImGuiConfigFlags\_NavEnableKeyboard; // 启用键盘控制  // 加载中文字体  ImFont \*font = nullptr;  // 尝试加载多个可能的中文字体  const char \*fontPaths[] = {  "c:\\Windows\\Fonts\\simhei.ttf", // 黑体  "c:\\Windows\\Fonts\\simsun.ttc", // 宋体  "c:\\Windows\\Fonts\\msyh.ttc", // 微软雅黑  "c:\\Windows\\Fonts\\simkai.ttf", // 楷体  "c:\\Windows\\Fonts\\simfang.ttf" // 仿宋  };  for (const char \*fontPath : fontPaths)  {  font = io.Fonts->AddFontFromFileTTF(fontPath, 18.0f, NULL, io.Fonts->GetGlyphRangesChineseFull());  if (font)  {  std::cout << "成功加载字体: " << fontPath << std::endl;  break;  }  }  if (!font)  {  std::cout << "警告: 无法加载中文字体，将使用默认字体" << std::endl;  // 使用默认字体，但仍然设置中文字符范围  io.Fonts->AddFontDefault();  io.Fonts->GetGlyphRangesChineseFull();  }  // 设置现代化ImGui风格  setupModernTheme();  // 平台/渲染器后端  ImGui\_ImplGlfw\_InitForOpenGL(window, true);  ImGui\_ImplOpenGL3\_Init("#version 130");  return true;  }  // 连接到服务器  bool ClientUI::connectToServer(const std::string &address, int port)  {  if (!networkClient)  {  return false;  }  return networkClient->connect();  }  // 断开服务器连接  void ClientUI::disconnectFromServer()  {  if (networkClient)  {  if (isLoggedIn)  {  networkClient->logout();  }  networkClient->disconnect();  }  }  // 主循环  void ClientUI::mainLoop()  {  // 主循环  while (!glfwWindowShouldClose(window) && !shouldClose)  {  glfwPollEvents();  // 开始新帧  ImGui\_ImplOpenGL3\_NewFrame();  ImGui\_ImplGlfw\_NewFrame();  ImGui::NewFrame();  // 全屏窗口  ImGui::SetNextWindowPos(ImVec2(0, 0));  ImGui::SetNextWindowSize(ImVec2(width, height));  ImGui::Begin("商城系统", nullptr,  ImGuiWindowFlags\_NoTitleBar |  ImGuiWindowFlags\_NoResize |  ImGuiWindowFlags\_NoMove |  ImGuiWindowFlags\_NoScrollbar);  // 根据当前页面渲染不同的UI  switch (currentPage)  {  case 0: // 主菜单  renderMainMenu();  break;  case 1: // 用户菜单  renderUserMenu();  break;  case 2: // 商城  renderStore();  break;  case 3: // 注册页面  renderUserRegister();  break;  case 4: // 登录页面  renderUserLogin();  break;  }  // 显示购物车窗口  if (showCartWindow && isLoggedIn && userType == Protocol::UserType::CUSTOMER)  {  renderCartWindow();  }  // 显示订单管理窗口  if (showOrderWindow && isLoggedIn && userType == Protocol::UserType::CUSTOMER)  {  renderOrderWindow();  }  // // 显示添加商品窗口  // if (isLoggedIn && userType == Protocol::UserType::SELLER)  // {  // renderAddProductWindow();  // } // 渲染状态消息  // renderStatusMessages(); // 渲染确认对话框和弹窗  if (showPurchaseConfirmDialog)  {  renderPurchaseConfirmDialog();  }  if (showDirectPurchaseConfirmDialog)  {  renderDirectPurchaseConfirmDialog();  }  if (showCartCheckoutConfirmDialog)  {  renderCartCheckoutConfirmDialog();  }  if (showAddToCartSuccessPopup)  {  renderAddToCartSuccessPopup();  }  if (showDirectPurchaseSuccessPopup)  {  renderDirectPurchaseSuccessPopup();  }  if (showPasswordChangeDialog)  {  renderPasswordChangeDialog();  }  if (showPasswordChangeSuccessDialog)  {  renderPasswordChangeSuccessDialog();  }  if (showPasswordChangeFailDialog)  {  renderPasswordChangeFailDialog();  }  if (showRechargeDialog)  {  renderRechargeDialog();  }  if (showRechargeSuccessDialog)  {  renderRechargeSuccessDialog();  }  if (showRechargeFailDialog)  {  renderRechargeFailDialog();  }  ImGui::End();  // 渲染  ImGui::Render();  int displayW, displayH;  glfwGetFramebufferSize(window, &displayW, &displayH);  glViewport(0, 0, displayW, displayH);  glClearColor(0.45f, 0.55f, 0.60f, 1.00f);  glClear(GL\_COLOR\_BUFFER\_BIT);  ImGui\_ImplOpenGL3\_RenderDrawData(ImGui::GetDrawData());  glfwSwapBuffers(window);  }  }  // 渲染主菜单  void ClientUI::renderMainMenu()  {  // 创建现代化的欢迎界面  ImVec2 windowSize = ImGui::GetWindowSize();  float centerX = windowSize.x \* 0.5f;  // 标题区域  ImGui::PushStyleVar(ImGuiStyleVar\_ItemSpacing, ImVec2(0, 20));  // 主标题  ImGui::PushFont(ImGui::GetIO().Fonts->Fonts[0]);  ImGui::SetWindowFontScale(2.0f);  ImGui::SetCursorPosX(centerX - ImGui::CalcTextSize("电子商城系统").x \* 0.5f);  ImGui::TextColored(this->primaryColor, "电子商城系统");  ImGui::SetWindowFontScale(1.0f);  ImGui::PopFont();  // 副标题  ImGui::SetWindowFontScale(1.2f);  const char \*subtitle = "现代化购物体验平台";  ImGui::SetCursorPosX(centerX - ImGui::CalcTextSize(subtitle).x \* 0.5f);  ImGui::TextColored(ImVec4(0.7f, 0.7f, 0.7f, 1.0f), subtitle);  ImGui::SetWindowFontScale(1.0f);  ImGui::Spacing();  ImGui::Separator();  ImGui::Spacing();  ImGui::Spacing();  // 主要功能按钮区域  ImVec2 buttonSize(260, 60);  float buttonSpacing = 20;  ImGui::SetCursorPosX(centerX - buttonSize.x \* 0.5f);  renderModernButton("注册新用户", buttonSize, true);  if (ImGui::IsItemClicked())  {  currentPage = 3; // 切换到注册页面  }  ImGui::Spacing();  ImGui::SetCursorPosX(centerX - buttonSize.x \* 0.5f);  renderModernButton("用户登录", buttonSize, false);  if (ImGui::IsItemClicked())  {  currentPage = 4; // 切换到登录页面  memset(usernameBuffer, 0, sizeof(usernameBuffer));  memset(passwordBuffer, 0, sizeof(passwordBuffer));  }  ImGui::Spacing();  ImGui::SetCursorPosX(centerX - buttonSize.x \* 0.5f);  // 游客模式按钮 - 不同的样式  ImGui::PushStyleColor(ImGuiCol\_Button, ImVec4(0.3f, 0.3f, 0.3f, 0.8f));  ImGui::PushStyleColor(ImGuiCol\_ButtonHovered, ImVec4(0.4f, 0.4f, 0.4f, 1.0f));  ImGui::PushStyleColor(ImGuiCol\_ButtonActive, ImVec4(0.2f, 0.2f, 0.2f, 1.0f));  if (ImGui::Button("进入商城 (游客)", buttonSize))  {  currentPage = 2; // 切换到商城  refreshProducts();  }  ImGui::PopStyleColor(3);  ImGui::Spacing();  ImGui::Spacing();  // 退出按钮 - 放在底部  ImGui::SetCursorPosX(centerX - 100);  ImGui::PushStyleColor(ImGuiCol\_Button, ImVec4(0.6f, 0.2f, 0.2f, 0.8f));  ImGui::PushStyleColor(ImGuiCol\_ButtonHovered, ImVec4(0.8f, 0.3f, 0.3f, 1.0f));  ImGui::PushStyleColor(ImGuiCol\_ButtonActive, ImVec4(0.5f, 0.1f, 0.1f, 1.0f));  if (ImGui::Button("退出程序", ImVec2(200, 40)))  {  shouldClose = true;  }  ImGui::PopStyleColor(3);  ImGui::PopStyleVar();  // 在底部显示版本信息和主题切换  ImGui::SameLine();  ImGui::SetCursorPosX(windowSize.x - 120);  if (ImGui::SmallButton(isDarkTheme ? "浅色主题" : "深色主题"))  {  isDarkTheme = !isDarkTheme;  setupModernTheme();  }  }  // 渲染用户菜单  void ClientUI::renderUserMenu()  {  if (!isLoggedIn)  {  currentPage = 0; // 回到主菜单  return;  }  std::string userTypeStr;  if (userType == Protocol::UserType::CUSTOMER)  userTypeStr = "消费者";  else if (userType == Protocol::UserType::SELLER)  userTypeStr = "商家";  else if (userType == Protocol::UserType::ADMIN)  userTypeStr = "管理员";  else  userTypeStr = "未知";  ImGui::SetWindowFontScale(1.5f);  ImGui::Text("%s菜单 - %s", userTypeStr.c\_str(), currentUser.username.c\_str());  ImGui::SetWindowFontScale(1.0f);  ImGui::Separator();  // 用户信息  ImGui::BeginChild("UserInfo", ImVec2(width \* 0.4f, height - 150), true);  ImGui::Text("用户名: %s", currentUser.username.c\_str());  ImGui::Text("用户类型: %s", userTypeStr.c\_str());  ImGui::Text("账户余额: CNY%.2f", currentUser.balance);  ImGui::EndChild();  // 操作区域  ImGui::SameLine();  ImGui::BeginChild("UserActions", ImVec2(0, height - 150), true); // 修改密码  if (ImGui::Button("修改密码", ImVec2(100, 30)))  {  showPasswordChangeDialog = true;  }  ImGui::Separator();  // 消费者特有 - 充值  if (userType == Protocol::UserType::CUSTOMER)  {  if (ImGui::Button("账户充值", ImVec2(100, 30)))  {  showRechargeDialog = true;  rechargeAmount = 0.0f;  }  }  ImGui::EndChild();  // 底部按钮  ImGui::Separator();  if (ImGui::Button("进入商城", ImVec2(120, 40)))  {  currentPage = 2; // 切换到商城  searchResults.clear();  memset(searchBuffer, 0, sizeof(searchBuffer));  refreshProducts();  }  ImGui::SameLine();  if (ImGui::Button("退出登录", ImVec2(120, 40)))  {  logout();  currentPage = 0; // 回到主菜单  }  }  // 渲染商城  void ClientUI::renderStore()  {  ImGui::SetWindowFontScale(1.5f);  if (isLoggedIn)  {  if (userType == Protocol::UserType::SELLER)  {  ImGui::Text("商城管理 - %s", currentUser.username.c\_str());  }  else  {  ImGui::Text("商城 - %s", currentUser.username.c\_str());  }  }  else  {  ImGui::Text("商城 - 游客模式");  }  ImGui::SetWindowFontScale(1.0f);  ImGui::Separator();  // 根据用户类型显示不同的商城界面  if (userType == Protocol::UserType::SELLER)  {  // 商家  renderStoreSeller();  }  else  {  // 消费者或游客  renderStoreCustomer();  }  // 底部按钮  ImGui::Separator();  if (ImGui::Button("返回", ImVec2(120, 40)))  {  if (isLoggedIn)  {  currentPage = 1; // 返回用户菜单  }  else  {  currentPage = 0; // 返回主菜单  }  }  }  // 渲染注册页面  void ClientUI::renderUserRegister()  {  ImGui::SetWindowFontScale(1.5f);  ImGui::Text("用户注册");  ImGui::SetWindowFontScale(1.0f);  ImGui::Separator();  static char regUsername[128] = {0};  static char regPassword[128] = {0};  static int regUserType = 0; // 0=消费者, 1=商家  ImGui::SetNextItemWidth(250);  ImGui::InputText("用户名", regUsername, IM\_ARRAYSIZE(regUsername));  ImGui::SetNextItemWidth(250);  ImGui::InputText("密码", regPassword, IM\_ARRAYSIZE(regPassword), ImGuiInputTextFlags\_Password);  ImGui::RadioButton("消费者", &regUserType, 0);  ImGui::SameLine();  ImGui::RadioButton("商家", &regUserType, 1);  if (ImGui::Button("注册", ImVec2(120, 40)))  {  if (strlen(regUsername) == 0)  {  ImGui::OpenPopup("注册失败");  }  else  {  Protocol::UserType type = (regUserType == 0) ? Protocol::UserType::CUSTOMER : Protocol::UserType::SELLER;  if (networkClient && networkClient->registerUser(regUsername, regPassword, type))  {  // 注册成功后自动登录  Protocol::UserData userData;  if (networkClient->login(regUsername, regPassword, userData))  {  currentUser = userData;  userType = userData.userType;  isLoggedIn = true;  ImGui::OpenPopup("注册成功");  currentPage = 1; // 直接跳转到用户菜单  clearMessages();  }  else  {  ImGui::OpenPopup("登录失败");  }  memset(regUsername, 0, sizeof(regUsername));  memset(regPassword, 0, sizeof(regPassword));  }  else  {  ImGui::OpenPopup("用户名已存在");  }  }  }  ImGui::SameLine();  if (ImGui::Button("返回", ImVec2(120, 40)))  {  currentPage = 0; // 回到主菜单  }  // 注册相关弹窗  if (ImGui::BeginPopupModal("注册失败", NULL, ImGuiWindowFlags\_AlwaysAutoResize))  {  ImGui::Text("注册失败，用户名不能为空！");  if (ImGui::Button("确定", ImVec2(120, 0)))  {  ImGui::CloseCurrentPopup();  }  ImGui::EndPopup();  }  if (ImGui::BeginPopupModal("用户名已存在", NULL, ImGuiWindowFlags\_AlwaysAutoResize))  {  ImGui::Text("该用户名已存在，请选择其他用户名！");  if (ImGui::Button("确定", ImVec2(120, 0)))  {  ImGui::CloseCurrentPopup();  }  ImGui::EndPopup();  }  if (ImGui::BeginPopupModal("注册成功", NULL, ImGuiWindowFlags\_AlwaysAutoResize))  {  ImGui::Text("注册成功！