实训报告书

实训题目：嵌入式系统开发及应用

系　（部）：智能装备学院
专业班级：电子信息科学与技术 17-2 班
姓　　名：张厚今
学　　号：201723010237
完成日期：2020 年 6 月 19 日

实习类型	课程实训	实训地点	学校
组 别		实习课题	飞机票网络售票模拟系统
实训人姓名	张厚今、孙硕、戚莘凯		
任务分配	张厚今：售票端　　孙硕：购票端　　戚莘凯：服务端		
指导教师	张清菊	实习日期	2020.06.08 至 2020.06.13
实训成绩			
指导教师评语			

指导教师签名：_____

_____年 ____ 月____日

目录

1 实训目的.....1
2 网络购票系统原理.....1
2.1 程序功能分析.....1
2.2 系统框架分析.....1
2.3 基础知识总结.....2
3 实训过程记录.....7
3.1 头文件介绍.....7
3.2 购买机票功能.....11
3.3 退出功能.....12
3.4 查询航班信息.....13
3.5 增加航班信息.....14
3.6 更新航班信息.....14
3.7 删除航班信息.....15
3.8 MySQL 数据库配置.....15
4 实现的效果.....18
4.1 登录界面及初始化.....18
4.2 查询特定航班功能.....20
4.3 查询所有航班功能.....22
4.4 增加航班信息功能.....22
4.5 更新航班信息功能.....25
4.6 删除航班信息功能.....26
4.7 帮助信息和退出程序功能.....28
5 遇到的问题及解决方案.....30
6 实训总结.....31

1 实训目的

本次实训需要设计并实现一个飞机票的网络售票模拟系统，主要包括服务端设计、售票端设计以及购票端设计。通过编写该系统，熟悉并掌握 Linux 多线程编程、线程同步、网络套接字通信等内容的操作过程。

2 网络购票系统原理

2.1 程序功能分析

首先分析一下整个航班系统的功能需求。航班模拟系统由三部分组成，分别为服务端、售票端和购票端。为方便表述，以下将售票端和购票端统称为客户端。服务端通常是处理性能较好的计算机，适合进行数据处理操作，因此服务端承担了读取航班信息，接收购票端和售票端的数据请求以及对航班数据进行相应处理等功能。售票端面向系统管理员，具有较高的管理权限，可以实现对机票的查询、增加、更新和删除等操作。购票端面向普通用户，只有查询航班和购买机票的功能。

2.2 系统框架分析

2.2.1 服务端程序框架

由于会存在多个售票端和购票端同时连接服务器的情况，所以在服务器端要采用多线程编程技术。主线程将使用倾听套接字接收新的客户端连接，当新的客户端接入后，主线程将为新连接创建一个服务线程，并为新的服务线程分配一个线程缓冲区，将服务线程需要的信息保存在线程缓冲区中。这种多线程的通信模式如下图 2.1 所示。

