C++课程设计报告

目录

一、 基础题目

1. C++基础知识实验

```
// matrix.cpp
#include<iostream>
#include<cstring>
using std::cin;
using std::cout;
using std::endl;

// 保存矩阵的结构体
typedef struct {
    int** data;
    int n;
    int m;
}matrix;

// 创建一个矩阵
matrix* create_matrix(int n,int m);

// 从标准输入中读取一个矩阵
void matrix_read(matrix* mat);

// 向标准输出打印一个矩阵
void matrix_print(matrix* mat);

// 两个矩阵相加
matrix* matrix_add(matrix* a,matrix* b);

// 两个矩阵相减，返回堆上构造的一个矩阵
matrix* matrix_minus(matrix* a,matrix* b);
```

```
//释放一个矩阵
void matrix_free(matrix* a);

int main(){
    matrix *A1,*A2,*A3;

    // 预分配两个矩阵
    A1 = create_matrix(4,5);
    A2 = create_matrix(4,5);

    // 读取矩阵数据
    matrix_read(A1);
    matrix_read(A2);

    // 计算矩阵加法
    A3 = matrix_add(A1,A2);
    matrix_print(A3);

    // 计算矩阵减法
    A3 = matrix_minus(A1,A2);
    matrix_print(A3);


    matrix_free(A1);
    matrix_free(A2);
    matrix_free(A3);
}

matrix* create_matrix(int n,int m){
    matrix *mat = new matrix();
    mat->n = n;
    mat->m = m;
    mat->data = new int*[n];
    for(int i =0;i<n;i++){
        mat->data[i] = new int[m];
    }
    // memset(matrix,0,sizeof(int)*n*m);
```

```
    return mat;
}
void matrix_read(matrix *mat){
    int n = mat->n;int m=mat->m;
    for(int i =0;i<n;i++)
        for(int j=0;j<m;j++)
            cin>>mat->data[i][j];
}
void matrix_print(matrix* mat){
    int n = mat->n;int m=mat->m;
    for(int i =0;i<n;i++){
        for(int j=0;j<m;j++)
            cout<<mat->data[i][j]<<"  ";
        cout<<endl;
    }
}
matrix* matrix_add(matrix* a,matrix* b){
    // 计算之前要先检查 shape 是否一致
    if (!(a->n==b->n) or !(a->m==b->m)){
        throw std::runtime_error("non-equal shape");
    }

    int n= a->n;int m=a->m;
    matrix* result = create_matrix(n,m);
    for(int i=0;i<n;i++)
        for(int j=0;j<m;j++){
            result->data[i][j] = a->data[i][j] + b->data[i][j];
        }
    return result;
}

matrix* matrix_minus(matrix* a,matrix* b){
    if (!(a->n==b->n) or !(a->m==b->m)){
        throw std::runtime_error("non-equal shape");
    }

    int n= a->n;int m=a->m;
    matrix* result = create_matrix(n,m);
```

```
    for(int i=0;i<n;i++)
        for(int j=0;j<m;j++)
            result->data[i][j] = a->data[i][j] - b->data[i][j];
    return result;

}
void matrix_free(matrix* a){
    int n= a->n;int m=a->m;
    for(int i=0;i<n;i++){
        delete [] a->data[i];
    }
    delete [] a->data;
}
```

这个实验中，为了传参方便用了结构体，但是结构体中没有带函数，相当于还是使用的 c 的结构体。这里设计函数的时候是采用 c 中常用的方式：传入指针，分配资源/释放资源。这样做的话可能会忘记释放资源导致内存泄露。

2.类与对象实验

1. 编写 C++程序完成“圆形”

```
// 2_1.h
#ifndef POINT_CIRCLE_H
#define POINT_CIRCLE_H
#include<cmath>
#include<cstdio>
#include<iostream>

class Point{
    public:

        Point():x(0),y(0){
            printf("Point(%.2lf,%.2lf) created\n",x,y);
        }
        Point(double x,double y):x(x),y(y){
            printf("Point(%.2lf,%.2lf) created\n",x,y);
        }
        ~Point(){
```

```
            printf("Point(%.2lf,%.2lf) destoryed\n",x,y);
        }

        //计算距离
        double distance(Point &p){
            return sqrt((x-p.x)*(x-p.x)+(y-p.y)*(y-p.y));
        }
    private:
        double x,y;
};


class Circle{
    public:
        Circle(double x,double y,double rad):x(x),y(y),rad(rad){
            printf("Cicle(center=(%.2lf,%.2lf),radius=%.2lf) created\n",x,y,rad);
        }
        ~Circle(){
            printf("Cicle(center=(%.2lf,%.2lf),radius=%.2lf) destoryed\n",x,y,rad);
        }
        bool is_intersect(Circle &c){
            Point p1(x,y),p2(c.x,c.y);
            double distance_of_radius = p1.distance(p2);
            // 比较圆心距离和半径距离
            return bool(distance_of_radius < rad+c.rad && distance_of_radius>fabs(rad-c.rad));
        }
    private:
        // rad 是半径
        double x,y,rad;
};

#endif
```

这个实验比较简单。主要是判断的时候需要注意，圆相交时，既要满足圆心距小于半径之和，还要满足不是内含（小圆在大圆内部），换成数学表达就是大半径减去小半径要小于圆心距。事实上，这个点我最初写的时候也没注意到，后来做了几个简单测试(圆心(0,0),半径 2;圆心(1,1),半径 0.5)，发现居然出现了错误，这次纠正过来。因此，测试还是很重要的。

2. 编写 C++程序完成“矩阵”类

```
//2_2.h
class matrix{
    public:
        matrix(int rows,int lines):rows(rows),lines(lines){
            cout<<"普通构造"<<endl;
            data = new int*[rows];
            for(int i=0;i<rows;i++)
                data[i] = new int[lines];
        }

        // 拷贝构造
        matrix(const matrix &m){
            cout<<"拷贝构造"<<endl;
            rows = m.rows;lines = m.lines;
            data = new int*[rows];
            for(int i=0;i<rows;i++){
                data[i] = new int[lines];
                for(int j=0;j<lines;j++){
                    data[i][j] = m.data[i][j];
                }
            }
        }

        // 拷贝赋值
        matrix& operator=(const matrix& m){
            // std::swap(*this,m);
            cout<<"拷贝赋值";
            if(this==&m){
                cout<<" 自赋值"<<endl;
                return *this;
            }
            rows = m.rows;
            lines = m.lines;
            data = new int*[rows];
            for(int i=0;i<rows;i++){
                data[i] = new int[lines];
                for(int j=0;j<lines;j++){
                    data[i][j] = m.data[i][j];
```

```
            }
        }
        cout<<endl;
        return *this;
    }

    // 移动赋值
    matrix& operator=(matrix &&m){
        cout<<"移动赋值"<<endl;
        if(this==&m){
            return *this;
        }
        free();
        data = m.data;
        rows = m.rows;
        lines = m.lines;
        m.data =nullptr;
        return *this;
    }

    // 移动构造
    matrix(matrix &&m)noexcept:data(m.data),rows(m.rows),lines(m.lines){//移动构造
        cout<<"移动构造"<<endl;
        m.data = nullptr;
    }
    ~matrix(){free();}

    //检查矩阵的 shape
    bool check_shape(const matrix &mat)const{
        if (rows!=mat.rows && lines!=mat.lines)
            throw runtime_error("different shape");
            return false;
        return true;
    }

    // 计算加法
    matrix operator+(const matrix &m){
        matrix ret(lines,rows);
```

```
            check_shape(m);
            for(int i=0;i<lines;i++)
                for(int j=0;j<rows;j++){
                    ret.data[i][j] = data[i][j] + m.data[i][j];
                }
            return ret;
        }

        // 计算减法
        matrix operator-(const matrix &m){
            matrix ret(lines,rows);
            check_shape(m);
            for(int i=0;i<lines;i++)
                for(int j=0;j<rows;j++){
                    ret.data[i][j] = data[i][j] - m.data[i][j];
                }
            return ret;
        }
        const unsigned row()const{return this->rows;}
        const unsigned line()const{return this->lines;}
        friend istream& operator>>(istream& is,matrix &mat);
        friend ostream& operator<<(ostream& os,const matrix &mat);
    private:
        void free(){
            cout<<"析构对象"<<endl;
            if(data!=nullptr){
                for(int i=0;i<rows;i++)
                    delete [] data[i];
                delete []data;
            }
        }
        unsigned rows,lines;
        int ** data;
};

istream& operator>>(istream& is,matrix &mat);
ostream& operator<<(ostream& os,const matrix &mat);
#endif
```

这个实验的功能部分相当于把实验 1 封装成类。构造、析构、计算加法和减法都与之前类似，不再赘述；此处计算加法时直接返回了构造的新矩阵，由于是栈上分配的，因此这里不用担心释放的问题了（当然后面用 new 还是需要记得 delete）。

这里处理自赋值问题时，把拷贝构造，拷贝赋值，移动构造，移动赋值都重写了一遍，主要是在 c++ primer 上了解到了相关概念，想顺手实践一下。

拷贝构造在使用对象构造对象时被调用，例如实参拷贝到形参，返回值拷贝到未初始化的对象；拷贝赋值在给已经初始化的对象赋值的时候被调用；移动构造在右值初始化对象时被调用，例如用函数返回值初始化未初始化的对象；移动赋值在用右值给初始化对象赋值的时候调用，例如用函数返回值给已初始化对象赋值。移动构造、移动赋值不是必需的，若缺少这两个函数，编译器会调用响应的拷贝构造和拷贝赋值。