已自动为您登录。");  if (ImGui::Button("确定", ImVec2(120, 0)))  {  ImGui::CloseCurrentPopup();  }  ImGui::EndPopup();  }  if (ImGui::BeginPopupModal("登录失败", NULL, ImGuiWindowFlags\_AlwaysAutoResize))  {  ImGui::Text("注册成功但自动登录失败，请手动登录。");  if (ImGui::Button("确定", ImVec2(120, 0)))  {  ImGui::CloseCurrentPopup();  currentPage = 4; // 跳转到登录页面  }  ImGui::EndPopup();  }  }  // 渲染登录页面  void ClientUI::renderUserLogin()  {  ImGui::SetWindowFontScale(1.5f);  ImGui::Text("用户登录");  ImGui::SetWindowFontScale(1.0f);  ImGui::Separator();  ImGui::SetNextItemWidth(250);  ImGui::InputText("用户名", usernameBuffer, IM\_ARRAYSIZE(usernameBuffer));  ImGui::SetNextItemWidth(250);  ImGui::InputText("密码", passwordBuffer, IM\_ARRAYSIZE(passwordBuffer), ImGuiInputTextFlags\_Password);  if (ImGui::Button("登录", ImVec2(120, 40)))  {  if (login(usernameBuffer, passwordBuffer))  {  clearMessages();  currentPage = 1; // 跳转至用户菜单  }  else  {  // 根据错误类型显示不同的弹窗  if (errorMessage == "该用户已经登录")  {  ImGui::OpenPopup("用户重复登录");  }  else if (errorMessage == "用户名或密码错误")  {  ImGui::OpenPopup("密码错误");  }  else  {  ImGui::OpenPopup("登录失败");  }  }  }  ImGui::SameLine();  if (ImGui::Button("返回", ImVec2(120, 40)))  {  currentPage = 0; // 回到主菜单  }  // 用户重复登录弹窗  if (ImGui::BeginPopupModal("用户重复登录", NULL, ImGuiWindowFlags\_AlwaysAutoResize))  {  ImGui::SetWindowFontScale(1.2f);  ImGui::Text("登录失败");  ImGui::SetWindowFontScale(1.0f);  ImGui::Separator();  ImGui::Text("该用户已经在其他客户端登录！");  ImGui::Text("请先在其他客户端退出登录，然后重试。");  if (ImGui::Button("确定", ImVec2(120, 30)))  {  ImGui::CloseCurrentPopup();  }  ImGui::EndPopup();  }  // 密码错误弹窗  if (ImGui::BeginPopupModal("密码错误", NULL, ImGuiWindowFlags\_AlwaysAutoResize))  {  ImGui::SetWindowFontScale(1.2f);  ImGui::Text("登录失败");  ImGui::SetWindowFontScale(1.0f);  ImGui::Separator();  ImGui::Text("用户名或密码错误！");  ImGui::Text("请检查您的登录信息后重试。");  if (ImGui::Button("确定", ImVec2(120, 30)))  {  ImGui::CloseCurrentPopup();  }  ImGui::EndPopup();  }  // 通用登录失败弹窗  if (ImGui::BeginPopupModal("登录失败", NULL, ImGuiWindowFlags\_AlwaysAutoResize))  {  ImGui::SetWindowFontScale(1.2f);  ImGui::Text("登录失败");  ImGui::SetWindowFontScale(1.0f);  ImGui::Separator();  ImGui::Text("错误信息：%s", errorMessage.c\_str());  if (ImGui::Button("确定", ImVec2(120, 30)))  {  ImGui::CloseCurrentPopup();  }  ImGui::EndPopup();  }  }  // 渲染消费者商城  void ClientUI::renderStoreCustomer()  {  // 现代化状态栏  renderStatusBar();  // 现代化搜索框容器  ImGui::PushStyleVar(ImGuiStyleVar\_ChildRounding, 15.0f);  ImGui::PushStyleColor(ImGuiCol\_ChildBg, ImVec4(0.12f, 0.13f, 0.14f, 1.0f));  ImGui::BeginChild("SearchContainer", ImVec2(0, 120), true);  ImGui::Spacing();  ImGui::Text("商品搜索");  ImGui::Spacing();  // 现代化搜索框  renderSearchBox();  // 快捷操作按钮  if (isLoggedIn && userType == Protocol::UserType::CUSTOMER)  {  ImGui::Spacing();  // 居中放置按钮  float buttonsWidth = 160.0f + 20.0f + 160.0f;  ImGui::SetCursorPosX((ImGui::GetContentRegionAvail().x - buttonsWidth) \* 0.5f);  ImGui::PushStyleVar(ImGuiStyleVar\_FrameRounding, 10.0f);  ImGui::PushStyleColor(ImGuiCol\_Button, ImVec4(0.8f, 0.4f, 0.2f, 0.8f));  ImGui::PushStyleColor(ImGuiCol\_ButtonHovered, ImVec4(0.9f, 0.5f, 0.3f, 1.0f));  if (ImGui::Button("查看购物车", ImVec2(160, 35)))  {  showCartWindow = true;  refreshCart();  }  ImGui::PopStyleColor(2);  ImGui::SameLine(0, 20.0f);  ImGui::PushStyleColor(ImGuiCol\_Button, ImVec4(0.2f, 0.6f, 0.8f, 0.8f));  ImGui::PushStyleColor(ImGuiCol\_ButtonHovered, ImVec4(0.3f, 0.7f, 0.9f, 1.0f));  if (ImGui::Button("查看订单", ImVec2(160, 35)))  {  showOrderWindow = true;  refreshOrders();  }  ImGui::PopStyleColor(2);  ImGui::PopStyleVar();  }  ImGui::EndChild();  ImGui::PopStyleColor();  ImGui::PopStyleVar();  ImGui::Spacing(); // 商品列表  if (strlen(searchBuffer) > 0)  {  // 如果有搜索关键词，显示搜索结果  if (!searchResults.empty())  {  renderProductList(searchResults, "搜索结果");  }  else  {  ImGui::Text("没有找到符合条件的商品");  }  }  else  {  // 没有搜索关键词，显示所有商品  renderProductList(allProducts, "所有商品");  }  // 购买商品区域  if (isLoggedIn && userType == Protocol::UserType::CUSTOMER)  {  ImGui::Separator();  ImGui::Text("购买商品");  ImGui::SetNextItemWidth(250);  ImGui::InputText("商品名称", buyProductName, IM\_ARRAYSIZE(buyProductName));  ImGui::SetNextItemWidth(150);  ImGui::InputInt("购买数量", &buyQuantity);  if (buyQuantity < 1)  buyQuantity = 1;  if (ImGui::Button("直接购买", ImVec2(100, 30)))  {  if (strlen(buyProductName) == 0)  {  ImGui::OpenPopup("购买失败");  }  else  {  // 查找商品  Protocol::ProductData \*targetProduct = nullptr;  for (auto &product : allProducts)  {  if (product.name == buyProductName)  {  targetProduct = &product;  break;  }  }  if (!targetProduct)  {  ImGui::OpenPopup("商品不存在");  }  else if (targetProduct->quantity < buyQuantity)  {  ImGui::OpenPopup("库存不足");  }  else  {  // 首先锁定库存  if (networkClient->lockInventory(targetProduct->id, buyQuantity))  {  // 库存锁定成功，设置确认对话框参数并显示  productToPurchase = \*targetProduct;  quantityToPurchase = buyQuantity;  inventoryLocked = true; // 标记库存已锁定  std::cout << "库存锁定成功，准备购买商品: " << productToPurchase.name << std::endl;  showDirectPurchaseConfirmDialog = true;  // 清空表单数据  memset(buyProductName, 0, sizeof(buyProductName));  buyQuantity = 1;  }  else  {  // 库存锁定失败  ImGui::OpenPopup("库存锁定失败");  }  }  }  }  ImGui::SameLine();  if (ImGui::Button("加入购物车", ImVec2(120, 30)))  {  if (strlen(buyProductName) == 0)  {  ImGui::OpenPopup("购买失败");  }  else  {  // 查找商品  Protocol::ProductData \*targetProduct = nullptr;  for (auto &product : allProducts)  {  if (product.name == buyProductName)  {  targetProduct = &product;  break;  }  }  if (!targetProduct)  {  ImGui::OpenPopup("商品不存在");  }  else if (targetProduct->quantity < buyQuantity)  {  ImGui::OpenPopup("库存不足");  }  else  {  if (networkClient->addToCart(targetProduct->id, buyQuantity))  {  // ImGui::OpenPopup("加入购物车成功");  addToCartSuccessMessage = "商品 \"" + targetProduct->name + "\" 已成功加入购物车！";  showAddToCartSuccessPopup = true;  memset(buyProductName, 0, sizeof(buyProductName));  buyQuantity = 1;  refreshCart();  }  else  {  ImGui::OpenPopup("加入购物车失败");  }  }  }  }  }  // 购买相关弹窗  if (ImGui::BeginPopupModal("购买失败", NULL, ImGuiWindowFlags\_AlwaysAutoResize))  {  ImGui::Text("请输入要购买的商品名称！");  if (ImGui::Button("确定", ImVec2(120, 0)))  {  ImGui::CloseCurrentPopup();  }  ImGui::EndPopup();  }  if (ImGui::BeginPopupModal("商品不存在", NULL, ImGuiWindowFlags\_AlwaysAutoResize))  {  ImGui::Text("找不到商品 \"%s\"！", buyProductName);  if (ImGui::Button("确定", ImVec2(120, 0)))  {  ImGui::CloseCurrentPopup();  }  ImGui::EndPopup();  }  if (ImGui::BeginPopupModal("库存不足", NULL, ImGuiWindowFlags\_AlwaysAutoResize))  {  ImGui::Text("商品 \"%s\" 库存不足！", buyProductName);  ImGui::Text("您要购买 %d 件，但库存只有 %d 件。", buyQuantity, 0); // 需要找到库存数量  if (ImGui::Button("确定", ImVec2(120, 0)))  {  ImGui::CloseCurrentPopup();  }  ImGui::EndPopup();  }  if (ImGui::BeginPopupModal("余额不足", NULL, ImGuiWindowFlags\_AlwaysAutoResize))  {  ImGui::Text("您的余额不足以完成此次购买！");  ImGui::Text("请先充值后再试。");  if (ImGui::Button("确定", ImVec2(120, 0)))  {  ImGui::CloseCurrentPopup();  }  ImGui::EndPopup();  }  if (ImGui::BeginPopupModal("库存锁定失败", NULL, ImGuiWindowFlags\_AlwaysAutoResize))  {  ImGui::Text("库存锁定失败！");  ImGui::Text("商品可能库存不足或已售完，请稍后重试。");  if (ImGui::Button("确定", ImVec2(120, 0)))  {  ImGui::CloseCurrentPopup();  }  ImGui::EndPopup();  }  // if (ImGui::BeginPopupModal("购买成功", NULL, ImGuiWindowFlags\_AlwaysAutoResize))  // {  // ImGui::Text("购买成功！");  // if (ImGui::Button("确定", ImVec2(120, 0)))  // {  // ImGui::CloseCurrentPopup();  // }  // ImGui::EndPopup();  // }  // if (ImGui::BeginPopupModal("加入购物车成功", NULL, ImGuiWindowFlags\_AlwaysAutoResize))  // {  // ImGui::Text("%s", addToCartSuccessMessage.c\_str());  // ImGui::Separator();  // if (ImGui::Button("查看购物车", ImVec2(120, 30)))  // {  // showCartWindow = true;  // showAddToCartSuccessPopup = false;  // ImGui::CloseCurrentPopup();  // }  // ImGui::SameLine();  // if (ImGui::Button("继续购物", ImVec2(120, 30)))  // {  // showAddToCartSuccessPopup = false;  // ImGui::CloseCurrentPopup();  // }  // ImGui::EndPopup();  // }  if (ImGui::BeginPopupModal("加入购物车失败", NULL, ImGuiWindowFlags\_AlwaysAutoResize))  {  ImGui::Text("加入购物车失败，请重试！");  if (ImGui::Button("确定", ImVec2(120, 0)))  {  ImGui::CloseCurrentPopup();  }  ImGui::EndPopup();  }  if (ImGui::BeginPopupModal("购买失败-cart", NULL, ImGuiWindowFlags\_AlwaysAutoResize))  {  ImGui::Text("购买失败：购物车操作异常！");  if (ImGui::Button("确定", ImVec2(120, 