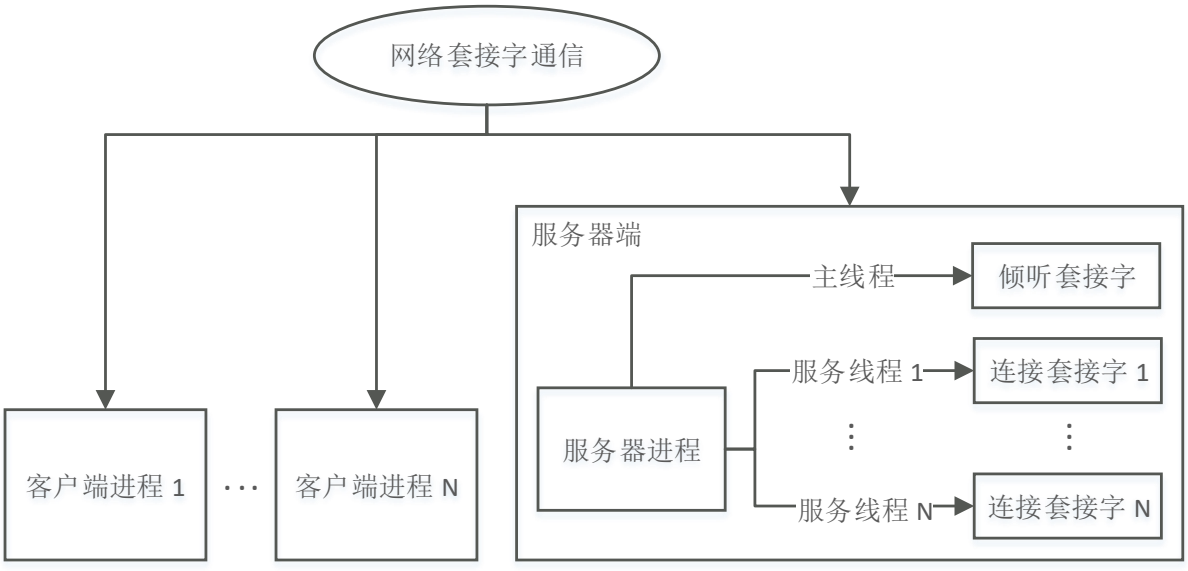

图 2.1 多线程通讯模式

主线程管理和分配线程缓冲区结构数组，每个服务线程对应自己的缓冲区，当主线程分配线程缓冲区时，需要检测 buffer_ status 变量的值。而服务线程在退出前，需要将线程缓冲区释放，也就是需要修改 buffer_ status 变量的值，所以主线程和服务线程间需要对 buffer_ status 变量进行互斥。本程序只使用了一个互斥锁对所有的 buffer_ status 变量进行互斥保护。这样虽然比较节省锁资源，但也会对多线程时的程序执行效果造成一定影响。

如果是多线程频繁访问互斥锁资源，就会降低程序的执行效率。我们现在假设只有少量的客户端程序，并且客户端程序不会频繁地登录和退出。这种情况下，进程对互斥锁的访问次数减少，互斥锁对程序造成的性能影响可以暂时忽略。

2.2.2 售票端和购票端框架

售票端和购票端的工作流程基本相同。客户端通过套接字与服务端进行数据通信。每当客户端要实现某个具体的功能时，都会首先向服务器发送相应的请求信息。服务器接收到该请求信息后，使用 MySQL 库函数与数据库进行数据通信，实现修改、更新、插入航班信息等功能。服务端处理完成后，通过套接字将处理结果和待返回的数据传送至客户端。这就是一种典型的客户端—服务器通信模型。

2.3 基础知识总结

2.3.1 套接字编程

Linux 中支持六种套接字，其中的数据流式套接字和数据报套接字最为常用。数据流套接字定义了一种可靠的面向连接的服务，实现了无差错无重复的顺序数据传输。数据报套接字定义了一种无连接的服务，数据通过相互独立的报文进行传输，是无序并且不保证可靠。

面向连接的套接字工作过程是，服务器首先启动，通过调用 socket 函数建立一个套接字，然后调用 bind 将该套接字和本地网络地址联系在一起。再调用 listen 使套接字做好侦听的准备，并规定它的请求队列的长度，之后就调用 accept 来接收连接。客户端在建立套接字后就可调用 connect 和服务器建立连接。连接一旦建立， 客户机和服务器之间就可以通过调用 read 和 write 来发送和接收效据。最后当数据传送结束后，双方调用 close 关闭套接字即可。

（1）socket 函数

函数 socket 创建一个套接字描述符，其定义如下图 2.2 所示。

```
#include <sys/types.h>
#include <sys/socket.h>
int socket(int domain, int type, int protocol);
```

图 2.2 函数声明

参数 domain 指定要创建的套接字的协议族；参数 type 指定套接字类型；参数 protocol 指定使用哪种协议。函数 socket 成功执行时，返回一个正整数用来标识这个套接字，称为套接字描述符，否则失败返回-1。

