面向对象程序设计与实践

(C++)2

成绩管理系统

程序说明书

班级：2014211314

姓名：叶文霆

学号：2014211519

班内序号：06

日期：2016.7.2

I. 第一部分：初级版的设计
A. 设计要求 2
B. 类设计 3
C. 实现方法 6
II. 第二部分：高级版的设计
A. 设计要求 7
B. 实现思路 8
C. 代码重构 9
III. 第三部分：网络版的设计
A. 设计要求 10
B. 实现思路 10
C. 实现方法 12
D. 代码重构 15

第一部分：初级版的设计

一、设计要求

设计一个命令行版本的成绩管理系统，实现以下几个功能：

- 老师、学生的登录/注销
- 显示老师学生的基本信息
- 支持学生选修必修课、查看所有必修课
- 支持老师查看学生成绩、修改某个学生成绩
- 计算和显示绩点

二、类设计

1. 老师&学生类设计

首先需要明确的是我们一共有四种类型的数据需要存储：老师、学生、选修课和必修课。其中老师和学生同属于Person类，而选修课与必修课同属于Course类，老师、学生与课程他们之间的调用关系大概是这样的：

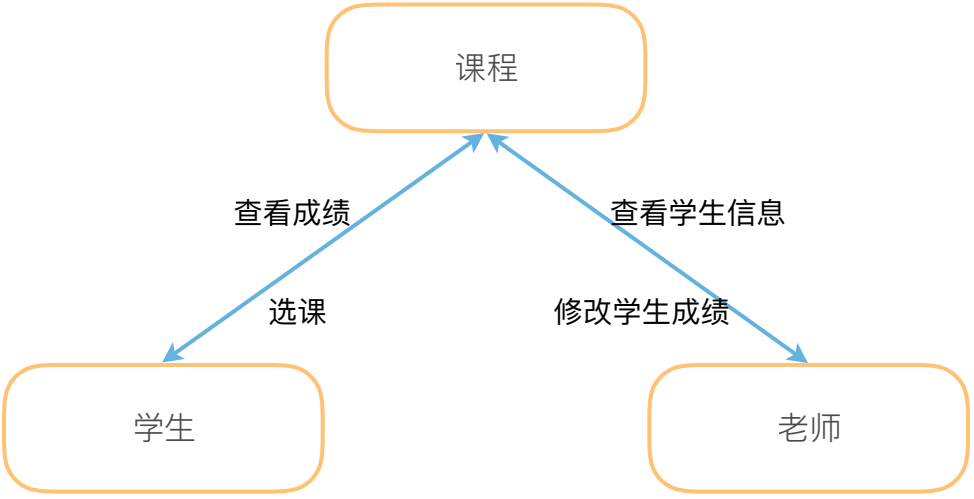
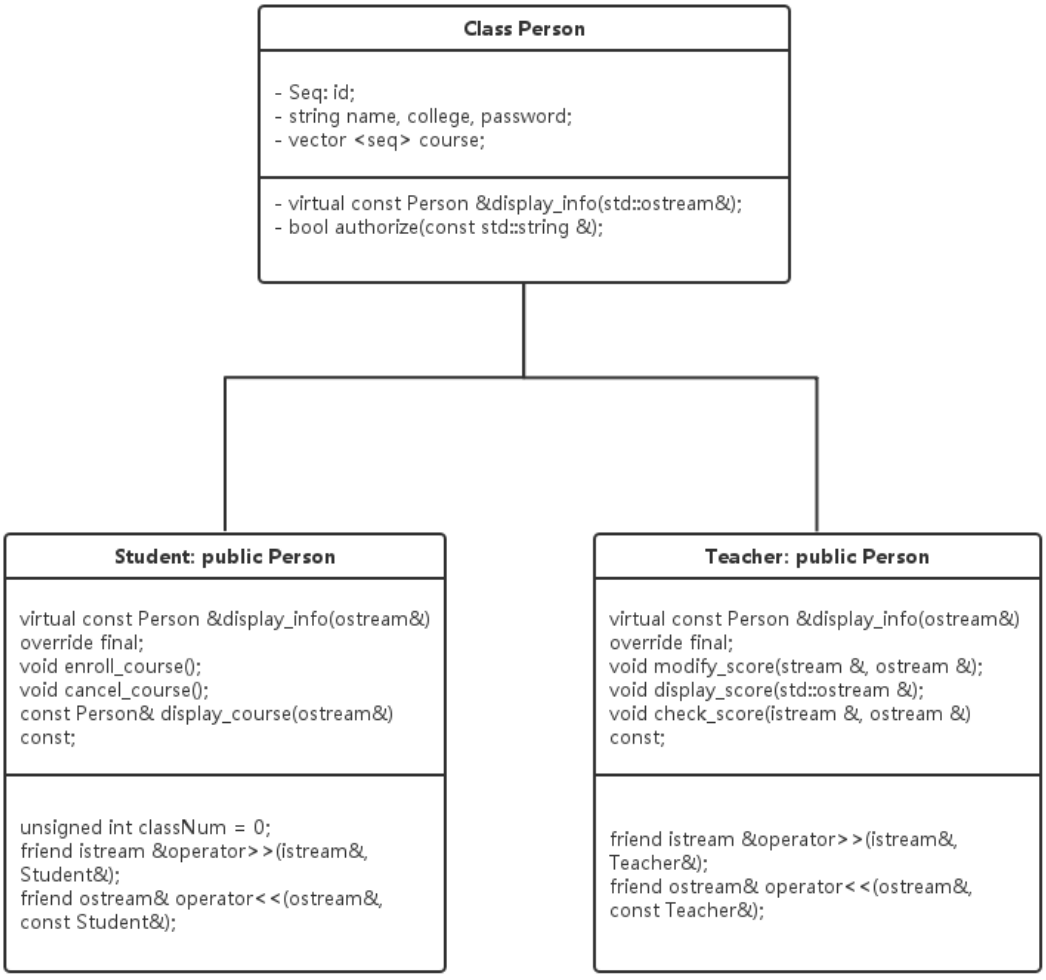