拷贝构造部分不需要考虑自赋值，因此和构造函数类似。拷贝赋值部分，这里首先判断了一下两个分配的内存地址是否相同；如果相同，说明是自赋值，那么直接跳过就行了。这样就可以解决自赋值问题了。

```
matrix& operator=(const matrix& m){
     if(this==&m){
          cout<<" 自赋值"<<endl;
          return *this;
     }
     // 否则，给当前对象赋值
     // ....略
}
```

对于右值，重写移动构造和移动拷贝可以稍微提高一些速度。

移动构造和赋值比较简单，移动赋值同样需要处理自赋值问题。这里采用的是和拷贝赋值同样的方案：直接判断地址。其他的解决方案可以先赋值，把左边的值拷贝到右边，然后把右边的值释放了，这样即使是自赋值也没事，当然这样比较费时间。在 c++ primer 上使用的是重载 swap()函数，然后调用自身的 swap 函数，这种方式更优雅。

```
// 移动构造
matrix& operator=(matrix &&m){
    cout<<"移动赋值"<<endl;
    if(this==&m){
        return *this;
    }
    free();
    data = m.data;
    rows = m.rows;
    lines = m.lines;
    m.data =nullptr; // 把右值的数据指针指向空
    return *this;
}
```

```
// 移动构造
matrix(matrix &&m)noexcept:data(m.data),rows(m.rows),lines(m.lines){//直接复制指针和值
    cout<<"移动构造"<<endl;
    m.data = nullptr; // 把右值的数据指针指向空
}
```

3. 继承与派生实验

```
// shape.h
class Shape{
    public:
        Shape(){}
        virtual ~Shape(){}//如果不加 virtual 子类不会被删掉
    //do nothing
        double square();
};
class Rect:public Shape{
    public:
        Rect(double a,double b):edgea(a),edgeb(b){
            cout<<"Rect("<<a<<","<<b<<") created"<<endl;
        }
        double square()const{
            return edgea*edgeb;
        }
        ~Rect(){
            cout<<"Rect("<<edgea<<","<<edgeb<<") destoryed"<<endl;
        }
    protected:
        double edgea,edgeb;//不区分长宽
};
class Circle:public Shape{
    public:
        Circle(double r):radius(r){
            cout<<"Circle ("<<radius<<") created"<<endl;
        }
        double square()const{
            return M_PI*radius*radius;
        }
```

```
        ~Circle(){
            cout<<"Circle("<<radius<<") destoryed"<<endl;
        }
    private:
        double radius;
};
class Square:public Rect{
    public:
        Square(int a):Rect(a,a){
            cout<<"Square ("<<edgea<<") created"<<endl;
        }
        double square()const{
            return edgea*edgea;
        }
        ~Square(){
            cout<<"Square ("<<edgea<<") destoryed"<<endl;
        }
};
```

测试的样例

```
// shape.cpp
#include"shape.h"

int main(){
    Circle circle(3);
    cout<<circle.square()<<endl;
    cout<<"\n";
    Rect rect(1,2);
    cout<<rect.square()<<endl;
    cout<<"\n";
    Square square(2);
    cout<<square.square()<<endl;
 }
```

输出内容

```
Circle (3) created
28.26
Rect(1,2) created
2
Rect(2,2) created
```

```
Square (2) created
4
Square (2) destoryed
Rect(2,2) destoryed
Rect(1,2) destoryed
Circle(3) destoryed
```

从调试信息中可以看到，构造时，父类先于子类构造；析构时，子类先于父类析构。因为父类可以看做是子类的一个成员，因此在构造时要先被构造。此外，如果用父类指针承接一个子类的对象，而父类没有设置虚析构，那么析构的时候不会调用子类的虚构函数。因此，一般父类的析构函数都要设置为虚函数。

4. I/O 流实验

```
// goods.h
class goods{
    public:
        goods(){
            static std::default_random_engine generator(time(NULL));
            static std::uniform_int_distribution<int> distribution(1,1000);
            _price = distribution(generator);
        }
        const unsigned price()const{return _price;}
    private:
        unsigned _price;
};
```

```
// main.cpp
int goods_value=0;


//捕获 ctrl+c
void func(int s){
    cout<<"\n 猜的数是: "<<goods_value<<endl;
    exit(0);
}


void guess(){
    int guess=0;
```

```
    goods g;
    if (signal(SIGINT,func)==SIG_ERR){
        cout<<"can't catch SIGINT";
    }
    goods_value = g.price();
    cout<<"输入一个数:";
    while(true){
        cin>>guess;
        if(!cin.good()){
            cout<<"错误"<<endl;
            cin.clear();
            cin.ignore(std::numeric_limits<int>::max(),'\n');
            continue;
        }
        if(guess<1 || guess > 1000){
            cout<<"超出范围"<<endl;
            continue;
        }
        if (g.price()==guess){
            cout<<"猜对了"<<endl;
            return;
        }else if(guess<g.price()){
            cout<<"太小了"<<endl;
        }else{
            cout<<"太大了"<<endl;
        }
    }
}
```

生成随机数时，我使用的是 c++11 的新特性<random>，这里生成的不是真随机数，而是和 rand 效果类似的伪随机数，但是可以方便地设定区间。

这个实验需要注意的是对异常值的处理，例如，如果用户输入了过大或者过小的数，或者用户输入了非数字的字符。在 guess 函数中，每读取一个数字时，都进行判断，看 cin.good()是否置位；如果没有置位，说明读入失败，要恢复流状态，cin.clear()用来清空标志，cin.ignore 跳过本行数据，直到遇到\n。

```
if(!cin.good()){
     cout<<"错误"<<endl;
     cin.clear();
     cin.ignore(std::numeric_limits<int>::max(),'\n');
     continue;
```

```
}
```

这里还加了一个 signal 来捕获 ctrl+c，可以在退出程序时查看需要猜的答案是多少。

5. 重载实验

1. 虚函数

下面第一个头文件是虚函数版本，第二个是纯虚函数版本

```
//shape.h 虚函数版本
class Shape{
    public:
        Shape(){}
        virtual ~Shape(){}//如果不加 virtual 子类不会被删掉
        virtual double square()const{return 0;};
};
class Rect:public Shape{
    public:
        Rect(double a,double b):edgea(a),edgeb(b){
            cout<<"Rect("<<a<<","<<b<<") created"<<endl;
        }
        double square()const override{
            return edgea*edgeb;
        }
        ~Rect(){
            cout<<"Rect("<<edgea<<","<<edgeb<<") destoryed"<<endl;
        }
    protected:
        double edgea,edgeb;//不区分长宽
};
class Circle:public Shape{
    public:
        Circle(double r):radius(r){
            cout<<"Circle ("<<radius<<") created"<<endl;
        }
        double square()const override{
            return M_PI*radius*radius;
        }
```

```
        ~Circle(){
            cout<<"Circle("<<radius<<") destoryed"<<endl;
        }
    private:
        double radius;
};
class Square:public Rect{
    public:
        Square(int a):Rect(a,a){
            cout<<"Square ("<<edgea<<") created"<<endl;
        }
        double square()const override{
            return edgea*edgea;
        }
        ~Square(){
            cout<<"Square ("<<edgea<<") destoryed"<<endl;
        }
};
```

```
//shape.h 虚函数版本
class Shape{
    public:
        Shape(){}
        virtual ~Shape(){}//如果不加 virtual 子类不会被删掉
    //do nothing
        virtual double square()const=0;
};
class Rect:public Shape{
    public:
        Rect(double a,double b):edgea(a),edgeb(b){
            cout<<"Rect("<<a<<","<<b<<") created"<<endl;
        }
        double square()const override{
            return edgea*edgeb;
        }
        ~Rect(){
            cout<<"Rect("<<edgea<<","<<edgeb<<") destoryed"<<endl;
```

```
        }
    protected:
        double edgea,edgeb;//不区分长宽
};
class Circle:public Shape{
    public:
        Circle(double r):radius(r){
            cout<<"Circle ("<<radius<<") created"<<endl;
        }
        double square()const override{
            return M_PI*radius*radius;
        }
        ~Circle(){
            cout<<"Circle("<<radius<<") destoryed"<<endl;
        }
    private:
        double radius;
};
class Square:public Rect{
    public:
        Square(int a):Rect(a,a){
            cout<<"Square ("<<edgea<<") created"<<endl;
        }
        double square()const override{
            return edgea*edgea;
        }
        ~Square(){
            cout<<"Square ("<<edgea<<") destoryed"<<endl;
        }
};
```

有虚函数的类，仍然可以被初始化；子类重写父类虚函数时，最好用 override 显式提醒编译器此处是重写。当父类指针指向父类对象，调用时会调用父类虚函数；如果父类指针指向子类对象，调用时会调用子类虚函数。

对于有纯虚函数的类，即抽象类，不能被初始化，一般来说，纯虚函数不应该提供实现，但是也可以提供实现，或者常用模板模式进行设计。例如:

```
class A{
    virtual int fun()=0;
```

```
    int func(){
        //方式二
        //....
        fun();
        //....
    }
};
class B:public A{
    int fun()override{}
};


int A::fun(){
    //方式一
    //do something
}
```

综合题目

环境

本项目使用 cmake 构建，生成出静态库，然后再构建 qt 项目，链接静态库，只调用静态库中的接口。这样的好处是显示与逻辑分离，同时理论上来说，只用了原生的 c++和一部分小的第三方库；比较麻烦的地方是链接，项目在 windows 下始终无法链接成功（可能是依赖顺序，或者缺少库），折腾了一周都没找到解决方案，于是我将其完全转到了 linux 下。
系统：ubuntu 20.04(WSL2, Windows Subsystem for Linux 2)
编译器：g++ 8.0 以上
链接：cmake + gnu ld

文档和注释

Doxygen 可以根据注释生成文档，因此我在源码下的 doc 文件夹生成了文档:
第一部分和第三部分的头文件都有注释和简单的说明（**第二部分因为大部分和第一部分重复，所以没加上注释**）。打开**/doc/index.html,**选择类列表，可以查看文档。