（2）connet 函数

函数 connect 用来与服务器建立连接，其定义如下图 2.3 所示。

```
#include <sys/types.h>
#include <sys/socket.h>
int connect(int sockfd, struct sockaddr *servaddr, int addrlen);
```

图 2.3 函数声明

参数 sockfd 是函数 socket 返回的套接字描述符；参数 servaddr 指定远程服务器的套接字地址，包括服务器的 IP 地址和端口号；参数 addrlen 指定这个套接字地址的长度。函数 connect 成功执行时返回 0，否则失败返回-1。

（3）bind 函数

函数 bind 将本地地址与套接字绑定在一起，其定义如下图 2.4 所示。

```
#include <sys/types.h>
#include <sys/socket.h>
int bind(int sockfd, struct sockaddr *myaddr, int addrlen);
```

图 2.4 函数声明

参数 sockfd 是函数 socket 返回的套接字描述符；参数 myaddr 是本地地址；参数 addrlen 是套接字地址结构的长度。函数 bind 成功执行时返回 0，否则失败返回-1。服务器和客户机都可以调用函数 bind 来绑定套接字地址，但一般是服务器调用函数 bind 来绑定自己的公认端口号。这里需要注意，一般有以下几种绑定组合方式，如下表 1-1 所示。

表 1-1 绑定方式

程序类型	IP 地址	端口号	含义
服务器	INADDR_ANY	非零值	指定服务器的公认端口号
服务器	本地 IP 地址	非零值	指定服务器 IP 地址和公认端口号
客户机	INADDR_ANY	非零值	指定客户机的连接端口号
客户机	本地 IP 地址	非零值	指定客户机的 IP 地址和端口号
客户机	本地 IP 地址	零	指定客户机的 IP 地址

一般我们只对服务器进行地址绑定操作，并且其端口号一般是选择 INADDR_ANY。这样表示服务器愿意接收来自任何网络设备接口的客户端连接。

（4）listen 函数

函数 listen 将一个套接字转换为倾听套接字(listening socket)，定义如下图 2.5 所示。

```
#include <sys/socket.h>
int listen(int sockfd, int backlog);
```

图 2.5 函数声明

参数 sockfd 指定要转换的套接字描述符；参数 backlog 设置请求队列的最大长度。函数 listen 成功执行时返回 0，否则失败返回-1。该函数只需要在服务器程序中调用，服务器需要调用 1isten 函数将套接字转换成倾听套接字，以便接收客户机请求。同时该函数可以设置 TCP 连接的最大请求队列长度。

（5）accept 函数

函数 accept 从倾听套接字的完成连接队列中接收一个连接。如果完成连接队列为空，那么这个进程休眠，其定义如下图 2.6 所示。

```
#include <sys/socket.h>
int accept(int sockfd, struct sockaddr *addr, int *addrlen);
```

图 2.6 函数声明

参数 sockfd 指定套接字描述符；参数 dddr 为指向一个 Internet 套接字地址结构的指针；参数 addrlen 为指向一个整型变量的指针。函数 accept 成功执行时返回 3 个结果：函数返回值为一个新的套接字描述符，标识这个接收的连接；参数 addr 指向的结构变量中存储客户机地址；参数 addrlen 指向的整型变量中存储客户机地址的长度。如果对客户机的地址和长度都不感兴趣，可以将参数 addr 和 addrlen 设置为 NULL。函数 accept 执行失败时返回-1。

（6）close 函数

函数 close 关闭一个套接字描述符，套接字描述符的 close 操作与文件描述符的 close 操作类似，其定义如下图 2.7 所示。

```
#include <unistd.h>
int close(int sockfd);
```

图 2.7 函数声明

参数 sockfd 指定要关闭的套接字描述符。函数成功执行时返回 0，否则失败返回-1。

（7）read 和 write 函数

函数 read 和 write 从套接字读和写数据，其定义如下图 2.8 所示。

```
#include <unistd.h>
ssize_t read(int fd, void *buf, size_t count);
ssize_t write(int fd, const void *buf, size_t count);
```