图1 类之间的调用关系

基于以上特征，我们在用户类与课程类里面都存储一个与之存在对应关系的对象的下标。具体类设计如下（去掉一些简单逻辑的成员函数）：



2. 课程类设计

课程类中一共有两个子类：必选课类与选修课类。他们都共同享有课程号，授课老师，学生成绩等成员变量，也都有显示课程基本信息，获得学生成绩等成员函数。二者的区别在于：

1. **选课与取消课时的反馈不同。**必修课不允许选择或退课，而选修课正好相反
2. **展示效果不同。**根据要求不同类型的课程应该区分展示。
3. **绩点计算公式不同。**根据题目要求选修课与必修课的绩点计算方式不同。

将二者不同之处用虚函数以及其重载来实现，最终课程类设计如下：

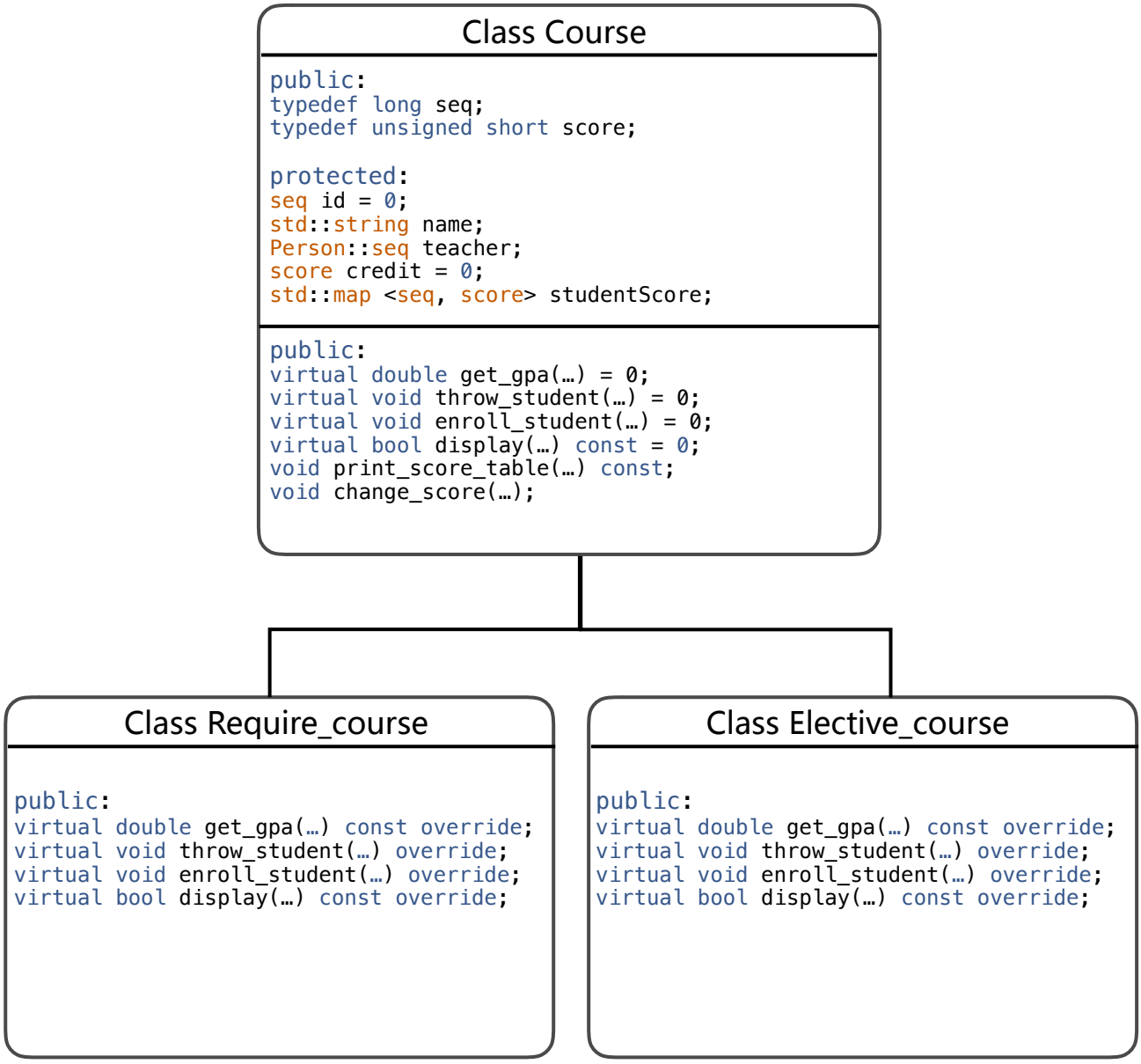


图3 课程类的继承关系

3. 系统类设计

逻辑上，系统类存储有课程数据地址与课程号、用户数据地址与用户号的映射关系，同时也是整个软件中存储系统的基础。考虑到其在整个程序中的唯一性与重要性，我是用了设计模式中最为常用的设计模式：单例模式。并与C++11最新的智能指针进行实现。

在网络上暂时没有智能指针实现单例模式的，这一点也可以是我算法的创新之一。通过使用 UNIQUE_PTR 指针，我们既可以保证指针所指向的东西在全局只能有唯一实例，也避免了由于程序员忘记删除指针而造成的内存泄漏。具体代码如下：

```
class Result_system
{
private:
    Result_system();
    static std::unique_ptr<Result_system> m_instance;

public:
    ~Result_system();
    static Result_system& get_instance()      //must be reference
    {
        if (m_instance == nullptr)
            m_instance.reset(new Result_system);
        return *m_instance;
    }

public:
    Person_ptr get_person(const Person::seq &);
    Course_ptr get_course(const Course::seq &);
    void print_available_course(const Student&, std::ostream &);
private:
    std::map <Person::seq, Person_ptr> num_to_person;
    std::map <Course::seq, Course_ptr> num_to_course;
};
```

图4 系统类的声明

三、实现方法

1. 绩点与成绩的存储

course 里面储存的是每个人的成绩而非绩点，是考虑到老师需要修改成绩的时候的方便。如果直接存储绩点，一个人的成绩改了，则所有人的绩点需要根据原来的成绩全部改变，实现复杂且不必要。