电子商城

目录

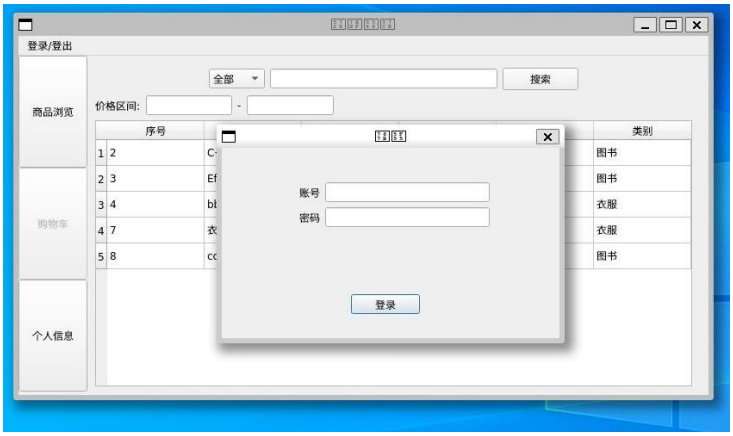
二、账户管理子系统和商品管理子系统

1. 运行截图

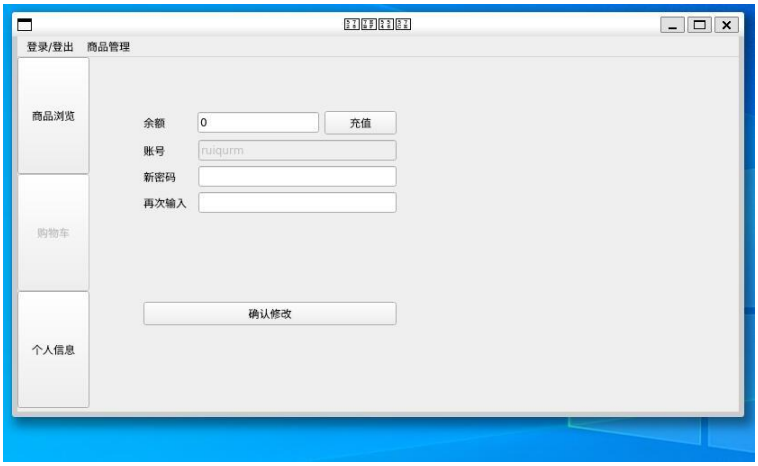
总览

注册

登录

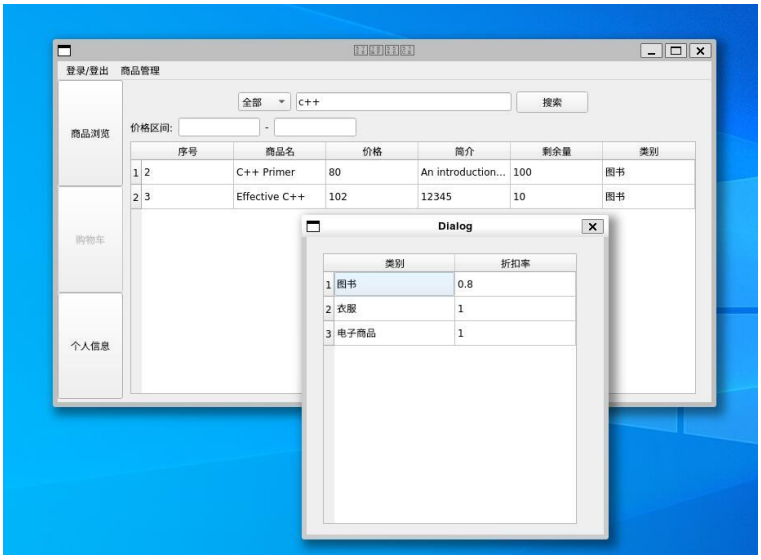


个人信息

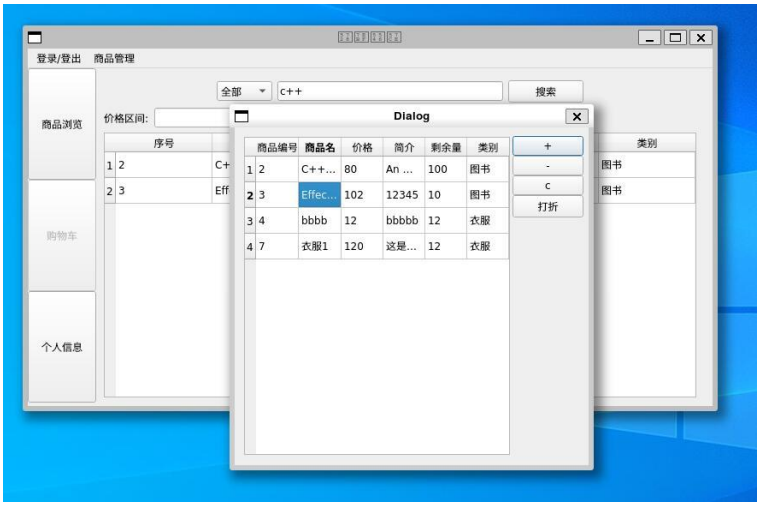


购买界面

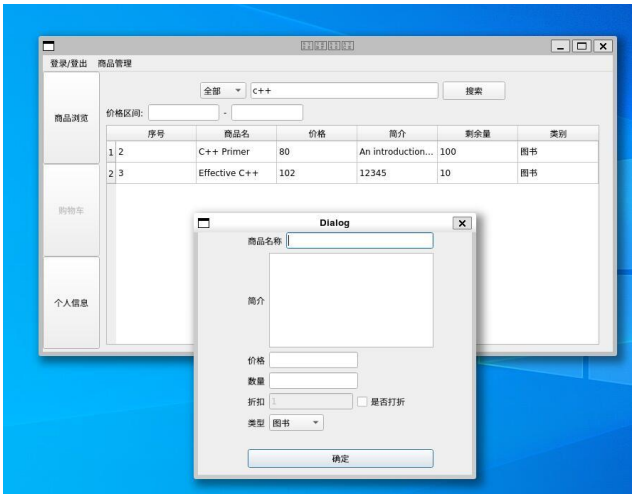
折扣管理界面



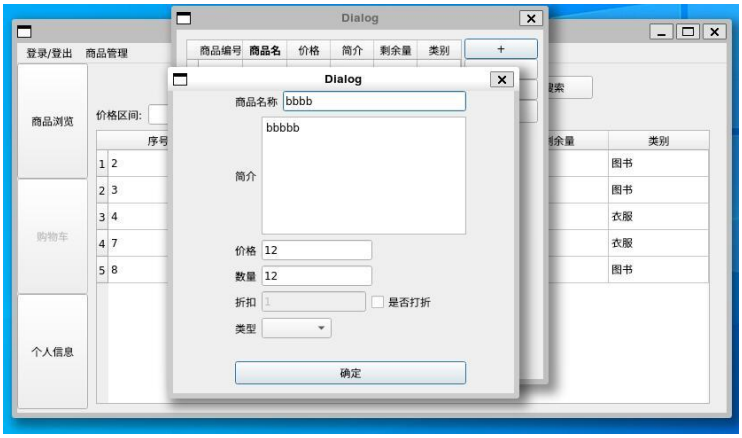
商品管理



添加商品



修改商品



2. 文件结构

为了使 gui 和逻辑分离，我将商品逻辑部分构建出静态库，然后 qt 链接静态库后，再编译。Qt 支持使用 cmake 构建，但是由于又出现了链接错误，我在 qt 部分使用 qmake 进行构建。

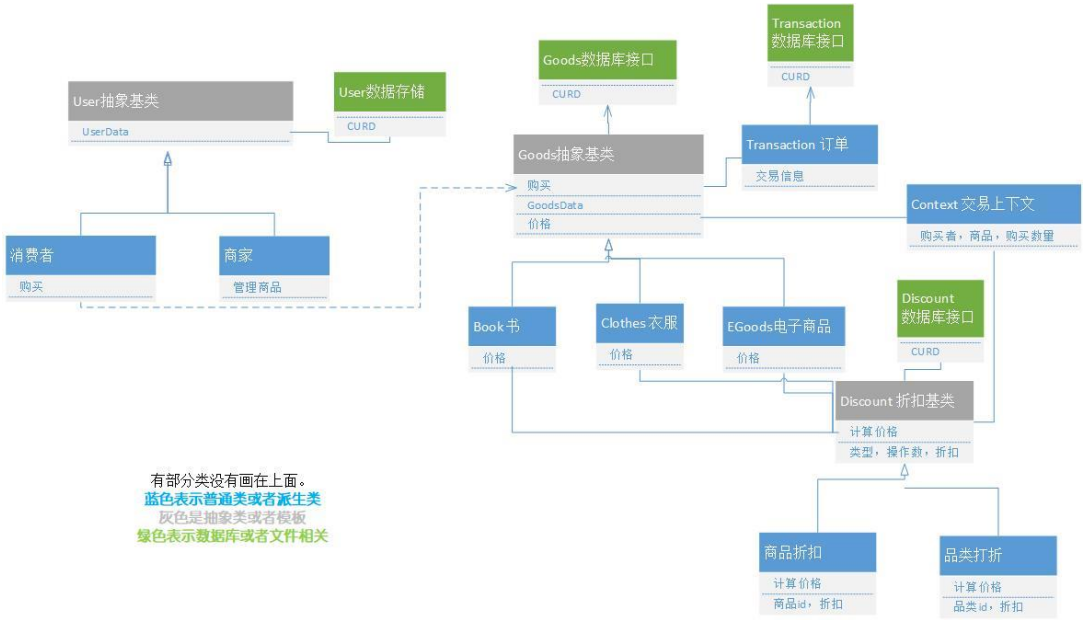
下面 include 文件夹和 src 文件夹是逻辑部分，可以单独运行；而 ui 部分是 qt 的界面。

```
├── CMakeLists.txt   cmake 配置
├── Doxyfile   注释文档生成
├── Readme.md
├── include   头文件
│   ├── concreteGoods.h    商品派生类
│   ├── concreteUser.h     用户派生类
│   ├── database.h         数据库基类接口
│   ├── goods.h            商品基类
│   ├── sqlite3.h          sqlite3 (外部库)
│   └── user.h             用户基类
├── src    实现
│   ├── CMakeLists.txt
│   ├── concreteGoods.cpp   商品派生类
│   ├── concreteUser.cpp    用户派生类
│   ├── database.cpp        数据库基类接口
│   ├── goods.cpp           商品基类
│   ├── sqlite3.c           sqlite3 (外部库)
│   └── user.cpp            用户基类
├── test       测试
└── ui         qt 界面
    ├── adapter.h      商品类型 id 转字符串
    ├── addgoodsdialog.h  添加商品的窗口
    ├── categorydiscount.h 管理品类打折
```

```
├── goodsbuying.h      商品购买页
├── include    链接库的头文件
├── libs       链接库
├── logindialog.h    登录窗口页
├── main.cpp       主函数
├── mainwindow.h   主窗口
├── market.h    商品市场界面
├── profile.h    用户信息界面
├── registerdialog.h   用户注册页
    ├── usergoodsmanagement.h   商家管理商品界面
```