图 2.8 函数声明

参数 fd 指定读写操作的套接字描述符；函数 read 的参数 buf 指定接收数据缓冲区，函数 write 的参数 buf 指定发送数据缓冲区；参数 count 指定接收或发送的数据量大小。函数 read 成功执行时，返回读到的数据量大小，否则失败返回-1。函数 write 成功执行时，返回写入的数据量大小，否则失败返回-1。

2.3.2 线程操作

（1）线程概念

进程是一个复合的实体，可以分为线程的集合和资源的集合两个部分。线程是一个动态的对象，它表示进程中的一个控制点，并且执行一系列的指令。资源包括地址空间、打开的文件、用户凭证和配额等，这些资源被进程中的所有线程所共享。此外，每一个线程有它自己的私有对象，如程序计数器、堆栈和寄存器的值。传统的 UNIX 进程有一个单独的控制线程，在多线程系统中进行了扩展，允许在一个进程中有多个控制线程。

（2）线程创建和终止

如果某个线程可在进程执行期间的任意时刻被创建，并且线程的数量事先没有必要指定，这样的线程称为动态线程。在 POSIX 中，线程是用 pthread_create 函数动态地创建的。pthread_create 能创建线程，并将它放入就绪队列。该函数的定义如下图 2.9 所示。

```
#include <pthread.h>
int pthread_create(pthread_t *thread, pthread_attr_t *attr, \
void *(*start_routine)(void *), void *arg);
```

图 2.9 函数声明

pthread_create 函数会创建一个线程， 这个线程与创建它的线程同步执行。新创建的线程将执行函数 start_routine，这个函数的参数由指针 arg 指定。该线程可以通过 pthread_exit 来终止，或者当函数 start_ routine 返回时自然终止。参数 attr 用来指定新线程的属性，可以为 NULL 表示默认的属性，即使用最小的堆栈空间和通常的调度策略等。要结束一个线程，需要调用函数 pthread_exit。它的原型如下图 2.10 所示。此函数用于结束一个线程。它将调用清除处理函数结束当前线程，函数返回值为 retval。

```
#include <pthread.h>
int pthread_exit(void *retval);
```

图 2.10 函数声明

（3）线程挂起

可以使用 pthread_join 函数将某个线程挂起，其函数定义如下图 2.11 所示。

```
#include <pthread.h>
int pthread_join(pthread_t th, void **thread_return);
```

图 2.11 线程挂起

此函数用于挂起当前线程直至指定线程终止。参数 th 是一个线程标识符，用于指定需要等待其终止的线程。参数 thread_ return 用于存放其他线程的返回值。对于每一个可连接的线程都必须调用该函数一次。任何线程都不能对相同的线程调用此函数。

（4）pthread_mutex_init 函数

互斥锁初始化的函数定义如下图 2.12 所示。

```
#include <pthread.h>
int pthread_mutex_init(pthread_mutex_t *mutex, const pthread_mutexattr_t *attr);
```

图 2.12 函数声明

该函数用来初始化由参数 mutex 指向的互斥锁，这个互斥锁的属性由参数 attr 指定，或者通过指定 attr 为 NULL 而使用默认的属性。不会出现有多个线程同时初始化同一个互斥锁的情形，一个互斥锁在使用期间一定不会被重新初始化。如果 pthread_ mutex jinit 执行成功，则返回 0，并将新创建的互斥锁的 ID 值放到参数 mutex 中。如果执行失败，那么将返回一个错误编号。

（5）pthread_mutex_lock 函数

互斥锁锁定函数的定义如下图 2.13 所示。

```
#include <pthread.h>
int pthread_mutex_lock(pthread_mutex_t *mutex);
```