0)))  {  ImGui::CloseCurrentPopup();  }  ImGui::EndPopup();  }  if (ImGui::BeginPopupModal("购买失败-add", NULL, ImGuiWindowFlags\_AlwaysAutoResize))  {  ImGui::Text("购买失败：无法加入购物车！");  if (ImGui::Button("确定", ImVec2(120, 0)))  {  ImGui::CloseCurrentPopup();  }  ImGui::EndPopup();  }  }  // 渲染商家商城  void ClientUI::renderStoreSeller()  {  if (!isLoggedIn || userType != Protocol::UserType::SELLER)  return;  // 显示选项卡  ImGui::BeginTabBar("SellerTabs");  if (ImGui::BeginTabItem("我的商品"))  {  ImGui::BeginChild("MyProducts", ImVec2(0, height - 200));  std::vector<Protocol::ProductData> sellerProducts;  for (const auto &product : allProducts)  {  if (product.sellerUsername == currentUser.username)  {  sellerProducts.push\_back(product);  }  }  renderProductList(sellerProducts, "我的商品");  ImGui::EndChild();  ImGui::EndTabItem();  }  if (ImGui::BeginTabItem("添加商品"))  {  ImGui::BeginChild("AddProduct", ImVec2(0, height - 200));  ImGui::SetNextItemWidth(250);  ImGui::InputText("商品名称", newProductName, IM\_ARRAYSIZE(newProductName));  ImGui::SetNextItemWidth(350);  ImGui::InputTextMultiline("商品描述", newProductDesc, IM\_ARRAYSIZE(newProductDesc));  ImGui::SetNextItemWidth(150);  ImGui::InputFloat("商品价格", &newProductPrice, 1.0f, 10.0f, "CNY%.2f");  if (newProductPrice < 0)  newProductPrice = 0;  ImGui::SetNextItemWidth(150);  ImGui::InputInt("库存数量", &newProductQuantity);  if (newProductQuantity < 0)  newProductQuantity = 0;  const char \*productTypes[] = {"图书", "服装", "食品", "其他"};  ImGui::SetNextItemWidth(150);  ImGui::Combo("商品类型", &newProductType, productTypes, IM\_ARRAYSIZE(productTypes));  // 根据不同的商品类型显示不同的输入字段  if (newProductType == 0)  { // 图书  ImGui::SetNextItemWidth(250);  ImGui::InputText("作者", newProductAttr1, IM\_ARRAYSIZE(newProductAttr1));  ImGui::SetNextItemWidth(250);  ImGui::InputText("ISBN", newProductAttr2, IM\_ARRAYSIZE(newProductAttr2));  }  else if (newProductType == 1)  { // 服装  ImGui::SetNextItemWidth(150);  ImGui::InputText("尺寸", newProductAttr1, IM\_ARRAYSIZE(newProductAttr1));  ImGui::SetNextItemWidth(150);  ImGui::InputText("颜色", newProductAttr2, IM\_ARRAYSIZE(newProductAttr2));  }  else if (newProductType == 2)  { // 食品  ImGui::SetNextItemWidth(150);  ImGui::InputText("过期日期", newProductAttr1, IM\_ARRAYSIZE(newProductAttr1));  }  else  {  ImGui::SetNextItemWidth(250);  ImGui::InputText("分类名称", newProductAttr1, IM\_ARRAYSIZE(newProductAttr1));  }  if (ImGui::Button("添加商品", ImVec2(120, 30)))  {  if (strlen(newProductName) == 0)  {  ImGui::OpenPopup("商品添加失败");  }  else  {  Protocol::ProductData newProduct;  newProduct.name = newProductName;  newProduct.description = newProductDesc;  newProduct.price = newProductPrice;  newProduct.originalPrice = newProductPrice; // 设置原价  newProduct.quantity = newProductQuantity;  newProduct.discountRate = 0.0; // 初始折扣为0  newProduct.sellerUsername = currentUser.username;  newProduct.id = newProductName; // 使用商品名作为ID  // 根据类型设置type和属性  if (newProductType == 0)  { // 图书  newProduct.type = "Book";  newProduct.attributes["author"] = newProductAttr1;  newProduct.attributes["isbn"] = newProductAttr2;  }  else if (newProductType == 1)  { // 服装  newProduct.type = "Clothing";  newProduct.attributes["size"] = newProductAttr1;  newProduct.attributes["color"] = newProductAttr2;  }  else if (newProductType == 2)  { // 食品  newProduct.type = "Food";  newProduct.attributes["expirationDate"] = newProductAttr1;  }  else  { // 其他（自定义商品）  newProduct.type = "Generic";  newProduct.attributes["categoryTag"] = newProductAttr1;  }  if (addProduct(newProduct))  {  ImGui::OpenPopup("商品添加成功");  memset(newProductName, 0, sizeof(newProductName));  memset(newProductDesc, 0, sizeof(newProductDesc));  memset(newProductAttr1, 0, sizeof(newProductAttr1));  memset(newProductAttr2, 0, sizeof(newProductAttr2));  newProductPrice = 0.0f;  newProductQuantity = 0;  refreshProducts();  }  else  {  ImGui::OpenPopup("商品添加失败");  }  }  }  if (ImGui::BeginPopupModal("商品添加成功", NULL, ImGuiWindowFlags\_AlwaysAutoResize))  {  ImGui::Text("商品添加成功！");  if (ImGui::Button("确定", ImVec2(120, 0)))  {  ImGui::CloseCurrentPopup();  }  ImGui::EndPopup();  }  if (ImGui::BeginPopupModal("商品添加失败", NULL, ImGuiWindowFlags\_AlwaysAutoResize))  {  ImGui::Text("商品添加失败！请填写商品名称。");  if (ImGui::Button("确定", ImVec2(120, 0)))  {  ImGui::CloseCurrentPopup();  }  ImGui::EndPopup();  }  ImGui::EndChild();  ImGui::EndTabItem();  }  if (ImGui::BeginTabItem("管理商品"))  {  ImGui::BeginChild("ManageProducts", ImVec2(0, height - 200));  ImGui::Text("修改商品价格");  static char priceProductName[128] = {0};  static float newPrice = 0.0f;  ImGui::SetNextItemWidth(250);  ImGui::InputText("商品名称##price", priceProductName, IM\_ARRAYSIZE(priceProductName));  ImGui::SetNextItemWidth(150);  ImGui::InputFloat("新价格", &newPrice, 1.0f, 10.0f, "CNY%.2f");  if (newPrice < 0)  newPrice = 0;  if (ImGui::Button("修改价格", ImVec2(120, 30)))  {  if (strlen(priceProductName) == 0)  {  ImGui::OpenPopup("修改失败");  }  else  {  if (networkClient && networkClient->manageProductPrice(priceProductName, newPrice))  {  ImGui::OpenPopup("价格修改成功");  memset(priceProductName, 0, sizeof(priceProductName));  newPrice = 0.0f;  refreshProducts(); // 刷新商品列表  }  else  {  ImGui::OpenPopup("价格修改失败");  }  }  }  if (ImGui::BeginPopupModal("修改失败", NULL, ImGuiWindowFlags\_AlwaysAutoResize))  {  ImGui::Text("请输入商品名称！");  if (ImGui::Button("确定", ImVec2(120, 0)))  {  ImGui::CloseCurrentPopup();  }  ImGui::EndPopup();  }  if (ImGui::BeginPopupModal("价格修改成功", NULL, ImGuiWindowFlags\_AlwaysAutoResize))  {  ImGui::Text("商品价格修改成功！");  if (ImGui::Button("确定", ImVec2(120, 0)))  {  ImGui::CloseCurrentPopup();  }  ImGui::EndPopup();  }  ImGui::Separator();  ImGui::Text("修改商品库存");  static char qtyProductName[128] = {0};  static int newQty = 0;  ImGui::SetNextItemWidth(250);  ImGui::InputText("商品名称##qty", qtyProductName, IM\_ARRAYSIZE(qtyProductName));  ImGui::SetNextItemWidth(150);  ImGui::InputInt("新库存", &newQty);  if (newQty < 0)  newQty = 0;  if (ImGui::Button("修改库存", ImVec2(120, 30)))  {  if (strlen(qtyProductName) == 0)  {  ImGui::OpenPopup("修改失败");  }  else  {  if (networkClient && networkClient->manageProductQuantity(qtyProductName, newQty))  {  ImGui::OpenPopup("库存修改成功");  memset(qtyProductName, 0, sizeof(qtyProductName));  newQty = 0;  refreshProducts(); // 刷新商品列表  }  else  {  ImGui::OpenPopup("库存修改失败");  }  }  }  if (ImGui::BeginPopupModal("库存修改成功", NULL, ImGuiWindowFlags\_AlwaysAutoResize))  {  ImGui::Text("商品库存修改成功！");  if (ImGui::Button("确定", ImVec2(120, 0)))  {  ImGui::CloseCurrentPopup();  }  ImGui::EndPopup();  }  ImGui::Separator();  ImGui::Text("修改商品折扣");  static char discountProductName[128] = {0};  static float newDiscount = 0.0f;  ImGui::SetNextItemWidth(250);  ImGui::InputText("商品名称##discount", discountProductName, IM\_ARRAYSIZE(discountProductName));  ImGui::SetNextItemWidth(150);  ImGui::SliderFloat("折扣 (%)", &newDiscount, 0.0f, 100.0f, "%.1f%%");  if (ImGui::Button("应用折扣", ImVec2(120, 30)))  {  if (strlen(discountProductName) == 0)  {  ImGui::OpenPopup("修改失败");  }  else  {  if (networkClient && networkClient->manageProductDiscount(discountProductName, newDiscount / 100.0))  {  ImGui::OpenPopup("折扣修改成功");  memset(discountProductName, 0, sizeof(discountProductName));  newDiscount = 0.0f;  refreshProducts(); // 刷新商品列表  }  else  {  ImGui::OpenPopup("折扣修改失败");  }  }  }  if (ImGui::BeginPopupModal("折扣修改成功", NULL, ImGuiWindowFlags\_AlwaysAutoResize))  {  ImGui::Text("商品折扣修改成功！");  if (ImGui::Button("确定", ImVec2(120, 0)))  {  ImGui::CloseCurrentPopup();  }  ImGui::EndPopup();  }  ImGui::Separator();  ImGui::Text("修改商品类型折扣");  static char categoryName[128] = {0};  static float categoryDiscount = 0.0f;  static int selectedCategory = 0;  const char \*categories[] = {"Book", "Clothing", "Food", "其他"};  ImGui::SetNextItemWidth(250);  ImGui::Combo("商品类型", &selectedCategory, categories, IM\_ARRAYSIZE(categories));  // 如果选择"其他"类型，则显示输入框  if (selectedCategory == 3)  { // "其他"选项的索引  ImGui::SetNextItemWidth(250);  ImGui::InputText("自定义类型", categoryName, IM\_ARRAYSIZE(categoryName));  }  ImGui::SetNextItemWidth(150);  ImGui::SliderFloat("类别折扣 (%)", &categoryDiscount, 0.0f, 100.0f, "%.1f%%");  if (ImGui::Button("应用类别折扣", ImVec2(140, 30)))  {  std::string category;  if (selectedCategory == 3 && strlen(categoryName) > 0)  {  // 如果选择"其他"且输入了自定义类型  category = categoryName;  }  else  {  category = categories[selectedCategory];  }  if (networkClient && networkClient->applyCategoryDiscount(category, categoryDiscount / 100.0))  {  ImGui::OpenPopup("类别折扣修改成功");  categoryDiscount = 0.0f;  if (selectedCategory == 3)  {  memset(categoryName, 0, sizeof(categoryName));  }  refreshProducts(); // 刷新商品列表  }  else  {  ImGui::OpenPopup("类别折扣修改失败");  }  }  if (ImGui::BeginPopupModal("类别折扣修改成功", NULL, ImGuiWindowFlags\_AlwaysAutoResize))  {  ImGui::Text("商品类型折扣修改成功！");  if (ImGui::Button("确定", ImVec2(120, 0)))  {  ImGui::CloseCurrentPopup();  }  ImGui::EndPopup();  }  if (ImGui::BeginPopupModal("价格修改失败", NULL, ImGuiWindowFlags\_AlwaysAutoResize))  {  ImGui::Text("价格修改失败！请检查商品名称和权限。");  if (ImGui::Button("确定", ImVec2(120, 0)))  {  ImGui::CloseCurrentPopup();  }  ImGui::EndPopup();  }  if (ImGui::BeginPopupModal("库存修改失败", NULL, ImGuiWindowFlags\_AlwaysAutoResize))  {  ImGui::Text("库存修改失败！