3. Uml 和文档

上图是大致的 UML 图，为了简洁起见，只表示出了关于商品创建、购买流程的主要关



系。一些类没有画出来，类内部具体的方法也没有写出来。下面分模块具体介绍。

我根据 doxygen 格式写了注释，可以自动生成文档。
下面几部分只会提一些比较特别的地方，具体的 api 可以查看文档，不再赘述。

4. 和数据库、文件流的接口

1. User 类和文件流的接口 UserRecord

● 文件流操作

UserRecord 提供对文件流的增删查改操作。User 类通过调用 UserRecord 来获取数据

库数据。这部分的接口不再赘述，下面说明一下对文件流的具体操作。

在文件流操作部分，文件的增加操作比较困难，如果想要在写好的东西里增加一段，就必须得移动后面全部的内容，这和链表很像。为了让文件增加操作更快一些，我想到了两种方法：

第一种就是采取链表的形式，起始时分配一定大小的空间，如果修改操作导致空间不足，那么就移除这个块，然后在文件末尾添加一个新块。这种形式比较通用，缺点可能是还需要定期整理空间，否则中间可能会多出很多空余的地方。

第二种是目前采用的方式，设定用户名和密码的最大长度，然后固定每一行的长度。如果修改的话，只需要跳转到这一行上进行修改就行了，不必移动后面的数据。文件内部的数据大致是这样的：

```
1 aaaaa 123456 0.5 1\0\0\0\0\0.....\0\n
2 bbbbb 123456 0.7 1\0\0\0\0\0.....\0\n
```

如果 aaaaa 修改了密码，那么就会变成这样：

```
1 aaaaa 12345678910 0.5 1\0\0.....\0\n
2 bbbbb 123456 0.7 1\0\0\0\0\0.....\0\n
```

具体的实现如下

```
void UserRecord::insert_data(const UserData&data){
    if(data.id<=0)return;
    set_write_cursor_to_nth_line(data.id);// 游标移到第 n 行
    database<<data.id<<" "<<data.username<<" "<<data.password<<" "<<data.balance<<" "<<data.type;// 依次输入数据
    database.put('\0');database.put('\0');database.put('\0');// 加上\0 防止和之前数据连起来
    write_LF_nth_line(data.id);// 在最后加上换行符
}
void UserRecord::remove_data(int id){
    if(id<=0)return;
    //不判断这一行是否不空行了
    set_write_cursor_to_nth_line(id); // 游标移到第 n 行
    database<<"0\0";database.flush(); // 在开头写上\0\0 表示这一行已经没有数据
}
```

- Cache

除此以外，因为文件操作比较费时，这边做了一个类似 cache 的结构。在每次初始化程序的时候，都会从文件中读取数据到内存，然后后面取数据就直接取的是内存中的数据了。当数据要修改的时候，同时修改内存和文件中的数据。

2. 和 sqlite3 的数据接口

除了 User 类外，其他需要保存的数据都是采用 sqlite3 进行存储。通过 sql 语句插入、修改、删除数据。

起初我采用的方式是做一个通用的模板接口，模板的参数是逻辑类和数据类，数据类主要是用于逻辑类和数据类之间传递数据的。

```
template<typename T_class,typename T_data>
class MetaRecord{//通用基类
    public:
        MetaRecord():db(Database::get_db()){};
        virtual std::shared_ptr<T_class> get(int id)=0; //获取
        int set(const T_data& data); // 创建
        int size()const; // 大小
        virtual void remove(int id); //删除
        void clear();// 移除
    protected:
        virtual void insert_data_to_string(char buffer[],const T_data&)=0; //构造插入的 sql 语句，子类实现
        static const char TABLE_NAME[]; // 表名 子类实现
        sqlite3 *db;
    private:
};
```

这样写有一个问题就是每个类都是实现两个类，一个供用户使用，一个用来传递数据。写一两个类还行，写得多了还是很麻烦的。前期设计也有一些不合理的地方，没有考虑各种情况。因此我只在 GoodsRecord 上面继承了这个接口，在后面写 DiscountRecord 和 TranscationRecord 等类的数据库接口的时候，虽然接口的名字还是一样的，但是我没有再继承 MetaRecord 了，也不再采用两个类的形式了。

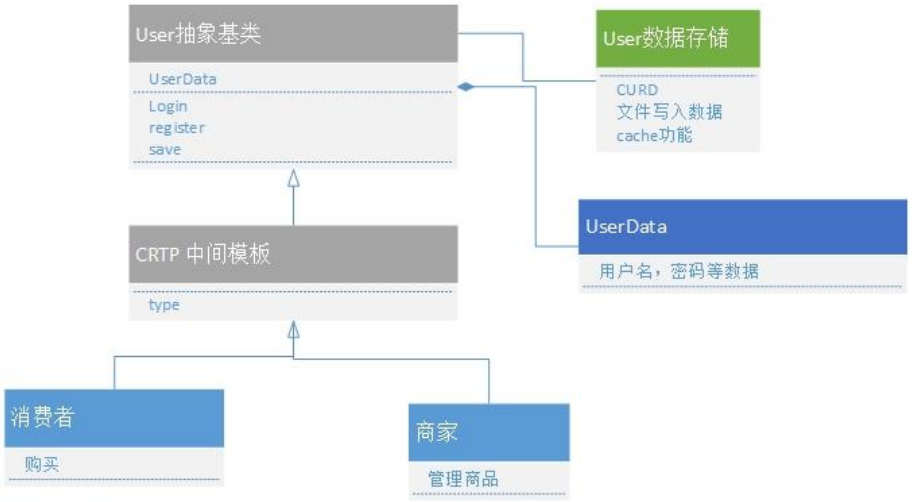
关于 sql 语句的构造，这里我用的是 sprintf 格式化输出到一个缓冲区上，然后交由 sqlite3 接口执行的形式，对于不定长的字符串，如果不检查，可能有缓冲区溢出的风险。比如：

```
std::shared_ptr<Goods> GoodsRecord::get(int id){
    static const char sql[] = "SELECT * FROM %s WHERE ID=%d";
    static char buffer[48];
    GoodsData goods;

    snprintf(buffer,48,sql,TABLE_NAME,id); // 格式化输出
    Database::exec(db,buffer,fetch_in_struct,&goods); // 执行 sql 语句
    return register_types[goods.type](goods);

}
```

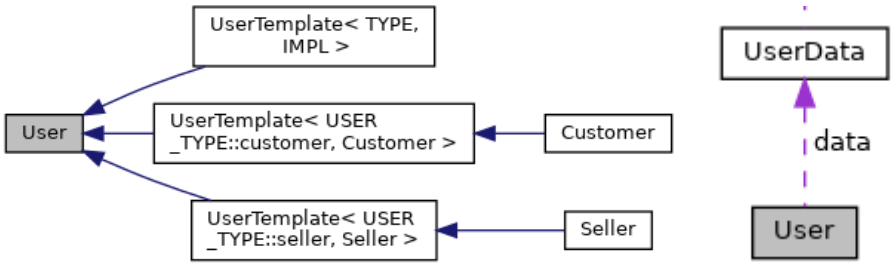
5. 用户类和其相关类



1. User 抽象基类

```
class User{
    public:
        static std::shared_ptr<User> login(const string& username,const string& password);
        static bool register_(int type,const string& username,const string& password);
        bool save();
       //..... 忽略了一些普通的函数
    protected:
        User(UserData* p):data(p){}
        // 成员数据是 protected 的主要是为了给子类用，后面发现这样的写法并不好.
        UserData* data;///< 用户数据
        unsigned short _TYPE;///< 用户类型
```

```
    private:
        static const std::regex USERNAME_PATTERN;// 校验用户名格式
        static const std::regex PASSWORD_PATTERN;// 校验密码格式
};
```



User 抽象基类有两个成员数据：UserData 和 Type。前者用来保存相关数据，后者没有用，只是用来在后面“提醒”编译器初始化静态（见 CRTP 中间模板部分）。

User 类提供了两个静态函数：login 和 register，同时，构造函数都是非 public 的，因此无法通过构造函数在类外构造 User，只能通过 login 来获取一个 User 对象，这样可以保证创建 User 对象时一定需要校验密码。