图 2.13 函数定义

用 pthread_mutex_lock 函数可以锁定由参数 mutex 指向的互斥锁。如果 mutex 已经被锁定，那么当前调用的线程将阻塞，直到互斥锁被其他线程释放(阻塞线程按照线程优先级等待)。当 pthread_mutex_lock 函数返回时，说明互斥锁已经被当前线程加锁完成。如果 pthread_ mutex_ lock 执行成功则返回 0，说明其他的值发生了错误。

（6）pthread_mutex_unlock 函数

互斥锁解锁函数的定义如下图 2.14 所示。

```
#include <pthread.h>
int pthread_mutex_unlock(pthread_mutex_t *mutex);
```

图 2.14 函数定义

用 pthread_mutex_unlock 函数给参数 mutex 指定的互斥锁解锁。互斥锁必须处于加锁状态而且调用本函数的线程必须是给互斥锁加锁的同一个线程才能给互斥锁解锁。如果有其他线程在等待互斥锁，那么由核心的调度程序决定哪个线程将获得互斥锁并脱离阻塞状态。如果 pthread_mutex_unlock 函数执行成功则返回 0，其他值意味着错误。

3 实训过程记录

我们小组参考了课本例题的代码，将例题代码修改后直接添加到了 QT 中。我负责的是售票端部分，下面对售票端的制作过程进行简单介绍。介绍过程中，我会尽量规避 QT 的前端设计，主要讲述 Linux 的后台逻辑功能。售票端的 QT 代码结构如下图 3.1 所示。

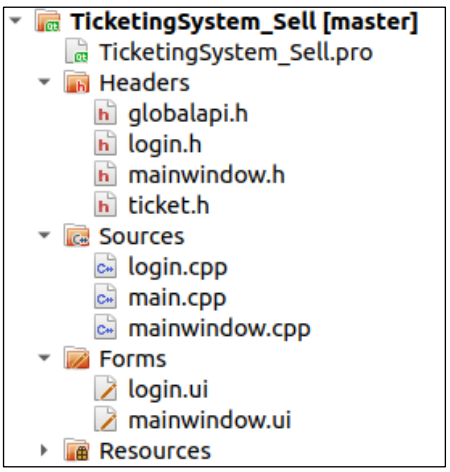

图 3.1 工程目录结构

3.1 头文件介绍

3.1.1 global.h 文件

global.h 主要包含了导入库文件和宏定义的代码。其中比较重要的是下面的几个宏定义。

```
/* 客户端使用的消息代码含义 */
#define    DISCONNECT          0
#define    BUY_TICKET          1
#define    INQUIRE_ONE         2
#define    INQUIRE_ALL         3
#define    ADD_TICKET          4
#define    UPDATE_TICKET       5
#define    DELETE_TICKET       6
```

上图中的七个宏定义代表了其中不同的消息类型，售票端将这些消息类型存储到消息结构体并发送至服务端。服务端接收到消息结构体后，通过判断消息结构体的这七种消息类型，进而执行不同的操作。包括“断开连接”、“购买机票”、“查询单个航班”、“查询所有航班”、“增加航班信息”、“更新航班信息”、“删除航班信息”这七种功能。

另外，服务端接收到售票端的消息并执行相应操作后，需要将执行结果返回给购票端。这种消息类型的返回也是通过宏定义实现的，如下所示。

```
/*服务器端使用的消息代码含义*/
#define   BUY_SUCCEED          255
#define   BUY_FAILED           256
#define   INQUIRE_SUCCEED      257
#define   UNKNOWN_CODE         258
```

上面的四个宏定义分别代表服务端“购买机票操作成功”、“购买机票操作失败”、“查询航班成功”以及“未知操作”。另外比较重要的是售票端向服务器发送的消息结构体类型，该结构体如下所示。

```
/* 服务器与客户端使用的消息结构定义，用来向服务器请求不同类型的信息 */
struct stMessage {
   unsigned int msg_type;              // 用来向服务器请求不同类型的信息
   unsigned int flight_ID;             // 航班号
   unsigned int ticket_num;            // 机票张数
   unsigned int ticket_total_price;    // 机票价钱
} message;
```