请检查商品名称和权限。");  if (ImGui::Button("确定", ImVec2(120, 0)))  {  ImGui::CloseCurrentPopup();  }  ImGui::EndPopup();  }  if (ImGui::BeginPopupModal("折扣修改失败", NULL, ImGuiWindowFlags\_AlwaysAutoResize))  {  ImGui::Text("折扣修改失败！请检查商品名称和权限。");  if (ImGui::Button("确定", ImVec2(120, 0)))  {  ImGui::CloseCurrentPopup();  }  ImGui::EndPopup();  }  if (ImGui::BeginPopupModal("类别折扣修改失败", NULL, ImGuiWindowFlags\_AlwaysAutoResize))  {  ImGui::Text("类别折扣修改失败！请检查分类名称和权限。");  if (ImGui::Button("确定", ImVec2(120, 0)))  {  ImGui::CloseCurrentPopup();  }  ImGui::EndPopup();  }  ImGui::EndChild();  ImGui::EndTabItem();  }  ImGui::EndTabBar();  }  // 渲染商品列表  void ClientUI::renderProductList(const std::vector<Protocol::ProductData> &products, const std::string &title)  {  ImGui::Text("%s (%d 件商品)", title.c\_str(), (int)products.size());  if (products.empty())  {  ImGui::Text("暂无商品");  return;  }  // 商品表格  if (ImGui::BeginTable("ProductTable", 6, ImGuiTableFlags\_Borders | ImGuiTableFlags\_RowBg))  {  ImGui::TableSetupColumn("商品名称", ImGuiTableColumnFlags\_WidthFixed, 100.0f);  ImGui::TableSetupColumn("价格", ImGuiTableColumnFlags\_WidthFixed, 200.0f);  ImGui::TableSetupColumn("库存", ImGuiTableColumnFlags\_WidthFixed, 120.0f);  ImGui::TableSetupColumn("商家", ImGuiTableColumnFlags\_WidthFixed, 80.0f);  ImGui::TableSetupColumn("类型", ImGuiTableColumnFlags\_WidthFixed, 100.0f);  ImGui::TableSetupColumn("操作", ImGuiTableColumnFlags\_WidthFixed, 150.0f);  ImGui::TableHeadersRow();  for (const auto &product : products)  {  ImGui::TableNextRow();  ImGui::TableSetColumnIndex(0);  ImGui::Text("%s", product.name.c\_str());  ImGui::TableSetColumnIndex(1);  // 显示价格信息，包含折扣  if (product.discountRate > 0.0)  {  // 有折扣，显示原价和现价  // ImGui::TextColored(ImVec4(0.7f, 0.7f, 0.7f, 1.0f), "CNY%.2f", product.originalPrice);  // ImGui::SameLine();  ImGui::TextColored(ImVec4(0.9f, 0.3f, 0.3f, 1.0f), "CNY%.2f", product.price);  ImGui::SameLine();  ImGui::TextColored(ImVec4(0.2f, 0.8f, 0.2f, 1.0f), "(%.0f%%折)", (1 - product.discountRate) \* 100);  }  else  {  // 无折扣，正常显示价格  ImGui::Text("CNY%.2f", product.price);  }  ImGui::TableSetColumnIndex(2);  ImGui::Text("%d", product.quantity);  ImGui::TableSetColumnIndex(3);  ImGui::Text("%s", product.sellerUsername.c\_str());  ImGui::TableSetColumnIndex(4);  ImGui::Text("%s", product.type.c\_str());  ImGui::TableSetColumnIndex(5);  if (isLoggedIn && userType == Protocol::UserType::CUSTOMER)  {  std::string buttonId = "加入购物车##" + product.id;  // 检查库存状态，如果没有库存则禁用按钮  bool hasStock = product.quantity > 0;  if (!hasStock)  {  ImGui::PushStyleColor(ImGuiCol\_Button, ImVec4(0.5f, 0.5f, 0.5f, 1.0f));  ImGui::PushStyleColor(ImGuiCol\_ButtonHovered, ImVec4(0.5f, 0.5f, 0.5f, 1.0f));  ImGui::PushStyleColor(ImGuiCol\_ButtonActive, ImVec4(0.5f, 0.5f, 0.5f, 1.0f));  }  if (ImGui::Button(hasStock ? buttonId.c\_str() : ("缺货##" + product.id).c\_str()) && hasStock)  {  addProductToCart(product);  }  if (!hasStock)  {  ImGui::PopStyleColor(3);  if (ImGui::IsItemHovered())  {  ImGui::SetTooltip("商品缺货，暂时无法购买");  }  }  }  }  ImGui::EndTable();  }  }  // 渲染购物车窗口  void ClientUI::renderCartWindow()  {  if (!isLoggedIn || userType != Protocol::UserType::CUSTOMER)  return;  ImGui::SetNextWindowPos(ImVec2(width / 2 - 400, height / 2 - 250));  ImGui::SetNextWindowSize(ImVec2(800, 500));  if (ImGui::Begin("购物车", &showCartWindow, ImGuiWindowFlags\_NoCollapse))  {  if (cartItems.empty())  {  ImGui::TextColored(ImVec4(0.7f, 0.7f, 0.7f, 1.0f), "购物车为空");  ImGui::Spacing();  ImGui::Text("去商城添加您喜欢的商品吧！");  }  else  {  if (ImGui::BeginTable("CartTable", 6, ImGuiTableFlags\_Borders | ImGuiTableFlags\_RowBg | ImGuiTableFlags\_ScrollY))  {  ImGui::TableSetupColumn("商品名称", ImGuiTableColumnFlags\_WidthFixed, 150.0f);  ImGui::TableSetupColumn("数量", ImGuiTableColumnFlags\_WidthFixed, 150.0f); // 增加数量列宽度从80到100  ImGui::TableSetupColumn("单价", ImGuiTableColumnFlags\_WidthFixed, 80.0f);  ImGui::TableSetupColumn("小计", ImGuiTableColumnFlags\_WidthFixed, 80.0f);  ImGui::TableSetupColumn("商家", ImGuiTableColumnFlags\_WidthFixed, 100.0f);  ImGui::TableSetupColumn("操作", ImGuiTableColumnFlags\_WidthFixed, 120.0f);  ImGui::TableHeadersRow();  double totalAmount = 0.0;  std::vector<int> itemsToRemove;  for (size\_t i = 0; i < cartItems.size(); ++i)  {  const auto &item = cartItems[i];  ImGui::TableNextRow();  ImGui::TableSetColumnIndex(0);  ImGui::Text("%s", item.productName.c\_str());  ImGui::TableSetColumnIndex(1);  // 可编辑的数量输入框 - 增加宽度使其更美观  ImGui::PushID(static\_cast<int>(i));  int quantity = item.quantity;  ImGui::SetNextItemWidth(120.0f); // 增加宽度从60到75  if (ImGui::InputInt("##qty", &quantity, 1, 10)) // 修复步长参数：slow\_step=1, fast\_step=10  {  if (quantity > 0)  {  // 检查库存  Protocol::ProductData \*product = nullptr;  for (auto &p : allProducts)  {  if (p.id == item.productId)  {  product = &p;  break;  }  }  if (product && product->quantity >= quantity)  {  updateCartQuantity(item.productId, quantity);  // 立即更新本地购物车以反映更改  refreshCart();  }  else if (product)  {  ImGui::OpenPopup("库存不足");  }  else  {  ImGui::OpenPopup("商品不存在");  }  }  else if (quantity == 0)  {  // 数量为0时移除商品  itemsToRemove.push\_back(static\_cast<int>(i));  }  }  ImGui::PopID();  ImGui::TableSetColumnIndex(2);  // 显示当前的折扣后价格，而不是加入时的价格  double currentPrice = item.priceAtAddition; // 默认使用加入时价格  bool priceChanged = false;  // 查找当前商品的实际价格  for (const auto &product : allProducts)  {  if (product.id == item.productId)  {  currentPrice = product.price; // 使用当前折扣后价格  priceChanged = (currentPrice != item.priceAtAddition);  break;  }  }  if (priceChanged)  {  // 价格有变化，显示当前价格并提示  ImGui::TextColored(ImVec4(0.9f, 0.3f, 0.3f, 1.0f), "CNY%.2f", currentPrice);  if (ImGui::IsItemHovered())  {  ImGui::SetTooltip("原价格: CNY%.2f", item.priceAtAddition);  }  }  else  {  ImGui::Text("CNY%.2f", currentPrice);  }  ImGui::TableSetColumnIndex(3);  double subtotal = currentPrice \* item.quantity;  ImGui::Text("CNY%.2f", subtotal);  totalAmount += subtotal;  ImGui::TableSetColumnIndex(4);  ImGui::Text("%s", item.sellerUsername.c\_str());  ImGui::TableSetColumnIndex(5);  std::string removeButtonId = "移除##" + item.productId;  if (ImGui::Button(removeButtonId.c\_str(), ImVec2(50, 25)))  {  itemsToRemove.push\_back(static\_cast<int>(i));  }  }  ImGui::EndTable();  // 处理要删除的商品（从后往前删除，避免索引问题）  for (int i = itemsToRemove.size() - 1; i >= 0; i--)  {  int idx = itemsToRemove[i];  if (idx >= 0 && idx < cartItems.size())  {  removeFromCart(cartItems[idx].productId);  }  }  ImGui::Separator();  ImGui::Text("总计: CNY%.2f", totalAmount);  // 结算和操作按钮  if (ImGui::Button("结算", ImVec2(100, 30)))  {  if (totalAmount > 0)  {  // 检查用户余额  if (currentUser.balance < totalAmount)  {  ImGui::OpenPopup("余额不足");  }  else  {  showCartCheckoutConfirmDialog = true;  }  }  }  ImGui::SameLine();  if (ImGui::Button("继续购物", ImVec2(100, 30)))  {  showCartWindow = false;  }  ImGui::SameLine();  if (ImGui::Button("清空购物车", ImVec2(120, 30)))  {  if (!cartItems.empty())  {  ImGui::OpenPopup("确认清空");  }  }  // 余额不足弹窗  if (ImGui::BeginPopupModal("余额不足", NULL, ImGuiWindowFlags\_AlwaysAutoResize))  {  ImGui::Text("您的余额不足，需要充值。");  ImGui::Text("当前余额: CNY%.2f", currentUser.balance);  ImGui::Text("订单金额: CNY%.2f", totalAmount);  ImGui::Text("还需充值: CNY%.2f", totalAmount - currentUser.balance);  if (ImGui::Button("去充值", ImVec2(120, 0)))  {  showRechargeDialog = true;  ImGui::CloseCurrentPopup();  }  ImGui::SameLine();  if (ImGui::Button("取消", ImVec2(120, 0)))  {  ImGui::CloseCurrentPopup();  }  ImGui::EndPopup();  }  // 库存不足弹窗  if (ImGui::BeginPopupModal("库存不足", NULL, ImGuiWindowFlags\_AlwaysAutoResize))  {  ImGui::Text("商品库存不足，请调整购买数量。");  if (ImGui::Button("确定", ImVec2(120, 0)))  {  ImGui::CloseCurrentPopup();  refreshCart(); // 刷新购物车显示正确数量  }  ImGui::EndPopup();  }  // 商品不存在弹窗  if (ImGui::BeginPopupModal("商品不存在", NULL, ImGuiWindowFlags\_AlwaysAutoResize))  {  ImGui::Text("商品可能已下架，请刷新购物车。");  if (ImGui::Button("确定", ImVec2(120, 0)))  {  ImGui::CloseCurrentPopup();  refreshCart();  }  ImGui::EndPopup();  } // 确认清空购物车弹窗  if (ImGui::BeginPopupModal("确认清空", NULL, ImGuiWindowFlags\_AlwaysAutoResize))  {  ImGui::Text("确定要清空购物车吗？");  ImGui::Text("这将删除所有已添加的商品。");  if (ImGui::Button("确定", ImVec2(120, 0)))  {  if (networkClient && networkClient->clearCart())  {  cartItems.clear();  setStatus("购物车已清空");  }  ImGui::CloseCurrentPopup();  }  ImGui::SameLine();  if (ImGui::Button("取消", ImVec2(120, 0)))  {  ImGui::CloseCurrentPopup();  }  ImGui::EndPopup();  }  // 加入购物车失败弹窗（用于产品列表中的加入购物车按钮）  if (ImGui::BeginPopupModal("加入购物车失败", NULL, ImGuiWindowFlags\_AlwaysAutoResize))  {  ImGui::Text("加入购物车失败，请重试！");  ImGui::Text("可能是网络错误或商品已下架。");  if (ImGui::Button("确定", ImVec2(120, 0)))  {  ImGui::CloseCurrentPopup();  }  ImGui::EndPopup();  }  }  }  }  ImGui::End();  }  // 渲染订单窗口  void ClientUI::renderOrderWindow()  {  if (!isLoggedIn || userType != Protocol::UserType::CUSTOMER)  return;  // 窗口定位居中，方便用户查看  ImGui::SetNextWindowPos(ImVec2(width / 2 - 300, height / 2 - 200), ImGuiCond\_Once);  ImGui::SetNextWindowSize(ImVec2(600, 450), ImGuiCond\_Once);  if (ImGui::Begin("我的订单", &showOrderWindow, ImGuiWindowFlags\_NoCollapse))  {  static bool showPending = true;  static bool showCompleted = true;  static int selectedOrderIndex = -1;  // 过滤选项  ImGui::Checkbox("显示待处理订单", &showPending);  ImGui::SameLine();  