在这个项目中，有一个问题就是如何获取数据。起初我的想法是用一个 struct（即上图的 UserData）来传递数据，这样可以保证数据只在交换的时候暴露数据。但是后来我发现这并不能解决问题。最后，我把数据都按照形如 get_username(),get_password()的形式都暴露出来了。这样的形式是否违背了封闭性的原则？我觉得没有。《Effective C++》一书中作者认为不应该直接访问数据成员，而是要通过成员函数访问，这样可以保证接口的一致性：即对对象的访问都是 A.xxx(), A.yyyy() 。同时，也可以在接口不变动的情况下修改实现。

User 类还提供了一个 save 函数，修改完数据后，会把新数据提交给文件流。

2. CRTP 中间基类

```
template<int TYPE, typename IMPL>
class UserTemplate: public User{
   enum {_USER_TYPE_ID = TYPE };
   public:
         // 保存子类的构造函数，在父类调用构造的时候，会委托这个函数进行构造
        static std::shared_ptr<User> instance(UserData* data) { return std::make_shared<IMPL>(
data);}
        static const unsigned short USER_TYPE_ID;
       // 把基类指针转换成子类指针
```

```
        static std::shared_ptr<IMPL> cast(std::shared_ptr<User>u);
    protected:
        UserTemplate(UserData* p):User(p) { _TYPE = USER_TYPE_ID; } // 这是必需的
};

/// 动态为所有继承的用户子类注册类型
template <int TYPE, typename IMPL>
const unsigned short UserTemplate<TYPE, IMPL>::USER_TYPE_ID = UserRecord::get_record().registe
r_type(
    UserTemplate<TYPE, IMPL >::_USER_TYPE_ID, &UserTemplate<TYPE, IMPL >::instance);
```

CRTP(Curiously recurring template pattern,奇异递归模板模式)。CRTP 可以解决原生 C++没有反射机制的问题。

起初，我希望能实现通过用户类型，动态创建用户对象的效果。一种比较简单的实现就是用一个 if 或者 switch，在创建时动态判断，比如：

```
std::shared_ptr<User*> create(UserData data){
    switch (data.type)
    {
    case USER_TYPE::Customer:
        return std::shared_ptr<User*>(new Customer(data));
    case USER_TYPE::Seller:
        return  std::shared_ptr<User*>(new Seller(data));
    default:
        return std::shared_ptr<User*>(nullptr);
    }
}
```

这有一个缺点，就是每次添加新的类的时候都必须在 case 语句后面加上新的类型。而且每次都必须修改基类所在的文件，重新编译就得编译基类部分。

CRTP 可以解决这个问题。它的方法是用模板绑定数字和类。然后我们可以用这个特化的模板把原来类的构造函数注册到一个数组中。

比如：

```
class A:public UserTemplate<1,A>{ // Template<*,class>就是 CRTP
```

```
  public:
      A(UserData* p):UserTemplate(p){}
      int get_user_type()const override{
          return 1;
      }
//......
};
```

这样，就创建了一个特化出<数字，当前类>的模板。

接下来需要让模板注册新加的类。这里可以用程序每次执行的时候都会初始化静态成员的特性。所有类模板被特化之后都会执行下面的这一行。register_type 是一个函数，他把当前类的工厂函数（UserTemplate<TYPE, IMPL >::instance）放到一个数组里，其下标是我们给的数字。

```
template <int TYPE, typename IMPL>
const unsigned short UserTemplate<TYPE, IMPL>::USER_TYPE_ID = UserRecord::get_record().register_type
(UserTemplate<TYPE, IMPL >::_USER_TYPE_ID, &UserTemplate<TYPE, IMPL >::instance);

// 工厂函数指针
typedef std::shared_ptr<User> (*p_user_construct)(UserData *);

unsigned short UserRecord::register_type(unsigned short id, p_user_construct factoryMethod){
    register_types[id] = factoryMethod;
    return id;
}
```

这样就实现了动态注册派生类的效果。别的类如果想构造派生类，只需要调用 UserRecord::register_types[id]或者委托 UserRecord 构造即可。

最后还有一个小地方：

```
    UserTemplate(UserData* p):User(p) { _TYPE = USER_TYPE_ID; } // 这是必需的
```

这里是必需的，如果不拿来赋值的话，编译器会不初始化静态成员对象 USER_TYPE_ID（可能是因为没用到，所以编译器优化掉了？）

参考：https://stackoverflow.com/questions/2850213/dynamically-register-constructor-

methods-in-an-abstractfactory-at-compile-time-u

3. 用户类的派生类

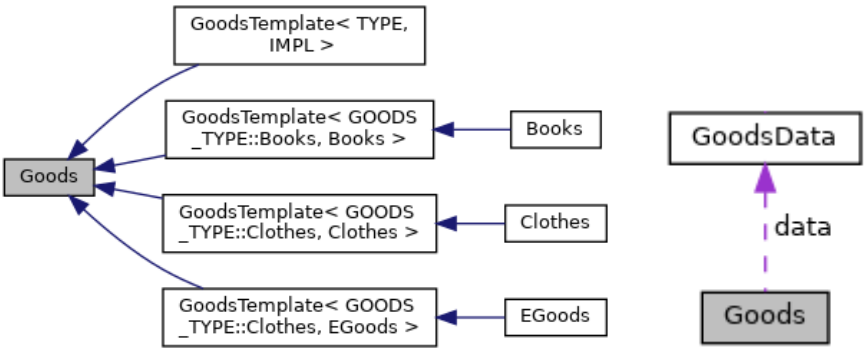
在这一版中，消费者只有购买功能；商家有管理商品的功能，这部分在商品类部分再说明。

商家和商品是组合的关系，即商家有一个指向若干商品的智能指针。因为商家不是每次都需要查看商品，这里做了一个懒加载，等第一次访问商家商品时才加载商品数据。同时，返回的是一个引用，下次加载就不需要再取数据库了。

```
std::vector<std::shared_ptr<Goods>>& goods(){
    if (!has_load_goods){
        auto& record = GoodsRecord::get_record();
        _goods = record.get_user_goods(data->id);
        has_load_goods = true;
    }
    return *_goods;
}
```

6. 商品类和交易相关类

1. 商品基类 Goods 类



商品基类的设计和用户基类的设计类似，都是采用逻辑和数据分离的形式，商品类保管一个商品数据 GoodsData 的对象。同时，商品类也做了一个 CRTP 的中间层，派生类可以通过继承自动注册。原理类似，不再赘述

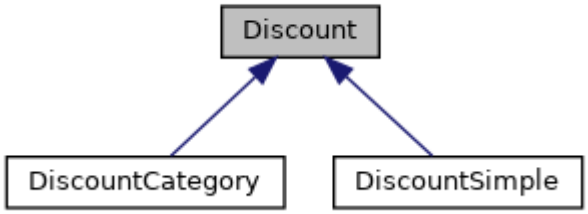
2. 购买时上下文 Context 类

这个类主要是把购买时信息包装其他给 Discount 计算价格的。

这个类设计的不是很好，因为只能包含一种商品的信息，只能算一类商品的价格；比较好 context 应该包含订单商品的全部信息

Context 内部有商品对象指针，用户指针，购买数量三个属性。Context 只在计算价格时被构造并传递给 Discount 的 price 函数。

3. 折扣 Discount 类



在折扣类上，为了同时实现品类打折和商品打折，这里我采用的是类似解释器模式的设计。Discount 主要有 4 个字段：类型，操作数，效果值，阈值。
类型有以下几种：

枚举值	
OFF	单品打折
TYPE_DISCOUNT	单品满减，未用
TYPE_OFF	同品打折
TOTAL_DISCOUNT	同品满减，未用
TOTAL_OFF	全单打折 全单满减，未用

由于时间原因，只做了单品打折和品类打折。
操作数对于单品打折，指的是商品 id，对于品类打折，指的是类型 id。
效果值对于这两个打折，都是折扣数；如果对于满减，那可以是满减的金额。
阈值是触发效果的值，如果大于阈值，才会触发打折效果。

比如，单品打折，商品 id=1，7 折，无阈值的情况下，它的数据是：

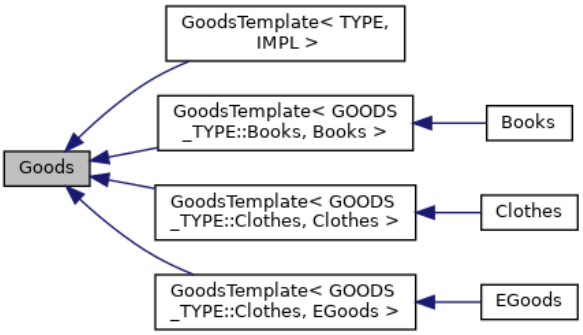
类型	操作数	效果值	阈值
0(单品打折)	1	0.7	-1（无穷大）

对于折扣的基类，有一个虚函数 price()；对于每个种类的折扣，都有一个对应的派生类，它重写了基类的 price 函数。例如，对于单品打折的 price 函数：

```
class DiscountSimple:public Discount{
    public:
        DiscountSimple(int id,unsigned char type,int operand,double discount,double threshold=
-1):Discount(id,type,operand,discount,threshold){}
        double price(const GoodsContext&context)override{ // 重写基类 price 函数
            if ( (_threshold<=0) || (_threshold>0&& context.goods().price() * context.num() >_
threshold)){ // 如果阈值小于 0（那么认为阈值为无穷大） 或者 有阈值并且金额大于阈值了
                return context.goods().price() *context.num()*  _discount; //打折价格
            }else{
                return context.goods().price()*context.num(); //没打折价格
             }
        }
```

```
};
```