2. ID的存储

A. **主关键字使用size_t类型**——即某一系统最大所能存储的数组大小——来存储，在64位系统中支持存储19位学号/教工号。同时也保证了在不同平台上软件系统的自适应。

B. **使用 std::map来存储主关键字与其对应对象指针的映射。**使得vector需要O(n)的查询操作减少为O(nlogn)，提高了算法的效率。

3. Person类 & Course 类

A. **使用虚函数和protected标签的成员以实现继承。**使用 protected 的成员变量保证了其子类拥有父类成员变量的访问与修改权限。

B. **简化了数据结构的逻辑复杂度。**在存储老师/学生/课程的 System 类中，只保存了Person/Course的两个map（而不是老师、学生、必修课、选修课四个），具体子类与子类之间的不同通过重载虚函数可以实现；当需要调用某一子类独有的成员函数时，可以使用指针类型转换 dynamic_pointer_cast。

4. Result_system类的智能指针的单例模式实现

A. 使用 unique_ptr 指向全局唯一的静态私有成员，保证全局只有一个实例化的system类。

B. 使用 unique_ptr 有两个好处：

a）**保证了全局唯一**

b）**在主函数结束的时候会自动调用析构函数。**而如果用C风格的指针，需要借助另写一个Garbo类实现自动析构。自动析构的好处在于避免了因为程序员忘记手动析构而造成的内存泄漏。

c）get_instance()返回的是引用，保证了在结束子函数的时候System不被销毁。

5. 使用C++11的新特性

A. **使用 constexpr。**节省内存&翻译过程会检查类型错误；

B. **合理地使用inline。**通过对小函数使用inline关键字可以减小函数调用的开销，提高程序的效率；同时也没有对较大的成员函数滥用inline，其将导致编译出的代码过于冗杂和庞大。

C. **使用Lambda表达式灵活的进行模板算法调用。**在第一版中我是用了诸如 accumulate(), copy() 等标准算法库中常用的泛型算法

D. **使用智能指针。**避免了因为忘记 delete 函数中的局部指针变量而导致的内存泄漏的同时，也保证了代码的简洁。

E. **使用 stringstream 类进行单行文件的读入**，简洁有效。

F. **类内 typedef，保证不同类内的数据类型的可读性与有效性（不同的类 index_size 不一样）。**这种做法保证了当不同对象所申请的数组的下标范围不同时，能够自适应每个对象的特征。

G. **在涉及输入输出的函数中需要传参流指针。**这一点在未来扩展功能的时候非常方便。

第二部分：高级版的设计

一、设计要求

在版本一的基础上，新增以下功能：

a）实现所有数据的文件存取功能。

b）实现排序功能。

c）增加输入异常处理

二、实现思路

I. 文件存取

考虑两点事实：

a）老师、学生、选修课、必修课需要存储到不同的文件下。

b）所有类的实例对象在程序结束时候需要调用各自析构函数释放内存。

于是我考虑在基类设一个析构函数的结构，其子类去重写对应的析构函数。然后在析构函数中，实现不同类存储到不同文件下的功能。以Student类为例，其对应的析构函数代码为：

```
Student::~Student()
{
    using std::endl;

    /*store the data */
    static std::ofstream out_file("student.txt");
    out_file << id << ' ' << password << ' ' << name << ' '
             << college << ' ' << className << endl;
    copy(course.begin(), course.end(),
             std::ostream_iterator<int>(out_file, " "));
    out_file << endl;

    //destory vector
    course.clear();
}
```

图5. STUDENT类的析构函数

具体析构过程如下：在程序结束运行的时候，Result_system的静态成员变量m_instance被释放。由于它是一个智能指针，将自动调用所指向对象的析构函数。在系统类的析构函数中，将会分别释放所有用户、课程的对象，在释放内存的同时，数据也就被自动的保存到了文件中。

这样做的好处在于既保证了简洁，同时整个过程不需要在显式地调用任何的函数，彻底避免了内存泄漏的问题。

II. 成绩排序

考虑到对成绩排序的两种情况有所不同，我没有用虚函数重载的方式实现成绩排序，而是分别内置在 Student::display_info()、Course::print_score_table() 两个成员函数中，通过函数所传的实参不同实现以降序/升序输出，下面以学生角度为例，以下是实现代码（省略与初级班相同之处）：