ImGui::Checkbox("显示已完成订单", &showCompleted);  // 刷新按钮  if (ImGui::Button("刷新订单列表", ImVec2(150, 30)))  {  refreshUserOrders();  }  ImGui::Separator();  // 创建子窗口显示订单列表  ImGui::BeginChild("OrdersList", ImVec2(0, 250), true);  if (userOrders.empty())  {  ImGui::TextColored(ImVec4(0.7f, 0.7f, 0.7f, 1.0f), "暂无订单");  ImGui::TextColored(ImVec4(0.7f, 0.7f, 0.7f, 1.0f), "请点击\"刷新订单列表\"按钮加载订单");  }  else  {  // 分离待处理和已完成订单  std::vector<Protocol::OrderData> pendingOrders, completedOrders;  for (const auto &order : userOrders)  {  if (order.status == Protocol::OrderStatus::PENDING)  {  pendingOrders.push\_back(order);  }  else  {  completedOrders.push\_back(order);  }  }  // 显示待处理订单  if (showPending && !pendingOrders.empty())  {  if (ImGui::CollapsingHeader("待处理订单", ImGuiTreeNodeFlags\_DefaultOpen))  {  for (int i = 0; i < pendingOrders.size(); i++)  {  // 修复金额显示格式  std::ostringstream amountStream;  amountStream << std::fixed << std::setprecision(2) << pendingOrders[i].totalAmount;  std::string orderIdPart = pendingOrders[i].orderId.length() >= 18 ? pendingOrders[i].orderId.substr(4, 14) : pendingOrders[i].orderId;  std::string displayName = "订单 " + orderIdPart + "... (CNY" + amountStream.str() + ")";  std::string selectableId = displayName + "##pending\_" + std::to\_string(i);  if (ImGui::Selectable(displayName.c\_str(), selectedOrderIndex == i))  {  selectedOrderIndex = i;  selectedOrder = pendingOrders[i];  ImGui::OpenPopup("订单详情");  }  }  }  }  // 显示已完成订单  if (showCompleted && !completedOrders.empty())  {  if (ImGui::CollapsingHeader("已完成订单", ImGuiTreeNodeFlags\_DefaultOpen))  {  for (int i = 0; i < completedOrders.size(); i++)  {  int index = i + pendingOrders.size();  std::string displayName = "订单 " + completedOrders[i].orderId.substr(0, 18) + "... (CNY" +  std::to\_string(completedOrders[i].totalAmount).substr(0, 6) + ")";  if (ImGui::Selectable(displayName.c\_str(), selectedOrderIndex == index))  {  selectedOrderIndex = index;  selectedOrder = completedOrders[i];  ImGui::OpenPopup("订单详情");  }  }  }  }  }  // 订单详情弹窗  if (ImGui::BeginPopupModal("订单详情", NULL, ImGuiWindowFlags\_AlwaysAutoResize))  {  renderOrderDetails(selectedOrder);  ImGui::EndPopup();  }  ImGui::EndChild();  ImGui::Separator();  // 底部按钮  if (ImGui::Button("关闭窗口", ImVec2(120, 30)))  {  showOrderWindow = false;  }  }  ImGui::End();  }  // 渲染状态消息  void ClientUI::renderStatusMessages()  {  if (!statusMessage.empty())  {  ImGui::SetNextWindowPos(ImVec2(width / 2 - 150, 50));  ImGui::SetNextWindowBgAlpha(0.8f);  if (ImGui::Begin("状态消息", nullptr, ImGuiWindowFlags\_NoDecoration | ImGuiWindowFlags\_AlwaysAutoResize))  {  ImGui::TextColored(ImVec4(0, 1, 0, 1), "%s", statusMessage.c\_str());  }  ImGui::End();  }  if (showErrorMessage && !errorMessage.empty())  {  ImGui::SetNextWindowPos(ImVec2(width / 2 - 150, 50));  ImGui::SetNextWindowBgAlpha(0.8f);  if (ImGui::Begin("错误消息", nullptr, ImGuiWindowFlags\_NoDecoration | ImGuiWindowFlags\_AlwaysAutoResize))  {  ImGui::TextColored(ImVec4(1, 0, 0, 1), "%s", errorMessage.c\_str());  if (ImGui::Button("确定"))  {  showErrorMessage = false;  errorMessage.clear();  }  }  ImGui::End();  }  }  // 渲染订单详情  void ClientUI::renderOrderDetails(const Protocol::OrderData &order)  {  // 订单基本信息  ImGui::Text("订单编号: %s", order.orderId.c\_str());  ImGui::Text("订单总额: CNY %.2f", order.totalAmount);  std::string statusStr;  switch (order.status)  {  case Protocol::OrderStatus::PENDING:  statusStr = "待处理";  break;  case Protocol::OrderStatus::COMPLETED:  statusStr = "已完成";  break;  case Protocol::OrderStatus::CANCELLED\_STOCK:  statusStr = "库存不足取消";  break;  case Protocol::OrderStatus::CANCELLED\_FUNDS:  statusStr = "余额不足取消";  break;  case Protocol::OrderStatus::CANCELLED\_USER:  statusStr = "用户取消";  break;  default:  statusStr = "未知";  break;  }  ImGui::Text("订单状态: %s", statusStr.c\_str());  // 时间格式化显示  if (order.timestamp > 0)  {  time\_t timestamp = static\_cast<time\_t>(order.timestamp);  char timeBuffer[64];  struct tm \*timeInfo = localtime(&timestamp);  strftime(timeBuffer, sizeof(timeBuffer), "%Y-%m-%d %H:%M:%S", timeInfo);  ImGui::Text("下单时间: %s", timeBuffer);  }  else  {  ImGui::Text("下单时间: 未知");  }  ImGui::Separator();  ImGui::Text("订单商品:");  // 订单商品列表  if (ImGui::BeginTable("OrderItemTable", 4, ImGuiTableFlags\_Borders | ImGuiTableFlags\_RowBg))  {  ImGui::TableSetupColumn("商品名称");  ImGui::TableSetupColumn("数量");  ImGui::TableSetupColumn("单价");  ImGui::TableSetupColumn("商家");  ImGui::TableHeadersRow();  if (order.items.empty())  {  ImGui::TableNextRow();  ImGui::TableSetColumnIndex(0);  ImGui::TextColored(ImVec4(0.7f, 0.7f, 0.7f, 1.0f), "（无商品数据）");  }  else  {  for (const auto &item : order.items)  {  ImGui::TableNextRow();  ImGui::TableSetColumnIndex(0);  ImGui::TextUnformatted(item.productName.c\_str());  ImGui::TableSetColumnIndex(1);  ImGui::Text("%d", item.quantity);  ImGui::TableSetColumnIndex(2);  ImGui::Text("CNY %.2f", item.priceAtAddition);  ImGui::TableSetColumnIndex(3);  ImGui::TextUnformatted(item.sellerUsername.c\_str());  }  }  ImGui::EndTable();  }  ImGui::Separator();  // 订单操作按钮  if (order.status == Protocol::OrderStatus::PENDING)  {  if (ImGui::Button("取消订单", ImVec2(120, 30)))  {  ImGui::OpenPopup("订单取消确认");  }  if (ImGui::BeginPopupModal("订单取消确认", NULL, ImGuiWindowFlags\_AlwaysAutoResize))  {  ImGui::Text("确定要取消此订单吗？操作无法撤销。");  if (ImGui::Button("确认取消", ImVec2(120, 30)))  {  // TODO: 实现取消订单功能  if (networkClient)  {  // networkClient->cancelOrder(order.orderId);  }  ImGui::CloseCurrentPopup();  ImGui::CloseCurrentPopup(); // 关闭订单详情弹窗  }  ImGui::SameLine();  if (ImGui::Button("取消", ImVec2(120, 30)))  {  ImGui::CloseCurrentPopup();  }  ImGui::EndPopup();  }  ImGui::SameLine();  }  if (ImGui::Button("关闭", ImVec2(120, 30)))  {  ImGui::CloseCurrentPopup();  }  }  // 刷新用户订单  void ClientUI::refreshUserOrders()  {  if (!networkClient || !isLoggedIn)  {  return;  }  std::vector<Protocol::OrderData> orders;  if (networkClient->getUserOrders(orders))  {  userOrders = orders;  setStatus("订单列表已刷新");  }  else  {  setError("获取订单列表失败");  }  }  // 业务逻辑方法实现  bool ClientUI::login(const std::string &username, const std::string &password)  {  if (!networkClient)  return false;  Protocol::UserData userData;  std::string errorMessage;  if (networkClient->loginWithError(username, password, userData, errorMessage))  {  currentUser = userData;  userType = userData.userType;  isLoggedIn = true;  return true;  }  else  {  // 设置具体的错误信息  setError(errorMessage);  return false;  }  }  bool ClientUI::registerUser(const std::string &username, const std::string &password, Protocol::UserType userType)  {  if (!networkClient)  return false;  return networkClient->registerUser(username, password, userType);  }  void ClientUI::logout()  {  if (networkClient && isLoggedIn)  {  networkClient->logout();  }  isLoggedIn = false;  userType = Protocol::UserType::CUSTOMER;  currentUser = Protocol::UserData();  allProducts.clear();  searchResults.clear();  cartItems.clear();  userOrders.clear();  }  void ClientUI::refreshProducts()  {  if (!networkClient)  return;  std::cout << "刷新产品列表前，当前产品数量: " << allProducts.size() << std::endl;  allProducts.clear();  if (networkClient->getAllProducts(allProducts))  {  std::cout << "产品列表刷新成功，新产品数量: " << allProducts.size() << std::endl;  // 调试：显示前几个产品的库存信息  for (size\_t i = 0; i < std::min((size\_t)3, allProducts.size()); ++i)  {  const auto &prod = allProducts[i];  std::cout << "产品 \"" << prod.name << "\" 库存: " << prod.quantity << std::endl;  }  }  else  {  std::cout << "产品列表刷新失败!" << std::endl;  }  }  void ClientUI::refreshCart()  {  if (!networkClient || !isLoggedIn)  return;  cartItems.clear();  networkClient->getCart(cartItems);  }  void ClientUI::refreshOrders()  {  if (!networkClient || !isLoggedIn)  return;  userOrders.clear();  networkClient->getUserOrders(userOrders);  }  void ClientUI::performSearch()  {  if (!networkClient)  return;  searchResults.clear();  if (strlen(searchBuffer) > 0)  {  networkClient->searchProducts(searchBuffer, searchResults);  }  }  void ClientUI::addProductToCart(const Protocol::ProductData &product)  {  if (!networkClient || !isLoggedIn)  return;  if (networkClient->addToCart(product.id, 1))  {  // 显示成功弹窗  addToCartSuccessMessage = "商品 \"" + product.name + "\" 已成功加入购物车！";  showAddToCartSuccessPopup = true;  refreshCart();  }  else  {  ImGui::OpenPopup("加入购物车失败");  }  }  void ClientUI::updateCartQuantity(const std::string &productId, int newQuantity)  {  if (!networkClient || !isLoggedIn)  return;  if (networkClient->updateCartItem(productId, newQuantity))  {  refreshCart();  }  else  {  setError("更新购物车数量失败，请重试");  refreshCart(); // 刷新以显示正确的数量  }  }  void ClientUI::removeFromCart(const std::string &productId)  {  if (!networkClient || !isLoggedIn)  return;  if (networkClient->removeFromCart(productId))  {  setStatus("商品已从购物车移除！");  refreshCart();  }  }  void ClientUI::checkout()  {  if (!networkClient || !isLoggedIn || cartItems.empty())  return;  std::string orderId;  if (networkClient->createOrder(cartItems, orderId))  {  setStatus("订单创建成功！订单号: %s", orderId.c\_str());  cartItems.clear();  refreshOrders();  }  else  {  setError("订单创建失败！");  }  }  void ClientUI::checkoutWithInventoryLock()  {  if (!networkClient || !isLoggedIn || cartItems.empty())  {  setError("结账失败：未登录或购物车为空");  return;  }  // 第一步：尝试锁定所有购物车商品的库存  if (!networkClient->lockCartInventory(cartItems))  {  setError("库存锁定失败！