4. 商品派生类



有三个商品派生类，它们也是按照 CRTP 的形式从 goods 中继承得到的。

7. QT

这部分主要介绍界面功能。

```
── ui          qt 界面
    ├── adapter.h       商品类型 id 转字符串
    ├── addgoodsdialog.h   添加商品的窗口
    ├── categorydiscount.h 管理品类打折
    ├── goodsbuying.h       商品购买页
    ├── logindialog.h     登录窗口页
    ├── main.cpp        主函数
    ├── mainwindow.h    主窗口
    ├── market.h      商品市场界面
    ├── profile.h     用户信息界面
    ├── registerdialog.h   用户注册页
    ├── usergoodsmanagement.h   商家管理商品界面
```

1. 主界面

右边这个黑色的框框是主要显示的区域，商品市场界面(market.h)或者用户信息界面（profile.h）可以显示在右边。当切换页面时，原页面不会被完全关闭，而是展示隐藏起来。比如当前在商品浏览页，如果点击②进入个人信息页，那么商品浏览页会被隐藏，个人信息页被打开，而下次再点击①时，原页面不会被刷新。

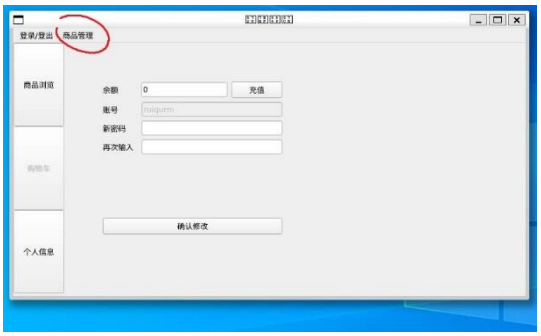
双击表格中的一个项会打开购买页，购买页中无法输入比剩余量大的值，也无法输入比 0 小的值。

搜索如果是空的话，会返回全部商品。

2. 个人信息界面

- 登录界面不需要选择用户类型，会动态选择类型。

- 点击个人信息页时，如果未登录，会弹出登录界面
- 商家登录后，上方会自动出现商品管理的选项



- 点击充值，弹出窗口进行充值。
- 修改密码后，提示已经登出，需要重新登录。

3. 商品管理

- 添加和修改商品时，如果有异常的值，会在下面提示。

- 管理品类打折时，为了方便起见，下面获取了所有的品类打折，可以点击折扣率修改折扣。

8. 测试

```
test/
├── test.h              测试用的一些函数
├── test_buying.cpp     测试购买功能
├── test_discount.cpp   测试用户功能
└── test_goodsrecord.cpp  测试商品保存
```

```
├── test_transaction.cpp    测试订单生成
└── test_user.cpp    测试用户
```

这几个测试文件测试的是基本的功能，但是对于一些像内存未释放、指针错误的问题没有测试。

9.其他

在验收的时候，突然出现了一个段错误。事后查看发现这一个部分有一个 BUG：注册第二个新用户的时候会发生段错误。这个 bug 在后面被修复了，出现错误的原因可能是下面这段：

```
void UserRecord::insert_data(const UserData&data){
    if(data.id<=0)return;
    set_write_cursor_to_nth_line(data.id);///> 游标移到第 id-1 行
    database<<data.id<<" "<<data.username<<" "<<data.password<<" "<<data.balance<<" "<<data.ty
pe<<"\0\0\0\0"; 这里有问题
    write_LF_nth_line(data.id);///> 在最后加上换行符
}
```

fstream<<"\0\0"这种写法是有问题的。因为\0 被当做字符串结尾，因此没能把\0 输入到文件里，造成后续的错误。正确的写法如下：

```
database.put('\0');database.put('\0');database.put('\0');///> 加上\0 防止和之前数据连起
```

三、交易管理子系统（单机版）

这一部分因为时间原因，没有带注释。可以看第三版的注释

1. 概览

购物车

订单

在个人信息界面查看未支付的订单

2. 文件结构

```
├── CMakeLists.txt   cmake 配置
├── Doxyfile   注释文档生成
├── Readme.md
├── include   头文件
|   ├── concreteGoods.h    商品派生类
|   ├── concreteUser.h    用户派生类
|   ├── database.h    数据库基类接口
|   ├── goods.h    商品基类
|   ├── sqlite3.h    sqlite3 (外部库)
|   ├── transaction.h    交易类，购物车
|   └── user.h    用户基类
├── src    实现
|   ├── CMakeLists.txt
|   ├── concreteGoods.cpp   商品派生类
|   ├── concreteUser.cpp    用户派生类
|   ├── database.cpp    数据库基类接口
|   ├── goods.cpp    商品基类
|   ├── sqlite3.c    sqlite3 (外部库)
|   ├── transaction.cpp    交易类，购物车
|   └── user.cpp    用户基类
├── test    测试
```

3. 购物车

购物车是一个不定长的列表，数据库保存的话用上面那种方式反而麻烦了。这里我用的是软外键的形式。表里数字是 int，而不是外键类型。

例如下面的表。需要获取 USER 1 的购物车时，只需要查询键 USER=1 的所有项即可。这样就保存了用户的购物车。

ID	USER	GOODS	COUNT
21	1	1	2
27	1	2	5
28	1	3	5
44	2	2	3
45	2	1	3

4. 订单类

在这一部分，我把上一部分的订单类修改了一下，以保存多种商品。因为 transaction 要保存的商品数据是静态的，订单是无法修改的（至少在本项目中是无法修改的），因此数据库中可以用格式化的字符串来保存商品信息。

最后的设计是这样的：在创建订单时，需要传入一个字典，其键为商品 id，其值为商品名，单品最终价格，数量的元组。然后在数据库写入时，用换行符隔开保存。

下面是一个订单在数据库中的实际格式：

第一个数字是商品的数量，第 2~5 行是第一件商品，第 6~9 行是第二件商品的信息。

订单还保存了时间戳，可以获取创建订单的时间。

订单可以取消和确认。取消时，订单会被移除，而确认时，订单的 FINISHED 被改为 1.

5. 购买

购买时，用户传入他的购物车和选择的选项。然后逐一检查每个购买的项，如果购买的数量大于当前数量，或者用户作为商家买自己卖的东西，那么会返回错误。此处的返回值是一个字符串和 vector 的元组，字符串是错误信息，vector 是生成的订单 id，因为可能购买的商品中可能会有多个商家，对于每对商家-卖家，都会产生一张订单。

接下来遍历每一份将要生成的订单，从购物车中删除它们，计算价格，生成对应的描述，累计价格，最后生成一张订单。

```
std::tuple<string,vector<int>> Customer::buy(Cart&cart,std::vector<int> &selection){
    auto&record = GoodsRecord::get_record();
    auto data =  record.get(selection);
    if (data->size()<selection.size())return {"存在已删除的商品",vector<int>()};
    if(selection.size()==0)return {"传入商品数量为空",vector<int>()};
    // int seller = -1;
    map<int,vector<Goods*>> tasks; //分类不同的商家的产品
    for(auto& goods:*data){
        if(goods->remain()<cart[goods->id()]){
            return {"商品数量不足",vector<int>()};
        }
        tasks[goods->seller()].push_back(goods.get());// 把不同商家的商品放到对应列表中
    }
    if(tasks.find(this->data->id)!=tasks.end()){
        return {"尝试自交易",vector<int>()};//自交易
    }
    vector<int> transaction_ids;
    transaction_ids.reserve(tasks.size());
    for (const auto &[seller, goods_vector] : tasks){
       // 遍历每一份订单
        //c++ 17 的写法
        //https://stackoverflow.com/questions/4207346/how-can-i-traverse-iterate-an-stl-map
        std::map<int,std::tuple<std::string,int,double>>m;
        double total = 0;
        for(auto& goods:goods_vector){
            int count  = cart[goods->id()];  //查看购买的商品量
            cart.remove(goods->id());   //在购物车中移除商品
```

```
            GoodsContext context(goods,count,this); // 构造上下文
            double p = goods->get_price(context); //获取价格
            m[goods->id()] = std::make_tuple(
                goods->name(),
                count,
                p
            ); //生成订单描述，按照上面订单说明的方法保存
            total += p; //统计总价
        }
        std::cout<<"本份订单有"<<goods_vector.size()<<"条"<<std::endl;
      Transaction t(seller,this->data->id,total,m); //生成订单。生成订单成功时会自动扣除商品的数量
        transaction_ids.push_back(t.id()); // 把订单号放进列表
    }


   return {"",transaction_ids};
}
```