结构体中包含了消息类型、航班号、票价、票数、票价这四项成员变量，用来实现向服务器发送不同的请求消息。另外在 global.h 中，定义了 init_message 函数，用来实现消息结构体的数据初始化。函数定义如下所示。

```
/* 将消息数据类型进行初始化 */
void init_message(){
   message.msg_type=INITIAL_VALUE;
   message.flight_ID=0;
   message.ticket_num=0;
   message.ticket_total_price=0;
}
```

上述函数将消息结构体中的数据类型初始化为初始值，将机票价格、票数、航班价格初始化为 0。

3.1.2 ticket.h 文件

ticket.h 函数定义了机票信息的结构体、机票信息更新函数以及航班数量更新函数。机票信息结构体如下所示。

```
/* 机票信息的结构体 */
typedef struct ticket_struct_t {
   unsigned int flight_ID;             // 航班号
   unsigned int ticket_num;            // 机票剩余票数
   unsigned int ticket_price;          // 票价
   // 多个线程操作时，必须对机票的剩余数量进行保护
   // 应当对每一个 ticket_num 使用不同的互斥锁
   // 否则将对线程间并行性有较大影响
   pthread_mutex_t      ticket_mutex;
```

```
} ticket_struct;
ticket_struct ticket_list[FLIGHT_NUM];
```

该结构体与之前定义的消息结构体不同的是，消息结构体是用来向服务器发送操作请求的，而这个结构体是用来在售票端本身记录航班信息的。它记录了当前航班的航班号、票数票价等信息，同时为了与服务端的结构体对应，结构体中还加入了线程互斥锁这个成员变量。

另外 ticket.h 文件中还定义了 read_ticket_list 函数，该函数用来实现读取数据库中的航班数据，其关键代码如下所示。

```
mysql_init(&mysql);
mysql_real_connect(&mysql, "localhost", "zhj", "666588", "linux", 0, NULL, 0);
// 调用 mysql_store_result 之前必须检索数据库
mysql_query(&mysql, "select * from tickets");
result = mysql_store_result(&mysql);            // 将查询的全部结果读取到客户端
numRows = mysql_num_rows(result);               // 统计结果集的行数
if(result){
   for(i=0;i<numRows;i++){
      if((row = mysql_fetch_row(result)) != NULL){
          ticket_list[i].flight_ID = atoi(row[0]);
          ticket_list[i].ticket_num = atoi(row[1]);
          ticket_list[i].ticket_price = atoi(row[2]);
      }
    }
 }
mysql_free_result(result);                      // 释放 result 空间，避免内存泄漏
mysql_close(&mysql);
```

该函数首先初始化数据库对象，设置数据库配置信息并连接数据库。使用 mysql_store_result 函数读取数据库中的内容，由 mysql_num_rows 函数获取数据库中的表单行数，也就是当前航班的班次总数。之后使用 mysql_fetch_row 函数用来获取每行的数据，将获取到的数据分配到 ticket_list 数组中。

在使用数据库连接函数 mysql_real_connect 时，需要事先创建好数据库“linux”。在使用数据库执行函数 mysql_query 时，也要事先创建好数据表“tickets”。我们使用 Linux C 代码对 MySQL 数据库进行了初始化，部分代码如下所示。

```
MYSQL mysql;
mysql_init(&mysql);
mysql_real_connect(&mysql, "localhost", "zhj", "666588", "linux", 0, NULL, 0);
mysql_query(&mysql, "create table tickets(flight_ID int AUTO_INCREMENT PRIMARY KEY, ticket_num int, ticket_price int)");
mysql_query(&mysql, "insert into tickets values(1, 100, 300),(2, 100, 300),(3, 100, 300);
mysql_query(&mysql, "select * from tickets");
mysql_close(&mysql);
```