部分商品可能库存不足或已售完");  return;  }  // 第二步：创建订单  std::string orderId;  bool orderCreated = networkClient->createOrder(cartItems, orderId);  if (orderCreated)  {  // 订单创建成功 - 库存已在服务端正确处理  setStatus("订单创建成功！订单号: %s", orderId.c\_str());  cartItems.clear();  refreshOrders();  refreshCart(); // 刷新购物车显示  // 更新用户余额信息  Protocol::UserData updatedUser;  if (networkClient->getUserInfo(updatedUser))  {  currentUser = updatedUser;  }  }  else  {  // 订单创建失败 - 解锁库存  if (!networkClient->unlockCartInventory(cartItems))  {  setError("警告：订单创建失败且库存解锁失败！请联系客服处理");  }  else  {  setError("订单创建失败！可能是余额不足或其他原因，库存已释放");  }  }  }  bool ClientUI::addProduct(const Protocol::ProductData &product)  {  if (!networkClient)  return false;  return networkClient->addProduct(product);  }  void ClientUI::addToCartByName(const std::string &productName, int quantity)  {  // 根据商品名称找到商品ID  for (const auto &product : allProducts)  {  if (product.name == productName)  {  if (networkClient->addToCart(product.id, quantity))  {  setStatus("商品已加入购物车！");  refreshCart();  }  else  {  setError("加入购物车失败！");  }  return;  }  }  setError("找不到商品: %s", productName.c\_str());  }  void ClientUI::purchaseProduct(const std::string &productName, int quantity)  {  // 直接购买逻辑 - 库存已在确认对话框显示前锁定  for (const auto &product : allProducts)  {  if (product.name == productName)  {  // 库存已锁定，直接创建订单  std::string orderId;  if (networkClient->createDirectOrder(product.id, quantity, orderId))  {  // 购买成功，重置库存锁定状态  inventoryLocked = false;  // 显示购买成功弹窗  directPurchaseSuccessMessage = "购买成功！\n\n商品：" + product.name +  "\n数量：" + std::to\_string(quantity) +  "\n总价：¥" + std::to\_string(product.price \* quantity) +  "\n订单ID：" + orderId;  showDirectPurchaseSuccessPopup = true;  // 立即刷新产品信息以显示最新库存  refreshProducts();  // 更新用户余额信息  Protocol::UserData updatedUser;  if (networkClient->getUserInfo(updatedUser))  {  currentUser = updatedUser;  }  std::cout << "直接购买成功，正在刷新商品信息..." << std::endl;  }  else  {  // 订单创建失败，解锁库存  if (inventoryLocked && !networkClient->unlockInventory(product.id, quantity))  {  setError("警告：购买失败且库存解锁失败！请联系客服处理");  }  else  {  inventoryLocked = false;  setError("直接购买失败！可能是余额不足或其他原因，库存已释放");  }  }  return;  }  }  setError("找不到商品: %s", productName.c\_str());  }  // 网络操作辅助方法  void ClientUI::setStatus(const std::string &message)  {  statusMessage = message;  showErrorMessage = false;  errorMessage.clear();  }  void ClientUI::setError(const std::string &error)  {  errorMessage = error;  showErrorMessage = true;  statusMessage.clear();  }  void ClientUI::setStatus(const char \*format, ...)  {  va\_list args;  va\_start(args, format);  char buffer[512];  vsnprintf(buffer, sizeof(buffer), format, args);  va\_end(args);  statusMessage = std::string(buffer);  showErrorMessage = false;  errorMessage.clear();  }  void ClientUI::setError(const char \*format, ...)  {  va\_list args;  va\_start(args, format);  char buffer[512];  vsnprintf(buffer, sizeof(buffer), format, args);  va\_end(args);  errorMessage = std::string(buffer);  showErrorMessage = true;  statusMessage.clear();  }  void ClientUI::clearMessages()  {  statusMessage.clear();  errorMessage.clear();  showErrorMessage = false;  }  // 设置现代化主题  void ClientUI::setupModernTheme()  {  ImGuiStyle &style = ImGui::GetStyle();  // 设置圆角  style.WindowRounding = 10.0f;  style.ChildRounding = 8.0f;  style.FrameRounding = 6.0f;  style.PopupRounding = 8.0f;  style.ScrollbarRounding = 6.0f;  style.GrabRounding = 6.0f;  style.TabRounding = 6.0f;  // 设置间距  style.WindowPadding = ImVec2(15, 15);  style.FramePadding = ImVec2(12, 8);  style.ItemSpacing = ImVec2(12, 8);  style.ItemInnerSpacing = ImVec2(8, 6);  style.TouchExtraPadding = ImVec2(0, 0);  style.IndentSpacing = 25;  style.ScrollbarSize = 15;  style.GrabMinSize = 10;  // 设置边框  style.WindowBorderSize = 1;  style.ChildBorderSize = 1;  style.PopupBorderSize = 1;  style.FrameBorderSize = 0;  style.TabBorderSize = 0;  if (isDarkTheme)  {  setupDarkTheme();  }  else  {  setupLightTheme();  }  }  void ClientUI::setupDarkTheme()  {  ImVec4 \*colors = ImGui::GetStyle().Colors;  // 主要颜色  colors[ImGuiCol\_Text] = ImVec4(1.00f, 1.00f, 1.00f, 1.00f);  colors[ImGuiCol\_TextDisabled] = ImVec4(0.50f, 0.50f, 0.50f, 1.00f);  colors[ImGuiCol\_WindowBg] = ImVec4(0.13f, 0.14f, 0.15f, 1.00f);  colors[ImGuiCol\_ChildBg] = ImVec4(0.16f, 0.17f, 0.18f, 1.00f);  colors[ImGuiCol\_PopupBg] = ImVec4(0.16f, 0.17f, 0.18f, 0.94f);  // 边框  colors[ImGuiCol\_Border] = ImVec4(0.30f, 0.30f, 0.30f, 0.50f);  colors[ImGuiCol\_BorderShadow] = ImVec4(0.00f, 0.00f, 0.00f, 0.00f);  // 框架背景  colors[ImGuiCol\_FrameBg] = ImVec4(0.20f, 0.21f, 0.22f, 1.00f);  colors[ImGuiCol\_FrameBgHovered] = ImVec4(0.40f, 0.40f, 0.40f, 0.40f);  colors[ImGuiCol\_FrameBgActive] = ImVec4(0.18f, 0.18f, 0.18f, 0.67f);  // 标题栏  colors[ImGuiCol\_TitleBg] = ImVec4(0.04f, 0.04f, 0.04f, 1.00f);  colors[ImGuiCol\_TitleBgActive] = ImVec4(0.16f, 0.29f, 0.48f, 1.00f);  colors[ImGuiCol\_TitleBgCollapsed] = ImVec4(0.00f, 0.00f, 0.00f, 0.51f);  // 滚动条  colors[ImGuiCol\_ScrollbarBg] = ImVec4(0.02f, 0.02f, 0.02f, 0.53f);  colors[ImGuiCol\_ScrollbarGrab] = ImVec4(0.31f, 0.31f, 0.31f, 1.00f);  colors[ImGuiCol\_ScrollbarGrabHovered] = ImVec4(0.41f, 0.41f, 0.41f, 1.00f);  colors[ImGuiCol\_ScrollbarGrabActive] = ImVec4(0.51f, 0.51f, 0.51f, 1.00f);  // 按钮  colors[ImGuiCol\_Button] = ImVec4(0.26f, 0.59f, 0.98f, 0.40f);  colors[ImGuiCol\_ButtonHovered] = ImVec4(0.26f, 0.59f, 0.98f, 1.00f);  colors[ImGuiCol\_ButtonActive] = ImVec4(0.06f, 0.53f, 0.98f, 1.00f);  // 选择  colors[ImGuiCol\_CheckMark] = ImVec4(0.26f, 0.59f, 0.98f, 1.00f);  // Tab  colors[ImGuiCol\_Tab] = ImVec4(0.18f, 0.35f, 0.58f, 0.86f);  colors[ImGuiCol\_TabHovered] = ImVec4(0.26f, 0.59f, 0.98f, 0.80f);  colors[ImGuiCol\_TabActive] = ImVec4(0.20f, 0.41f, 0.68f, 1.00f);  colors[ImGuiCol\_TabUnfocused] = ImVec4(0.07f, 0.10f, 0.15f, 0.97f);  colors[ImGuiCol\_TabUnfocusedActive] = ImVec4(0.14f, 0.26f, 0.42f, 1.00f);  }  void ClientUI::setupLightTheme()  {  ImVec4 \*colors = ImGui::GetStyle().Colors;  // 主要颜色  colors[ImGuiCol\_Text] = ImVec4(0.00f, 0.00f, 0.00f, 1.00f);  colors[ImGuiCol\_TextDisabled] = ImVec4(0.60f, 0.60f, 0.60f, 1.00f);  colors[ImGuiCol\_WindowBg] = ImVec4(0.96f, 0.96f, 0.96f, 1.00f);  colors[ImGuiCol\_ChildBg] = ImVec4(0.98f, 0.98f, 0.98f, 1.00f);  colors[ImGuiCol\_PopupBg] = ImVec4(1.00f, 1.00f, 1.00f, 0.98f);  // 边框  colors[ImGuiCol\_Border] = ImVec4(0.70f, 0.70f, 0.70f, 0.50f);  colors[ImGuiCol\_BorderShadow] = ImVec4(0.00f, 0.00f, 0.00f, 0.00f);  // 框架背景  colors[ImGuiCol\_FrameBg] = ImVec4(1.00f, 1.00f, 1.00f, 1.00f);  colors[ImGuiCol\_FrameBgHovered] = ImVec4(0.80f, 0.80f, 0.80f, 0.40f);  colors[ImGuiCol\_FrameBgActive] = ImVec4(0.85f, 0.85f, 0.85f, 0.67f);  // 按钮  colors[ImGuiCol\_Button] = ImVec4(0.26f, 0.59f, 0.98f, 0.40f);  colors[ImGuiCol\_ButtonHovered] = ImVec4(0.26f, 0.59f, 0.98f, 1.00f);  colors[ImGuiCol\_ButtonActive] = ImVec4(0.06f, 0.53f, 0.98f, 1.00f);  }  // 渲染现代化按钮  bool ClientUI::renderModernButton(const char \*label, ImVec2 size, bool primary)  {  if (primary)  {  ImGui::PushStyleColor(ImGuiCol\_Button, this->primaryColor);  ImGui::PushStyleColor(ImGuiCol\_ButtonHovered, ImVec4(this->primaryColor.x \* 1.2f, this->primaryColor.y \* 1.2f, this->primaryColor.z \* 1.2f, this->primaryColor.w));  ImGui::PushStyleColor(ImGuiCol\_ButtonActive, ImVec4(this->primaryColor.x \* 0.8f, this->primaryColor.y \* 0.8f, this->primaryColor.z \* 0.8f, this->primaryColor.w));  }  bool pressed = ImGui::Button(label, size);  if (primary)  {  ImGui::PopStyleColor(3);  }  return pressed;  }  // 渲染卡片  void ClientUI::renderCard(const char \*title, const char \*content, bool collapsible)  {  ImGui::PushStyleVar(ImGuiStyleVar\_ChildRounding, 8.0f);  ImGui::PushStyleColor(ImGuiCol\_ChildBg, ImVec4(0.16f, 0.17f, 0.18f, 1.00f));  if (collapsible)  {  if (ImGui::CollapsingHeader(title, ImGuiTreeNodeFlags\_DefaultOpen))  {  ImGui::BeginChild(title, ImVec2(0, 0), true);  ImGui::Text("%s", content);  ImGui::EndChild();  }  }  else  {  ImGui::Text("%s", title);  ImGui::Separator();  ImGui::Spacing();  ImGui::BeginChild(title, ImVec2(0, 0), true);  ImGui::Text("%s", content);  ImGui::EndChild();  }  ImGui::PopStyleColor();  ImGui::PopStyleVar();  }  // 渲染现代化搜索框  void ClientUI::renderSearchBox()  {  ImGui::PushStyleVar(ImGuiStyleVar\_FrameRounding, 20.0f);  ImGui::PushStyleVar(ImGuiStyleVar\_FramePadding, ImVec2(15.0f, 10.0f));  ImGui::PushStyleColor(ImGuiCol\_FrameBg, ImVec4(0.2f, 0.21f, 0.22f, 1.0f));  ImGui::PushStyleColor(ImGuiCol\_FrameBgHovered, ImVec4(0.25f, 0.26f, 0.27f, 1.0f));  ImGui::PushStyleColor(ImGuiCol\_Text, ImVec4(1.0f, 1.0f, 1.0f, 1.0f));  // 搜索图标和输入框  ImGui::SetNextItemWidth(-150.0f); // 留出按钮空间  bool searchTextChanged = ImGui::InputTextWithHint("##search", "搜索商品名称、描述或类别...", searchBuffer, IM\_ARRAYSIZE(searchBuffer));  ImGui::PopStyleColor(3);  ImGui::PopStyleVar(2);  // 搜索按钮  ImGui::SameLine();  ImGui::PushStyleVar(ImGuiStyleVar\_FrameRounding, 15.0f);  if (ImGui::Button("搜索", ImVec2(70, 40)) || searchTextChanged)  {  if (searchTextChanged || strcmp(searchBuffer, lastSearchBuffer) != 0)  {  