四、电商交易平台（网络版）

1. 概览

服务端日志

```
22:37:28 INFO  /home/ruiqurm/bupt/cppcourse/online_retailers/part3/server/src/main.cpp:45: listen on 0.0.0
.0:12345
22:37:36 INFO  /home/ruiqurm/bupt/cppcourse/online_retailers/part3/server/src/main.cpp:87: server: got con
nection from 127.0.0.1 port 55714  in sockfd=5
22:37:37 DEBUG /home/ruiqurm/bupt/cppcourse/online_retailers/part3/server/./include/server.h:397: id =1528
1,type=68,length=0
22:37:37 DEBUG /home/ruiqurm/bupt/cppcourse/online_retailers/part3/server/src/server.cpp:228: 获取全部商品
22:37:37 DEBUG /home/ruiqurm/bupt/cppcourse/online_retailers/part3/server/src/server.cpp:233: 返回了7个商
品
22:37:45 DEBUG /home/ruiqurm/bupt/cppcourse/online_retailers/part3/server/./include/server.h:397: id =1528
9,type=0,length=17
22:37:45 DEBUG /home/ruiqurm/bupt/cppcourse/online_retailers/part3/server/src/database.cpp:166: 读取本地用
户数据:
22:37:45 DEBUG /home/ruiqurm/bupt/cppcourse/online_retailers/part3/server/src/database.cpp:183: username=r
uiqurm password=123456 type=1 balance=100.000000
22:37:45 DEBUG /home/ruiqurm/bupt/cppcourse/online_retailers/part3/server/src/database.cpp:183: username=a
aaa password=123456 type=0 balance=77207.000000
22:37:45 INFO  /home/ruiqurm/bupt/cppcourse/online_retailers/part3/server/src/database.cpp:154: init user-
record done
22:37:45 INFO  /home/ruiqurm/bupt/cppcourse/online_retailers/part3/server/src/database.cpp:20: 找到名字
22:37:45 DEBUG /home/ruiqurm/bupt/cppcourse/online_retailers/part3/server/src/server.cpp:86: token=IGLuxm7
j4oP4XqVj username=ruiqurm,type=1
22:37:45 DEBUG /home/ruiqurm/bupt/cppcourse/online_retailers/part3/server/./include/server.h:397: id =1528
9,type=256,length=2
22:37:45 DEBUG /home/ruiqurm/bupt/cppcourse/online_retailers/part3/server/src/server.cpp:454: 获取用户1的
购物车数据共0条
22:37:47 DEBUG /home/ruiqurm/bupt/cppcourse/online_retailers/part3/server/./include/server.h:397: id =1529
1,type=67,length=2
22:37:47 DEBUG /home/ruiqurm/bupt/cppcourse/online_retailers/part3/server/src/server.cpp:215: 获取商家id=1
的商品
22:37:47 DEBUG /home/ruiqurm/bupt/cppcourse/online_retailers/part3/server/src/server.cpp:221: 返回了7个商
品
22:37:50 DEBUG /home/ruiqurm/bupt/cppcourse/online_retailers/part3/server/./include/server.h:397: id =1529
4,type=71,length=2
22:37:50 DEBUG /home/ruiqurm/bupt/cppcourse/online_retailers/part3/server/src/server.cpp:266: 删除商品
DELETE FROM GOODS WHERE ID = 5;
22:37:53 DEBUG /home/ruiqurm/bupt/cppcourse/online_retailers/part3/server/./include/server.h:397: id =1529
7,type=192,length=2
22:37:53 DEBUG /home/ruiqurm/bupt/cppcourse/online_retailers/part3/server/src/server.cpp:402: 获取到和1有
关的记录共0条
22:37:56 DEBUG /home/ruiqurm/bupt/cppcourse/online_retailers/part3/server/./include/server.h:397: id =1530
0,type=3,length=28
22:37:56 INFO  /home/ruiqurm/bupt/cppcourse/online_retailers/part3/server/src/server.cpp:112: length=28,us
ername=ruiqurm,password=123456
1 ruiqurm 123456 129 1
22:38:03 DEBUG /home/ruiqurm/bupt/cppcourse/online_retailers/part3/server/./include/server.h:397: id =1530
7,type=2,length=0
22:38:07 DEBUG /home/ruiqurm/bupt/cppcourse/online_retailers/part3/server/./include/server.h:397: id =1531
1,type=0,length=14
22:38:07 INFO  /home/ruiqurm/bupt/cppcourse/online_retailers/part3/server/src/database.cpp:20: 找到名字
22:38:07 DEBUG /home/ruiqurm/bupt/cppcourse/online_retailers/part3/server/src/server.cpp:86: token=Y2EhbsI
OixfhsCaW username=aaaa,type=0
22:38:07 DEBUG /home/ruiqurm/bupt/cppcourse/online_retailers/part3/server/./include/server.h:397: id =1531
1,type=256,length=2
22:38:07 DEBUG /home/ruiqurm/bupt/cppcourse/online_retailers/part3/server/src/server.cpp:454: 获取用户2的
购物车数据共0条
22:38:09 DEBUG /home/ruiqurm/bupt/cppcourse/online_retailers/part3/server/./include/server.h:397: id =1531
3.type=192.length=2
```

客户端

2. 项目文件

```
├── CMakeLists.txt    cmake 配置
├── Doxyfile      dox 文档配置
├── Readme.md
├── client    客户端源码
│   ├── CMakeLists.txt
│   ├── include
│   │   ├── concreteGoods.h    商品派生类
│   │   ├── concreteUser.h      用户派生类
│   │   ├── database.h          数据库和 socket
│   │   ├── goods.h             商品，折扣
│   │   ├── transaction.h       订单，购物车
│   │   └── user.h              用户基类
│   ├── src
│   │   ├── CMakeLists.txt
│   │   ├── concreteGoods.cpp  商品派生类
│   │   ├── concreteUser.cpp     用户派生类
│   │   ├── database.cpp         数据库和 socket
│   │   ├── goods.cpp            商品，折扣
│   │   ├── transaction.cpp      订单，购物车
│   │   └── user.cpp             用户基类
│   └── test
│       ├── CMakeLists.txt
│       ├── test_user.cpp        测试用户
│       └── testfunction.h
├── common     基础类
│   ├── CMakeLists.txt
│   ├── log.c    日志
```

```
|   ├── log.h   日志
|   ├── protoData.pb.cc   protobuf 生成
|   ├── protoData.pb.h   protobuf 生成
|   ├── protoData.proto   protobuf 源文件
|   ├── protocol.cpp   协议
|   └── protocol.h   协议
├── server   服务端源代码
|   ├── CMakeLists.txt
|   ├── include
|   |   ├── database.h   sqlite3 数据库操作
|   |   ├── server.h   执行器及其派生类
|   |   └── sqlite3.h   sqlite3（外部）
|   ├── src
|   |   ├── CMakeLists.txt
|   |   ├── database.cpp   sqlite3 数据库操作
|   |   ├── main.cpp   服务器主函数
|   |   ├── server.cpp   执行器及其派生类
|   |   └── sqlite3.c   sqlite3（外部）
|   └── test   测试
|       ├── CMakeLists.txt
|       ├── test_cart.cpp   测试购物车（网络）
|       ├── test_discount.cpp   测试折扣（网络）
|       ├── test_goods.cpp   测试商品（网络）
|       ├── test_transaction.cpp   测试订单（网络）
|       ├── test_user.cpp   测试用户（网络）
|       └── testfunction.h
└── ui   qt ui 文件
```

3. 网络部分的设计

这一部分在网络交互部分有很多方法。

最简单的方法就是直接把服务端的数据库下载到本地，对于读取操作，读取本地即可；对于更新和删除操作，重新把数据库上传到服务器上；服务器如果监听到更新操作，那么把就再发数据回其他客户端。当然这种操作很蠢，很浪费带宽。

还有一种也是相对简单的方法是直接传 sql 语句到对端。然后对端只作为一个 sqlite3 远程的 shell。这种方式的缺点就是不方便进行权限控制；且用户是无状态的。

最后我采用的方式是类似 RPC 的方式。客户端调用指定编号的服务带上数据，服务端返回所请求的数据。

数据的序列化方面，起初我是想用计算机网络上学到的成帧方法插入分隔符实现序列

化的，但是后面时间比较紧张，加上预估的工作量可能挺大，就放弃了。最后决定采用现成的序列化工具。查阅资料后发现，protobuf 是一个很好用的工具，唯一的缺点可能是不太好移植（要提前编译好 protobuf 的库）

Protobuf 的介绍

Protobuf 是一种语言无关、平台无关、可扩展的序列化结构数据的方法，它可用于（数据）通信协议、数据存储等。可类比 XML，但是比 XML 更小（3 ~ 10 倍）、更快（20 ~ 100 倍）、更为简单。

在本部分项目文件中，common/protoData.proto 文件就是 protobuf 的源文件。通过 protoc 命令可以生成一个 protoData.pb.h 和 protoData.pb.cc 文件，引入头文件并一起编译就可以使用 protobuf 的类了。

举个例子：
例如下面这个 protobuf 类。

```
message Goods{
    bytes name = 1;
    bytes description = 2;
    double price = 3;
    int32 seller = 4;// primary key of seller
    int32 id = 5;
    int32 type = 6;
    int32 remain = 7;
}
```

可以生成出来一个类似之前 GoodsData 的类。通过 set_name,set_price 等自动生成的函数，可以给它赋值。Goods.SerializeToArray 可以导出二进制格式的数据到字符缓冲区中，而 Goods.ParseFromArray 可以从缓冲区中读出二进制数据。

1. 协议

```
0             16            32bit
--------------------------------
|    id       |    type      |
--------------------------------
|   status    | 回复 |  未用  |
--------------------------------
|           长度              |
--------------------------------
|      token(16字节,略)       |
--------------------------------
|（不定长，使用protobuf序列化）  |
--------------------------------
```

为了方便传送数据，这边还设计了一个基于 tcp 的简单协议。上面展示的是头部分，头部分一共 28 字节。Id 是报文编号，type 是报文种类，有以下报文种类：

枚举值	
USER_AUTHENTICATE_PASSWORD	登录
USER_LOGOUT	登录后校验密码
USER_UPDATE	登出
USER_REGISTER	更新用户数据
USER_INFO	注册
GOOOS_CREATE	用户信息
GOOOS_GET_BY_ID	创建商品
GOODS_GET_BY_ID_MULITIPLE	通过id获取商品
GOODS_GET_BY_SELLER	>通过id获取多个商品
GOODS_GET_ALL	通过卖家id获取商品
GOODS_GET_BY_NAME	获取全部商品
GOODS_UPDATE	通过名字模糊查找获取商品
GOODS_REMOVE	更新商品信息

DISCOUNT_CREATE	移除商品
DISOCUNT_UPDATE	创建折扣
DISCOUNT_GET_ALL_CATEGORY	更新折扣
DISCOUNT_GET_GOODS_DISCOUNT	获取所有品类折扣
DISCOUNT_GET_CATEGORY_DISCOUNT	获取某商品的折扣
DISCOUNT_REMOVE_BY_GOODS	获取某品类折扣
DISCOUNT_REMOVE	通过商品id移除折扣
TRANSACTION_GET_BY_USER	移除折扣
TRANSACTION_GET	通过用户id获取所有交易订单
TRANSACTION_SET	通过订单id获取订单
TRANSACTION_CANCEL	创建订单
TRANSACTION_SET_FINISHED	取消订单
CART_GET	完成订单
CART_SET	获取购物车
CART_REMOVE	新建购物车项