代码中使用了Lambda表达式，节省了函数调用的时间，同时也保证了代码的简洁。“老师”的代码实现类似，此处就不再赘述。

```cpp
const Person& Student::display_info(std::ostream &os, const Score_mode
&mode)
{
/* output the basic info */

Result_system &system = Result_system::get_instance();

if (mode == INCREASE_BY_SCORE)
        sort(course.begin(), course.end(), [&system, this](const seq &a,
                const seq &b)
             { return system.get_course(a)->get_score(this->id) <
                system.get_course(b)->get_score(this->id); });
    else
        sort(course.begin(), course.end(), [&system, this](const seq &a,
                const seq &b)
             { return system.get_course(b)->get_score(this->id) >
                system.get_course(a)->get_score(this->id); });

/* output the score table */
}
```

图6 实现按成绩排序

三、代码重构

在做第二部分的时候，我决定放弃图形化，就把版本一中丑陋的main.cpp重构了一下，把与用户交互的子函数封装进了类 User_interface，所以版本二的main.cpp 如下图所示：

```cpp
//
//  main.cpp
//  Results Management System
//
//  Created by YeWenting.

#include "client.hpp"

int main(int argc, const char *
      argv[])
{
    User_interface window;

    window.show();
    return 0;
}
```

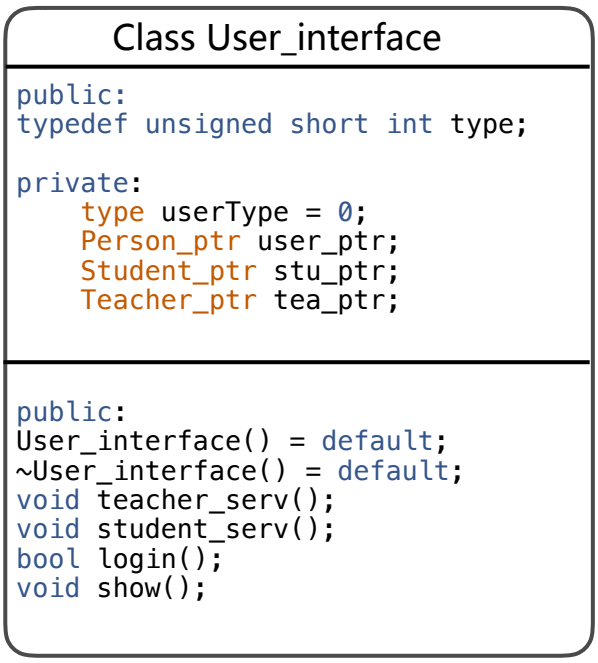


图7 重构后的USER-INTERFACE类

第三部分：高级版的设计

一、设计要求

在版本二的基础上，该版本增加了一下功能：

a）基于 Socket API 为软件增加了网络访问功能，即实现 C/S 模型。

b）基于 Select() 函数，增加多用户访问和操作结果实时同步的功能。

c）提升安全性，保障服务器端数据的安全。

重构了以下代码：

a）简化了try-catch语句

b）借鉴 MVC 模型，分离了用于用户交互的子程序与数据处理的子程序。

二、实现思路

粗略一想，总体而言实现网络功能有一下两种方式：

1. 服务器端将数据传给用户，用户在本地可以直接调用 display 函数与用户交互。
2. 服务器端不将数据传给用户，而只把数据的处理结果发给用户。

第一种方法易于实现，只需要服务器将对应用户的类传给用户，用户在本地可以实现选课、修改成绩等功能，然后再将数据回传给服务器端。但是这种方法有两个严重问题在于：

1）**不易于实现多用户操作结果的实时同步**

2）**无法保证用户在本地操作的合法性。**试想如果不怀好意的用户将服务器发来的数据肆意篡改以后，再回传给服务器端，就会造成难以想象的后果。

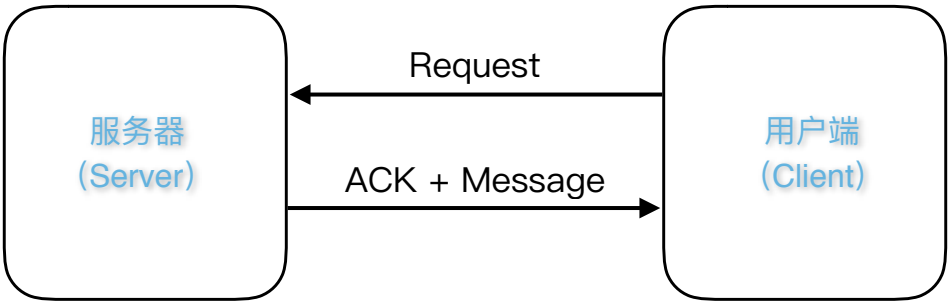


图8 本程序采用的 C/S 模型

本人采用了第二种方法。具体实现方法是：用户端接收到客户的一个请求，服务器端接受请求、在远端处理数据、将处理结果（一串字符串）回传给用户。ack为-1表示用户身份验证有误或请求有误，message为错误原因；当 ack 为 1时，message为对应的处理结果。这样做的好处在于：