在 ticket.h 文件中，还定义了 update_ticket_number 函数。该函数用来更新当前的航班数量，主要是使用到了 mysql_num_rows 函数。其代码如下所示。

```
void update_ticket_number(void){
   MYSQL mysql;
   MYSQL_RES * result;
   mysql_init(&mysql);
   mysql_real_connect(&mysql, "localhost", "zhj", "666588", "linux", 0, NULL, 0);
   mysql_query(&mysql, "select * from tickets");
   result = mysql_store_result(&mysql);          // 将查询的全部结果读取到客户端
   numRows = mysql_num_rows(result);          // 统计结果集的行数
   mysql_free_result(result);
   mysql_close(&mysql);
}
```

在 QT 工程中，除了 global.h 和 ticket.h 头文件外，还包含了其他的头文件。这些头文件与 Linux 功能函数的关系不大，主要是用于 QT 的界面设计，因此就不再详细描述了。在 QT 的源文件中，与 Linux 功能函数相关的是 mainwindow.cpp 文件。该文件定义了 QT 界面的各种槽函数，也就是实现了各种 C 语言的底层逻辑功能。

on_action_connect_triggered 函数用来处理 QT 中“连接”按钮被触发时的事件，其主要代码如下所示。

```
   if(!isconnected){
      /* 创建套接字 */
      socket_fd=socket(AF_INET,SOCK_STREAM,0);
      if(socket_fd<0) {
         sprintf(msg,"创建套接字出错！ \n");
         display_info(msg);
         return;
      }
      /* 设置接收、发送超时值 */
      struct timeval time_out;
      time_out.tv_sec=5;
      time_out.tv_usec=0;
      setsockopt(socket_fd,SOL_SOCKET,SO_RCVTIMEO,&time_out,sizeof(time_out));
      /* 填写服务器的地址信息 */
      server.sin_family=AF_INET;
      server.sin_addr.s_addr=inet_addr("127.0.0.1");    //htonl(INADDR_ANY);
      server.sin_port=htons(SERVER_PORT_NO);
      /* 连接服务器 */
      ret = ::connect(socket_fd,(struct sockaddr*)&server, sizeof(server));
      if(ret<0) {
         // 这里改了一下，添加了格式控制符%d
         sprintf(msg,"连接服务器出错！%d\n",SERVER_PORT_NO);
         display_info(msg);
         ::close(socket_fd);
         return;
```

```
    }
    /* 成功后输出提示信息 */
    sprintf(msg,"连接服务器成功！\n");
    display_info(msg);
    isconnected=true;
    enable_button(isconnected);
}
```

该函数首先创建套接字描述符，配置接收和发送的超时值，填写服务器套接字地址并进行连接。连接服务器成功后，在 QT 界面输出相关的提示信息。整个过程与 Linux C 语言中的代码流程没有太大区别，所用到的函数也是用了 Linux C 的函数。

on_action_disconnect_triggered 函数是 QT 中用来响应“断开连接”按钮的事件处理函数，该函数主要是调用了 Linux C 语言中的 close 函数，实现了断开服务器的操作。其主要到代码如下所示。

```
if(isconnected) {
    ::close(socket_fd);
    sprintf(msg,"断开连接成功！\n");
    display_info(msg);
    isconnected=false;
}
```

3.2 购买机票功能

购买机票功能的事件处理函数为 on_action_buyticket_triggered，在 QT 中需要先弹出一个对话框以获取具体的购票信息，获取信息后，需要及时判断一下用户输入的购票信息是否有效，即是否在合理的航班区间。接下来可以使用 C 语言进行事件处理。首先把 message 结构体的 msg_type 成员变量设置为 BUY_TICKET，flight_ID 和 ticket_num 成员变量也都要进行赋值。