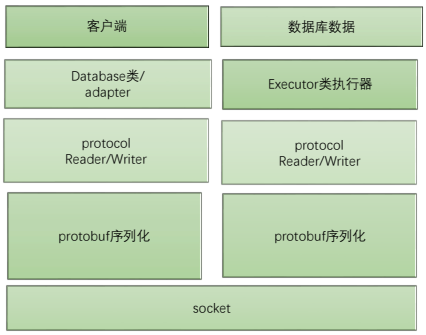
Status 是报文状态，当回复=1（表示是回复报文）时有效，有以下值：

枚举值	
OK_LOGOUT	成功
AUTHENTICATE_FAILED	成功，且已退出登录
PARAMETER_ERROR	认证失败
RUN_FAILED	参数错误
PERMISSION_DENIED	运行失败

长度部分是数据段的长度，不包括头部的长度。

Token 部分是用户登录后反还的令牌。大部分操作需要用户带着令牌才能使用。

2. 客户端访问服务端的流程



客户端上层代码和之前是一样的，修改的只是访问数据库的部分。

在写报文部分，我封装了 protocolReader/protocolWriter 的类，负责自动写报头和数据部分。其写数据部分类似下面这一段：

```
int ProtocolWriter::load(const protoData::Goods&goods){ // goods 是一个 protobuf 对象
    // buffer 是缓冲区，缓冲区前 28 字节已经写好头了
   goods.SerializeToArray(buffer+sizeof(Protocol),goods.ByteSizeLong());//把二进制数据导到缓冲区
    add_length(goods.ByteSizeLong());// 更新报头的长度字段
    return _size; // 返回新长度
}
```

然后封装好的报文就可以发送了。

数据库接口类有一个 Database 对象，这个对象负责发送和接收报文。Send 函数如下：

Send 函数首先加了锁；然后发送数据，如果收到 0 字节跳过（不知道为什么一直收到 0 字节的 tcp 报文？），如果发送失败或者接受失败那么异常退出；最后，判断报文是否有数据，有数据返回 2，没有返回 1。

```
int Database::send(ProtocolWriter& w,ProtocolReader&r){
    g_lock.lock();// 互斥锁
    int n;
    if ((n = ::send(sockfd,w.buf(),w.size(),0))<=0){
        // 如果发送报文失败
        printf("ERROR writing to socket,n=%d",n);
        g_lock.unlock();
        return false;
    }
    n=0;
    // 如果接收到 0 字节，跳过
    while( (n = recv(sockfd,r.buf(),r.buf_size(),0))==0)printf("接收失败\n");
    if(n<0){
        // 接收错误，退出
        printf("ERROR writing to socket,n=%d",n);
        g_lock.unlock();
        return false;
    }
    if (r.status() != Protocol::OK && r.status() != Protocol::OK_LOGOUT){
        // 如果报文正常
        printf("r.status =%d, %s\n",r.status(),protocol_status_to_str(r.status()));
        g_lock.unlock();
        return false;
    }
    if(r.length()>0){// 如果有收到数据
        g_lock.unlock();
        return 2;
    }
    g_lock.unlock();
    return 1;// 如果没收到数据（但有收到头）
}
```

对于服务端，处理也是类似的。只不过服务端是多线程的。

服务端处理客户端各种询问的是 executor 类及其派生类。

处理的客户端请求事务的线程函数如下:

1. 接收数据
2. 加锁
3. 调用对应类型的执行器，执行器除了执行外，还会校验 token，并生成出 user 对象（除了登录和注册）
4. 保存 user 对象的指针（不会被析构，因为全局只有一份 user 对象，,在下一轮可以使用
5. 解锁，回到 1

由于有 token 的存在，即使客户端断开了连接，下次连接时带着 token 也不需要再校验了。这边的 token 做的也比较粗糙，没有做 TTL，只是维护了登录状态的唯一性。此外，由于时间原因，也没有权限控制，任何用户可以访问任何的服务类型（也就是说其实 A 用户可以利用这点给 B 用户创建商品，或者给自己加钱）。这样服务端还是很不完善的。

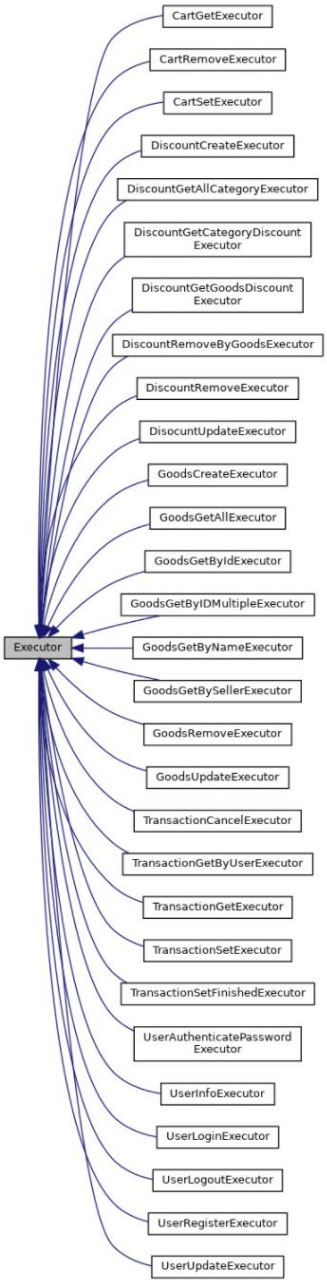
```
void handle_with_request(int sockfd){
    int size,n;
    char read_buffer[8192];
    char write_buffer[8192];
    Executor* executor;
    protoData::User*user=nullptr; // 执行用户数据的指针
    bool has_authenticated = false;// 是否登录
    while(true){
        size = 8192;

        n = recv(sockfd,read_buffer,size,0);
        if (n <= 0) {
            if(n==0){
                log_info("sockfd=%d close connection",sockfd);
            }else{
                log_error_shortcut("ERROR reading from socket");
            }
```

```
            break;
        }// 判断接收状态
        g_mutex.lock();// 锁
        if ((executor=ExecutorFactory::get_executor(read_buffer,write_buffer,user))!=nullptr){
            executor->exec();
            user = executor->get_user(); // 获取登录后的用户，并将在
        }
        g_mutex.unlock();// 解锁
        n = send(sockfd,write_buffer,executor->size(),0); // 发送数据
        if (n <= 0) {// 如果发送失败了
          delete executor;
          executor = nullptr;
          log_error("ERROR writing from socket");
          break;
        }
        delete executor;
        executor = nullptr;
    }
    g_mutex.unlock();// 解锁
    close(sockfd);
}
```

其中的 executor 指针是指向一个子类的基类指针。对于每一种服务类型，都有一个对应的 executor 派生类。每个派生类只需要实现基类的 execImp 函数即可。

这里没有用 CRTP 的形式，而是采用 switch 判断类型，选择派生类的形式。主要是因为派生类和基类都在一个文件里。当然，如果使用 CRTP 可能可以省去一大堆分支判断。



4. Log 功能

服务端的 log 函数是一个 c 函数。这是计算机网络课程设计写的东西，因为这里刚好也用到了，就拿来用了。

这个日志功能支持多等级(TRACE,DEBUG,INFO,WARN,ERROR,FATAL)、彩色输出，同时支持输出日志到文件。其打印格式为

其调用方式为调用宏，比如 log_trace(...),log_debug(...),log_info(...)等等。

打印出来的效果如下:

```
22:37:36 INFO  /home/ruiqurm/bupt/cppcourse/online_retailers/part3/server/src/main.cpp:87: server: got con
nection from 127.0.0.1 port 55714  in sockfd=5
```

五、心得体会

这个项目是我写的第一个规模比较大的项目（虽然有很多都是没有用的）。通过这个项目，我快速入门了 cmake/ctest，sqlite3，qt，protobuf，doxygen 等 c++常用的第三方库；在 C++特性方面，我尝试应用了抽象类，模板，智能指针，C++11 和 C++17 的新遍历容器方式等；设计模式方面，我学习并实践了 CRTP，单例模式，工厂模式等等。总的来说，虽然都是浅尝辄止，但是收获还是颇丰。

- **遇到的问题**

在这个项目中，我处理最久的是设计问题和调试程序。比如用户基类的设计，用户数据要怎么存储，折扣要怎么做成通用的等问题。为此，我查了很多资料，也尝试了很多解决方案，最终也得到了解决

在程序逻辑方面，由于项目代码量还是比较大的，我犯了不少特别低级的错误。由于程序依赖关系比较复杂，不太好用 gdb 进行调试，我都是使用 print 来检查错误的。在犯了多次相同的错误后，我深刻的意识到代码规范，测试文件和日志的重要性。代码规范包括命名和注释，如果前期就写好命名和注释，就可以大大减少后面进行 debug 的时间；日志主要是为了调试获取信息而存在的；测试文件能够测试功能的正确性（也能排除掉一些错误）。

项目使用 cmake 构建，主要是想学习一下 cmake，并顺便体验一下从零开始构建的感觉。实际体验并不好，我花了大量时间学习 cmake 语法，遇到了一大堆奇怪的问题，如果使用 vs 去做的话可能就能节省出一大堆时间。

- **对于本课程的建议**

建议以后提供一些初始代码（start code），比如商品数据的数据库操作可以封装成接口供学生调用。从零开始写一个项目要考虑的东西很多，我们很难集中精力去设计类的框架，反而是花了大量时间在搭建环境，学习界面，学习数据库操作等等，面向对象的思想之类反而被搁置在一边了。