a） **自然地实现多用户。**由于数据的处理过程发生在服务器端，处理完某一请求后的处理结果显然就自动保存了下来。

b） **保障了数据的安全。**在客户端我们可以限制不同种类用户的请求。在服务器端，对于每一个请求我们检查拥有该端口的用户是否具有请求的对应权限，如果没有的话则不予受理该请求。

在版本二中，我在所有的输入输出函数都留有 std::ostream 的传参，通过使用 std::stringstream 可以很方便的将输出流导入到一个字符串中。减少了对原来代码的的修改。

三、实现方法

1. Socket 编程

在创建 Socket 的子函数中，我复用了《深入理解计算机系统》一书中 Lab 的代码，该代码提供了一些很实用的特性，例如：

```
/*
 * open_listenfd - open and return a listening socket on port
 *     Returns -1 and sets errno on Unix error.
 */
/* $begin open_listenfd */
int open_listenfd(int port)
{
    int listenfd, optval=1;
    struct sockaddr_in serveraddr;

    /* Create a socket descriptor */
    if ((listenfd = socket(AF_INET, SOCK_STREAM, 0)) < 0)
     return -1;

    /* Eliminates "Address already in use" error from bind. */
    if (setsockopt(listenfd, SOL_SOCKET, SO_REUSEADDR,
         (const void *)&optval , sizeof(int)) < 0)
     return -1;

    /* Listenfd will be an endpoint for all requests to port
       on any IP address for this host */
    bzero((char *) &serveraddr, sizeof(serveraddr));
    serveraddr.sin_family = AF_INET;
    serveraddr.sin_addr.s_addr = htonl(INADDR_ANY);
    serveraddr.sin_port = htons((unsigned short)port);
    if (bind(listenfd, (SA *)&serveraddr, sizeof(serveraddr)) < 0)
        return -1;

    /* Make it a listening socket ready to accept connection requests */
    if (listen(listenfd, LISTENQ) < 0)
        return -1;
    return listenfd;
}
/* $end open_listenfd */
```

图9 实现创建监听SOCKET

以上是经过封装的创建一个监听端口的 socket 的子程序。这段代码考虑了不同阶段可能出现的函数调用失败，如果失败的话将会返回-1，并将错误信息记录在 errno 中，这一点对于调试错误非常有用；接下来将 socket 设置为可以重用地址和端口，这一点避免了 bind 错误；然后是处理监听的端口号，而端口号是作为传参传进来的，使得函数有较强的的灵活性。

除此之外，程序里还是用了封装好的读入写出函数，再此就不再赘述。

接下来我们考虑如何实现多用户。socket 网络编程中实现有两种主流的方法，第一种是用 fork()多线程地来处理不同用户的请求，第二种是使用 select() 函数每次取可读的 socket 进行处理。由于多线程的处理方法处理效率低，在业界一般较少使用，所以我使用了后者实现多用户，核心代码如下：

```
void Server::run()
{

    while (1)
    {
        memcpy(&rfds, &fds, sizeof(fds));
        if (Select(maxfd, &rfds, 0, 0, NULL) <= 0) break;

        /* if a new client come */
        if (FD_ISSET(listenSock, &rfds))
        {
            dataSock = Accept(listenSock, 0, 0);
            FD_SET(dataSock, &fds);
            if (maxfd <= dataSock) maxfd = dataSock + 1;
            client[dataSock].status = NO_LOGIN;
        }

        /* Process the request from clients */
        for (int fd = 0; fd < maxfd; fd++)
        {
            if (fd !=listenSock && FD_ISSET(fd, &rfds))
            {
                if (isclosed(fd))
                {
                    Close(fd);
                    FD_CLR(fd, &fds);
                    client[fd].status = NO_LOGIN;
                }
                else receive_data(fd);
            }
        }
    }
    Close(listenSock);
    Close(dataSock);
}
```