strcpy\_s(lastSearchBuffer, sizeof(lastSearchBuffer), searchBuffer);  if (strlen(searchBuffer) == 0)  {  searchResults.clear();  }  else  {  performSearch();  }  }  }  // 清除按钮  ImGui::SameLine();  if (ImGui::Button("清除", ImVec2(70, 40)))  {  searchResults.clear();  memset(searchBuffer, 0, sizeof(searchBuffer));  memset(lastSearchBuffer, 0, sizeof(lastSearchBuffer));  refreshProducts();  }  ImGui::PopStyleVar();  }  // 渲染现代化产品卡片  void ClientUI::renderProductCard(const Protocol::ProductData &product)  {  ImGui::PushID(product.id.c\_str());  // 卡片容器  ImGui::PushStyleVar(ImGuiStyleVar\_ChildRounding, 12.0f);  ImGui::PushStyleVar(ImGuiStyleVar\_ChildBorderSize, 1.0f);  ImGui::PushStyleColor(ImGuiCol\_ChildBg, ImVec4(0.15f, 0.16f, 0.17f, 1.0f));  ImGui::PushStyleColor(ImGuiCol\_Border, ImVec4(0.3f, 0.3f, 0.3f, 0.5f));  float cardWidth = 280.0f;  float cardHeight = 200.0f;  if (ImGui::BeginChild("ProductCard", ImVec2(cardWidth, cardHeight), true))  {  ImGui::PushStyleVar(ImGuiStyleVar\_ItemSpacing, ImVec2(8.0f, 6.0f));  // 产品名称 - 使用较大字体  ImGui::PushFont(ImGui::GetIO().Fonts->Fonts[0]); // 使用默认字体  ImGui::PushStyleColor(ImGuiCol\_Text, this->primaryColor);  ImGui::TextWrapped("%s", product.name.c\_str());  ImGui::PopStyleColor();  ImGui::PopFont();  ImGui::Spacing();  // 价格信息  ImGui::PushStyleColor(ImGuiCol\_Text, ImVec4(0.4f, 0.8f, 0.4f, 1.0f));  if (product.discountRate > 0.0f)  {  float discountedPrice = product.originalPrice \* (1.0f - product.discountRate);  ImGui::Text("💰 %.2f元", discountedPrice);  ImGui::SameLine();  ImGui::PushStyleColor(ImGuiCol\_Text, ImVec4(0.6f, 0.6f, 0.6f, 1.0f));  ImGui::Text("(原价: %.2f元)", product.originalPrice);  ImGui::PopStyleColor();  }  else  {  ImGui::Text("💰 %.2f元", product.originalPrice);  }  ImGui::PopStyleColor();  // 库存信息  ImGui::PushStyleColor(ImGuiCol\_Text, product.quantity > 10 ? ImVec4(0.4f, 0.8f, 0.4f, 1.0f) : ImVec4(0.8f, 0.6f, 0.4f, 1.0f));  ImGui::Text("📦 库存: %d", product.quantity);  ImGui::PopStyleColor();  // 商家信息  ImGui::PushStyleColor(ImGuiCol\_Text, ImVec4(0.7f, 0.7f, 0.7f, 1.0f));  ImGui::Text("🏪 %s", product.sellerUsername.c\_str());  ImGui::PopStyleColor();  // 产品描述  if (!product.description.empty())  {  ImGui::Spacing();  ImGui::PushStyleColor(ImGuiCol\_Text, ImVec4(0.8f, 0.8f, 0.8f, 1.0f));  ImGui::TextWrapped("%s", product.description.substr(0, 50).c\_str());  if (product.description.length() > 50)  {  ImGui::SameLine();  ImGui::Text("...");  }  ImGui::PopStyleColor();  }  ImGui::Spacing();  // 操作按钮  if (isLoggedIn && userType == Protocol::UserType::CUSTOMER && product.quantity > 0)  {  ImGui::PushStyleVar(ImGuiStyleVar\_FrameRounding, 8.0f);  // 加入购物车按钮  ImGui::PushStyleColor(ImGuiCol\_Button, ImVec4(0.2f, 0.6f, 1.0f, 0.8f));  ImGui::PushStyleColor(ImGuiCol\_ButtonHovered, ImVec4(0.3f, 0.7f, 1.0f, 1.0f));  if (ImGui::Button("🛒 加入购物车", ImVec2(120, 30)))  {  addProductToCart(product);  }  ImGui::PopStyleColor(2);  ImGui::SameLine();  // 立即购买按钮  ImGui::PushStyleColor(ImGuiCol\_Button, ImVec4(1.0f, 0.4f, 0.4f, 0.8f));  ImGui::PushStyleColor(ImGuiCol\_ButtonHovered, ImVec4(1.0f, 0.5f, 0.5f, 1.0f));  if (ImGui::Button("💳 立即购买", ImVec2(120, 30)))  {  productToPurchase = product;  quantityToPurchase = 1;  showDirectPurchaseConfirmDialog = true;  }  ImGui::PopStyleColor(2);  ImGui::PopStyleVar();  }  else if (product.quantity == 0)  {  ImGui::PushStyleColor(ImGuiCol\_Text, ImVec4(0.8f, 0.4f, 0.4f, 1.0f));  ImGui::Text("❌ 暂时缺货");  ImGui::PopStyleColor();  }  ImGui::PopStyleVar(); // ItemSpacing  }  ImGui::EndChild();  ImGui::PopStyleColor(2);  ImGui::PopStyleVar(2);  ImGui::PopID();  }  // 渲染现代化状态栏  void ClientUI::renderStatusBar()  {  ImGui::PushStyleVar(ImGuiStyleVar\_WindowPadding, ImVec2(20.0f, 10.0f));  if (ImGui::BeginChild("StatusBar", ImVec2(0, 40), true))  {  if (isLoggedIn)  {  ImGui::PushStyleColor(ImGuiCol\_Text, ImVec4(0.4f, 0.8f, 0.4f, 1.0f));  ImGui::Text("当前用户：%s", currentUser.username.c\_str());  ImGui::PopStyleColor();  if (userType == Protocol::UserType::CUSTOMER)  {  ImGui::SameLine();  ImGui::PushStyleColor(ImGuiCol\_Text, ImVec4(1.0f, 0.8f, 0.2f, 1.0f));  ImGui::Text("余额: %.2f元", currentUser.balance);  ImGui::PopStyleColor();  }  }  else  {  ImGui::PushStyleColor(ImGuiCol\_Text, ImVec4(0.8f, 0.6f, 0.4f, 1.0f));  ImGui::Text("未登录");  ImGui::PopStyleColor();  }  // 显示状态消息  if (!statusMessage.empty())  {  ImGui::SameLine();  ImGui::PushStyleColor(ImGuiCol\_Text, ImVec4(0.4f, 0.8f, 0.4f, 1.0f));  ImGui::Text("%s", statusMessage.c\_str());  ImGui::PopStyleColor();  }  if (showErrorMessage && !errorMessage.empty())  {  ImGui::SameLine();  ImGui::PushStyleColor(ImGuiCol\_Text, ImVec4(0.8f, 0.4f, 0.4f, 1.0f));  ImGui::Text("%s", errorMessage.c\_str());  ImGui::PopStyleColor();  }  }  ImGui::EndChild();  ImGui::PopStyleVar();  }  // 渲染购买确认对话框  void ClientUI::renderPurchaseConfirmDialog()  {  if (!showPurchaseConfirmDialog)  return;  ImGui::OpenPopup("确认购买");  if (ImGui::BeginPopupModal("确认购买", nullptr, ImGuiWindowFlags\_AlwaysAutoResize))  {  // 显示详细的商品信息  ImGui::Text("订单详情");  ImGui::Separator();  ImGui::Text("商品名称：%s", productToPurchase.name.c\_str());  ImGui::Text("商品描述：%s", productToPurchase.description.c\_str());  ImGui::Text("商品类型：%s", productToPurchase.type.c\_str());  ImGui::Text("商 家：%s", productToPurchase.sellerUsername.c\_str());  ImGui::Text("单 价：CNY %.2f", productToPurchase.price);  ImGui::Text("数 量：%d", quantityToPurchase);  ImGui::Separator();  double totalAmount = productToPurchase.price \* quantityToPurchase;  ImGui::Text("总金额：CNY %.2f", totalAmount);  ImGui::Text("当前余额：CNY %.2f", currentUser.balance);  ImGui::Text("支付后余额：CNY %.2f", currentUser.balance - totalAmount);  ImGui::Separator();  if (ImGui::Button("确认购买", ImVec2(90, 0)))  {  purchaseProduct(productToPurchase.name, quantityToPurchase);  showPurchaseConfirmDialog = false;  ImGui::CloseCurrentPopup();  }  ImGui::SameLine();  if (ImGui::Button("取消", ImVec2(80, 0)))  {  showPurchaseConfirmDialog = false;  ImGui::CloseCurrentPopup();  }  ImGui::EndPopup();  }  }  // 渲染直接购买确认对话框  void ClientUI::renderDirectPurchaseConfirmDialog()  {  if (!showDirectPurchaseConfirmDialog)  return;  ImGui::OpenPopup("直接购买确认");  if (ImGui::BeginPopupModal("直接购买确认", nullptr, ImGuiWindowFlags\_AlwaysAutoResize))  {  // 显示详细的商品信息  ImGui::Text("直接购买订单详情");  ImGui::Separator();  ImGui::Text("商品名称：%s", productToPurchase.name.c\_str());  ImGui::Text("商品描述：%s", productToPurchase.description.c\_str());  ImGui::Text("商品类型：%s", productToPurchase.type.c\_str());  ImGui::Text("商 家：%s", productToPurchase.sellerUsername.c\_str());  ImGui::Text("库存数量：%d", productToPurchase.quantity);  ImGui::Text("单 价：CNY %.2f", productToPurchase.price);  ImGui::Text("购买数量：%d", quantityToPurchase);  ImGui::Separator();  double totalAmount = productToPurchase.price \* quantityToPurchase;  ImGui::Text("总金额：CNY %.2f", totalAmount);  ImGui::Text("当前余额：CNY %.2f", currentUser.balance);  if (currentUser.balance >= totalAmount)  {  ImGui::TextColored(ImVec4(0.0f, 0.8f, 0.0f, 1.0f), "支付后余额：CNY %.2f", currentUser.balance - totalAmount);  }  else  {  ImGui::TextColored(ImVec4(0.8f, 0.0f, 0.0f, 1.0f), "余额不足！还需：CNY %.2f", totalAmount - currentUser.balance);  }  ImGui::Separator();  bool canPurchase = (currentUser.balance >= totalAmount) && (productToPurchase.quantity >= quantityToPurchase);  if (!canPurchase)  {  ImGui::PushStyleColor(ImGuiCol\_Button, ImVec4(0.5f, 0.5f, 0.5f, 1.0f));  ImGui::PushStyleColor(ImGuiCol\_ButtonHovered, ImVec4(0.5f, 0.5f, 0.5f, 1.0f));  ImGui::PushStyleColor(ImGuiCol\_ButtonActive, ImVec4(0.5f, 0.5f, 0.5f, 1.0f));  }  if (ImGui::Button("确认购买", ImVec2(90, 0)) && canPurchase)  {  purchaseProduct(productToPurchase.name, quantityToPurchase);  showDirectPurchaseConfirmDialog = false;  ImGui::CloseCurrentPopup();  }  if (!canPurchase)  {  ImGui::PopStyleColor(3);  }  ImGui::SameLine();  if (ImGui::Button("取消", ImVec2(80, 0)))  {  // 取消购买时需要解锁库存  if (inventoryLocked)  {  if (networkClient->unlockInventory(productToPurchase.id, quantityToPurchase))  {  inventoryLocked = false;  std::cout << "取消购买，库存已解锁" << std::endl;  }  else  {  setError("警告：取消购买时库存解锁失败！