图10 实现多用户处理

服务器首先查看是否有socket 可读，“Select(maxfd, &rfds, 0, 0, NULL)”中第五个参数 NULL 代表了当没有 socket 可读时，进程被挂起直到有 socket 可读为止。然后判断如果监听接口 socket 可读则说明来了一个新用户，则创建一个未登录的用户；如果是传输数据（Request）的 socket 可读，则执行对应处理请求的子程序。

2. 服务器端日志文件

通过使用 recvfrom()函数可以获得客户端的 IP地址与端口号，基于这个函数我在服务器端实现了一个简单的日志记录功能，能够同时输出在命令行和同目录下的./server.log 中，效果如图9：

日志函数的实现可以方便发现不良的访问请求，也能够记录不同用户的活动。

```
YeWentings-MacBook-Pro:Debug YeWenting$ ./server 12345
Results management system have started...
9s Recv a request form 1.0.0.0:27655
Request for: LOGIN

9s Recv a request form 1.0.0.0:27655
Request for: GET_INFO

13s Recv a request form 1.0.0.0:27655
Request for: PRINT_ELECTIVE_COURSE
```

图11 日志记录功能

3. 健全的权限管理

考虑到本系统是数据管理系统，安全性是关键。本程序实现了不同层次的安全性处理：

1. **用户需要使用用户名和密码来登录。**每一个账户都有“教师”与“学生”的标示，象征着用户的不同权限。
2. **客户端限制允许操作。**具体而言就是学生只能够查看选修课、选课、退课等操作，而教师只有查看学生成绩、修改成绩的权限。对于越权的操作将会报错。
3. **服务器检查用户的身份和请求类型。**服务器端收到一个请求的时候，首先检查该用户是否有使用该请求的对应权限。如果没有则否决这次请求并返回错误信息。

四、代码重构

```
void Student::enroll_course()
{
    while (true)
    {
        try
        {
            cout << "Please input the course ID" <<endl;
            cin >> courseNum;
            Course_ptr enrollCourse = system.get_course(courseNum);
            enrollCourse->enroll_student(get_id());
            course.push_back(enrollCourse->get_id());
            cout << "You attend the " << enrollCourse->get_name() << "
course successfully." << endl;
            break;
        }
        catch (std::invalid_argument err)
        {
            if (!process_error(err)) break;
        }
    }
}
```

图12 版本二中代码片段

图12展示了版本二中的一部分，其中有两点存在问题：1）用户交互与数据处理没有分离 2）每一个类似的子函数中都有重复且雷同的 try/catch 片段。在版本三中我重构了这部分的代码。

1. 用户交互与数据处理的解耦

实际上这个工作是必须的，因为在 C/S 模型中，服务器端负责数据处理，客户端负责与用户交互。这一部分也没有什么特别好讲的，就不浪费篇幅了...

2. try/catch 机制

之前，我将处理异常的代码块（即 catch）内嵌在每一个子程序内部，造成很较大一部分的代码冗余，在版本三中我将每一个子函数中的 catch 提出来，放在来最外面，代码如下：

```
/*
 process the data from client
*/
try
    {
        /* Check the authenrization */
        if (!authenrize(sock, req))
            throw std::invalid_argument("You have no right to do this.");

        /* Process diff requirement */
        switch (req)
        {
            case LOGIN:
                do_login(sock);
                break;
            case PRINT_SCORE_TABLE:
                do_check_score(sock);
                break;
            case ATTEND_COURSE:
                do_attend_course(sock);
                break;
            case CANCEL_COURSE:
                do_cancel_course(sock);
                break;
            case PRINT_ELECTIVE_COURSE:
                do_print_elective_course(sock);
                break;
            case MODIFY_SCORE:
                do_modify_score(sock);
                break;
            case GET_INFO:
                do_get_info(sock);
                break;
        }
    }
    catch (std::invalid_argument err)
    {
        send_error(sock, err.what());
    }
```

图12 重构后的请求处理代码块

在这里 send_error 能够发送一个 ack == -1 的消息，并将错误信息告知客户端。客户端对应代码类似。