请联系客服处理");  }  }  showDirectPurchaseConfirmDialog = false;  ImGui::CloseCurrentPopup();  }  ImGui::EndPopup();  }  }  // 渲染购物车结账确认对话框  void ClientUI::renderCartCheckoutConfirmDialog()  {  if (!showCartCheckoutConfirmDialog)  return;  ImGui::OpenPopup("购物车结账确认");  if (ImGui::BeginPopupModal("购物车结账确认", nullptr, ImGuiWindowFlags\_AlwaysAutoResize | ImGuiWindowFlags\_NoResize))  {  ImGui::SetWindowSize(ImVec2(500, 0), ImGuiCond\_Always);  ImGui::Text("购物车结账订单详情");  ImGui::Separator();  // 计算总金额和检查库存状态  double totalAmount = 0.0;  bool hasStockIssues = false;  // 显示商品列表  if (ImGui::BeginTable("CheckoutTable", 5, ImGuiTableFlags\_Borders | ImGuiTableFlags\_RowBg | ImGuiTableFlags\_ScrollY, ImVec2(0, 150)))  {  ImGui::TableSetupColumn("商品名称", ImGuiTableColumnFlags\_WidthFixed, 120.0f);  ImGui::TableSetupColumn("数量", ImGuiTableColumnFlags\_WidthFixed, 50.0f);  ImGui::TableSetupColumn("单价", ImGuiTableColumnFlags\_WidthFixed, 80.0f);  ImGui::TableSetupColumn("小计", ImGuiTableColumnFlags\_WidthFixed, 80.0f);  ImGui::TableSetupColumn("商家", ImGuiTableColumnFlags\_WidthFixed, 100.0f);  ImGui::TableHeadersRow();  for (const auto &item : cartItems)  {  ImGui::TableNextRow();  // 检查库存状态  bool stockAvailable = true;  Protocol::ProductData \*product = nullptr;  for (auto &p : allProducts)  {  if (p.id == item.productId)  {  product = &p;  stockAvailable = (p.quantity >= item.quantity);  break;  }  }  if (!stockAvailable)  {  hasStockIssues = true;  ImGui::TableSetBgColor(ImGuiTableBgTarget\_RowBg0, ImGui::GetColorU32(ImVec4(1.0f, 0.8f, 0.8f, 0.3f)));  }  ImGui::TableSetColumnIndex(0);  if (!stockAvailable)  {  ImGui::TextColored(ImVec4(0.8f, 0.0f, 0.0f, 1.0f), "%s", item.productName.c\_str());  }  else  {  ImGui::Text("%s", item.productName.c\_str());  }  ImGui::TableSetColumnIndex(1);  if (!stockAvailable && product)  {  ImGui::TextColored(ImVec4(0.8f, 0.0f, 0.0f, 1.0f), "%d (库存:%d)", item.quantity, product->quantity);  }  else  {  ImGui::Text("%d", item.quantity);  }  ImGui::TableSetColumnIndex(2);  ImGui::Text("CNY%.2f", item.priceAtAddition);  ImGui::TableSetColumnIndex(3);  double subtotal = item.priceAtAddition \* item.quantity;  ImGui::Text("CNY%.2f", subtotal);  totalAmount += subtotal;  ImGui::TableSetColumnIndex(4);  ImGui::Text("%s", item.sellerUsername.c\_str());  }  ImGui::EndTable();  }  ImGui::Separator();  // 显示总计和余额信息  ImGui::Text("订单总计：CNY %.2f", totalAmount);  ImGui::Text("当前余额：CNY %.2f", currentUser.balance);  if (currentUser.balance >= totalAmount)  {  ImGui::TextColored(ImVec4(0.0f, 0.8f, 0.0f, 1.0f), "支付后余额：CNY %.2f", currentUser.balance - totalAmount);  }  else  {  ImGui::TextColored(ImVec4(0.8f, 0.0f, 0.0f, 1.0f), "余额不足！还需：CNY %.2f", totalAmount - currentUser.balance);  }  // 显示库存警告  if (hasStockIssues)  {  ImGui::Separator();  ImGui::TextColored(ImVec4(0.8f, 0.0f, 0.0f, 1.0f), "⚠ 部分商品库存不足，请调整购买数量");  }  ImGui::Separator();  bool canCheckout = (currentUser.balance >= totalAmount) && !hasStockIssues;  if (!canCheckout)  {  ImGui::PushStyleColor(ImGuiCol\_Button, ImVec4(0.5f, 0.5f, 0.5f, 1.0f));  ImGui::PushStyleColor(ImGuiCol\_ButtonHovered, ImVec4(0.5f, 0.5f, 0.5f, 1.0f));  ImGui::PushStyleColor(ImGuiCol\_ButtonActive, ImVec4(0.5f, 0.5f, 0.5f, 1.0f));  }  if (ImGui::Button("确认结账", ImVec2(100, 0)) && canCheckout)  {  checkoutWithInventoryLock();  showCartCheckoutConfirmDialog = false;  ImGui::CloseCurrentPopup();  }  if (!canCheckout)  {  ImGui::PopStyleColor(3);  }  ImGui::SameLine();  if (ImGui::Button("取消", ImVec2(80, 0)))  {  showCartCheckoutConfirmDialog = false;  ImGui::CloseCurrentPopup();  }  ImGui::EndPopup();  }  }  // 渲染添加到购物车成功弹窗  void ClientUI::renderAddToCartSuccessPopup()  {  if (!showAddToCartSuccessPopup)  return;  ImGui::OpenPopup("添加成功");  if (ImGui::BeginPopupModal("添加成功", nullptr, ImGuiWindowFlags\_AlwaysAutoResize))  {  ImGui::Text("%s", addToCartSuccessMessage.c\_str());  ImGui::Separator();  if (ImGui::Button("确定", ImVec2(80, 0)))  {  showAddToCartSuccessPopup = false;  ImGui::CloseCurrentPopup();  }  ImGui::EndPopup();  }  }  // 渲染直接购买成功弹窗  void ClientUI::renderDirectPurchaseSuccessPopup()  {  if (!showDirectPurchaseSuccessPopup)  return;  ImGui::OpenPopup("购买成功");  if (ImGui::BeginPopupModal("购买成功", nullptr, ImGuiWindowFlags\_AlwaysAutoResize))  {  ImGui::Text("%s", directPurchaseSuccessMessage.c\_str());  ImGui::Separator();  if (ImGui::Button("确定", ImVec2(80, 0)))  {  showDirectPurchaseSuccessPopup = false;  ImGui::CloseCurrentPopup();  }  ImGui::EndPopup();  }  }  // 渲染密码修改对话框  void ClientUI::renderPasswordChangeDialog()  {  if (!showPasswordChangeDialog)  return;  static char oldPassword[128] = {0};  static char newPassword[128] = {0};  static char confirmPassword[128] = {0};  ImGui::OpenPopup("修改密码");  if (ImGui::BeginPopupModal("修改密码", nullptr, ImGuiWindowFlags\_AlwaysAutoResize))  {  ImGui::Text("用户：%s", currentUser.username.c\_str());  ImGui::Separator();  ImGui::InputText("原密码", oldPassword, sizeof(oldPassword), ImGuiInputTextFlags\_Password);  ImGui::InputText("新密码", newPassword, sizeof(newPassword), ImGuiInputTextFlags\_Password);  ImGui::InputText("确认新密码", confirmPassword, sizeof(confirmPassword), ImGuiInputTextFlags\_Password);  ImGui::Separator();  if (ImGui::Button("确认修改", ImVec2(100, 0)))  {  if (strcmp(newPassword, confirmPassword) == 0)  {  // 这里应该调用网络API修改密码  showPasswordChangeSuccessDialog = true;  showPasswordChangeDialog = false;  memset(oldPassword, 0, sizeof(oldPassword));  memset(newPassword, 0, sizeof(newPassword));  memset(confirmPassword, 0, sizeof(confirmPassword));  ImGui::CloseCurrentPopup();  }  else  {  showPasswordChangeFailDialog = true;  }  }  ImGui::SameLine();  if (ImGui::Button("取消", ImVec2(80, 0)))  {  showPasswordChangeDialog = false;  memset(oldPassword, 0, sizeof(oldPassword));  memset(newPassword, 0, sizeof(newPassword));  memset(confirmPassword, 0, sizeof(confirmPassword));  ImGui::CloseCurrentPopup();  }  ImGui::EndPopup();  }  }  // 渲染密码修改成功对话框  void ClientUI::renderPasswordChangeSuccessDialog()  {  if (!showPasswordChangeSuccessDialog)  return;  ImGui::OpenPopup("修改成功");  if (ImGui::BeginPopupModal("修改成功", nullptr, ImGuiWindowFlags\_AlwaysAutoResize))  {  ImGui::Text("密码修改成功！");  ImGui::Separator();  if (ImGui::Button("确定", ImVec2(80, 0)))  {  showPasswordChangeSuccessDialog = false;  ImGui::CloseCurrentPopup();  }  ImGui::EndPopup();  }  }  // 渲染密码修改失败对话框  void ClientUI::renderPasswordChangeFailDialog()  {  if (!showPasswordChangeFailDialog)  return;  ImGui::OpenPopup("修改失败");  if (ImGui::BeginPopupModal("修改失败", nullptr, ImGuiWindowFlags\_AlwaysAutoResize))  {  ImGui::Text("密码修改失败！请检查输入。");  ImGui::Separator();  if (ImGui::Button("确定", ImVec2(80, 0)))  {  showPasswordChangeFailDialog = false;  ImGui::CloseCurrentPopup();  }  ImGui::EndPopup();  }  }  // 渲染充值对话框  void ClientUI::renderRechargeDialog()  {  if (!showRechargeDialog)  return;  ImGui::OpenPopup("账户充值");  if (ImGui::BeginPopupModal("账户充值", nullptr, ImGuiWindowFlags\_AlwaysAutoResize))  {  ImGui::Text("当前余额：%.2f 元", currentUser.balance);  ImGui::Separator();  ImGui::InputFloat("充值金额", &rechargeAmount, 1.0f, 10.0f, "%.2f");  if (rechargeAmount < 0)  rechargeAmount = 0;  ImGui::Separator();  if (ImGui::Button("确认充值", ImVec2(100, 0)))  {  if (rechargeAmount > 0)  {  // 这里应该调用网络API进行充值  currentUser.balance += rechargeAmount;  showRechargeSuccessDialog = true;  showRechargeDialog = false;  rechargeAmount = 0.0f;  ImGui::CloseCurrentPopup();  }  }  ImGui::SameLine();  if (ImGui::Button("取消", ImVec2(80, 0)))  {  showRechargeDialog = false;  rechargeAmount = 0.0f;  ImGui::CloseCurrentPopup();  }  ImGui::EndPopup();  }  }  // 渲染充值成功对话框  void ClientUI::renderRechargeSuccessDialog()  {  if (!showRechargeSuccessDialog)  return;  ImGui::OpenPopup("充值成功");  if (ImGui::BeginPopupModal("充值成功", nullptr, ImGuiWindowFlags\_AlwaysAutoResize))  {  ImGui::Text("充值成功！");  ImGui::Text("当前余额：%.2f 元", currentUser.balance);  ImGui::Separator();  if (ImGui::Button("确定", ImVec2(80, 0)))  {  showRechargeSuccessDialog = false;  ImGui::CloseCurrentPopup();  }  ImGui::EndPopup();  }  }  // 渲染充值失败对话框  void ClientUI::renderRechargeFailDialog()  {  if (!showRechargeFailDialog)  return;  ImGui::OpenPopup("充值失败");  if (ImGui::BeginPopupModal("充值失败", nullptr, ImGuiWindowFlags\_AlwaysAutoResize))  {  ImGui::Text("充值失败！%s", rechargeMessage.c\_str());  ImGui::Separator();  if (ImGui::Button("确定", ImVec2(80, 0)))  {  showRechargeFailDialog = false;  ImGui::CloseCurrentPopup();  }  ImGui::EndPopup();  }  }  // 窗口响应式功能实现  // 更新窗口尺寸  void ClientUI::updateWindowSize()  {  int newWidth, newHeight;  glfwGetWindowSize(window, &newWidth, &newHeight);  if (newWidth != width || newHeight != height)  {  width = newWidth;  height = newHeight;  updateUIScale();  }  }  // 更新UI缩放比例  void ClientUI::updateUIScale()  {  // 计算基于宽度和高度的缩放比例  float scaleX = static\_cast<float>(width) / static\_cast<float>(originalWidth);  float scaleY = static\_cast<float>(height) / static\_cast<float>(originalHeight);  // 使用较小的缩放比例来确保UI元素不会超出窗口  uiScale = std::min(scaleX, scaleY);  // 限制缩放比例范围，避免过小或过大  uiScale = std::max(0.5f, std::min(uiScale, 3.0f));  // 更新ImGui样式缩放  ImGuiStyle &style = ImGui::GetStyle();  ImGuiIO &io = ImGui::GetIO();  // 缩放字体  io.FontGlobalScale = uiScale;  // 缩放ImGui样式元素  style.WindowPadding = ImVec2(15.0f \* uiScale, 15.0f \* uiScale);  style.FramePadding = ImVec2(12.0f \* uiScale, 8.0f \* uiScale);  style.ItemSpacing = ImVec2(12.0f \* uiScale, 8.0f \* uiScale);  style.ItemInnerSpacing = ImVec2(8.0f \* uiScale, 6.0f \* uiScale);  style.IndentSpacing = 25.0f \* uiScale;  style.ScrollbarSize = 15.0f \* uiScale;  style.GrabMinSize = 10.0f \* uiScale;  // 缩放圆角  style.WindowRounding = 10.0f \* uiScale;  style.ChildRounding = 8.0f \* uiScale;  style.FrameRounding = 6.0f \* uiScale;  style.PopupRounding = 8.0f \* uiScale;  style.ScrollbarRounding = 6.0f \* uiScale;  style.GrabRounding = 6.0f \* uiScale;  style.TabRounding = 6.0f \* uiScale;  }  // 缩放尺寸向量  ImVec2 ClientUI::scaleSize(const ImVec2 &size)  {  return ImVec2(size.x \* uiScale, size.y \* uiScale);  }  // 缩放浮点值  float ClientUI::scaleFloat(float value)  {  return value \* uiScale;  }  // GLFW窗口大小回调函数  void ClientUI::framebufferSizeCallback(GLFWwindow \*window, int width, int height)  {  // 获取用户指针（ClientUI实例）  ClientUI \*clientUI = static\_cast<ClientUI \*>(glfwGetWindowUserPointer(window));  if (clientUI)  {  clientUI->updateWindowSize();  // 更新OpenGL视口  glViewport(0, 0, width, height);  }  } |