使用 send 函数向服务器发送 message 结构体数据，服务端接收到该数据后，通过判断 msg_type 的值来决定具体的操作。以 BUY_TICKET 为例，当服务器判断到 msg_type 的值为 BUY_TICKET 时，会在线程处理函数中对用户输入的购票信息和当前航班信息进行判断。如果符合数量要求，就会更新数据库中的航班数量，并向客户端发送回复消息。服务器回复的消息结构体中，msg_type 成员变量被设为 BUY_SUCCEED，表示购买机票成功。如果客户端向服务器发送的航班信息不符合当前航班要求，服务端就会直接向客户端发送 msg_type 成员变量被设为 BUY_FAILED 的结构体信息。客户端通过 recv 函数接收结构体数据，通过判断成员变量 msg_type 的值，可以知道服务器端的操作是否成功。

该函数的部分代码如下所示。

```
/* 购买机票 */
init_message();
message.msg_type=BUY_TICKET;
message.flight_ID=flight_ID;
message.ticket_num=ticket_num;
memcpy(send_buf,&message,sizeof(message));
int ret=send(socket_fd, send_buf,sizeof(message),0);
/* 发送出错 */
if(ret == -1) {
   display_info("发送失败！请重新发送！");
   return ;
}
ret = recv(socket_fd,recv_buf,sizeof(message),0);
if(ret==-1) {
   display_info("接收失败！请重新发送！");
   return ;
}
memcpy(&message,recv_buf,sizeof(message));
if(message.msg_type==BUY_SUCCEED){
   sprintf(msg, "购买成功！航班号：%d, 票数：%d, 总票价：%d\n",
message.flight_ID, message.ticket_num, message.ticket_total_price);
}
else{
   sprintf(msg,"购买失败！航班号：%d, 剩余票数：%d, 请求票数：%d\n",
message.flight_ID, message.ticket_num,ticket_num);
}
display_info(msg);
```

3.3 退出功能

退出售票端功能是在按下 QT 界面的“退出”按钮后被触发的，具体的事件处理函数是 on_action_exit_triggered 函数。其关键代码如下所示。

```
while(isconnected){
   ::close(socket_fd);
   sprintf(msg,"断开连接成功！\n");
   display_info(msg);
   isconnected=false;
}
display_info("即将关闭客户端");
close();
```

当按下“退出”按钮时，售票端会判断当前程序是否与服务器保持连接状态。如果此时售票端正在与服务器连接中，那么需要首先关闭连接的套接字并显示提示信息。关闭连接后，使用 close 函数关闭 QT 的窗口。

3.4 查询航班信息

3.4.1 查询单个航班功能

查询单个航班功能是在事件处理函数 on_action_inquireone_triggered 中，实现了对单次航班的信息查询。

售票端点击查询单次航班后，会在 QT 界面弹出一个提示框，要求用户输入待查寻的航班班次。售票端会首先调用 update_ticket_number 函数，读取数据库中的航班数量信息，然后通过遍历的方式判断用户输入的航班信息是否符合要求。如果用户输入的航班信息在数据库中存在，那么售票端会向服务器发送请求。其主要代码如下所示。

```
init_message();
message.msg_type=INQUIRE_ONE;
message.flight_ID=flight_ID;
memcpy(send_buf,&message,sizeof(message));
int ret=send(socket_fd, send_buf,sizeof(message),0);
/* 发送出错 */
if(ret==-1) {
    display_info("发送失败！请重新发送！");
    return ;
}
ret=recv(socket_fd,recv_buf,sizeof(message),0);
```

售票端会将消息结构体中的 msg_type 成员变量设置为 INQUIRE_ONE，表示请求单次航班信息。另外 flight_ID 成员也需要被赋值，表示具体要查询的航班号。之后售票端会调用 send 函数，向服务器发送请求。服务器接收到请求后，通过判断 msg_type 成员变量的值，可以进行相应的查询操作。服务端执行完毕后，会向售票端发送回复信息，售票端使用 recv 函数接收该信息。

3.4.2 查询所有航班功能

查询所有航班功能与查询单个航班功能的逻辑类似，当用户点击“查询所有航班”按钮时，售票端会将 msg_type 成员变量的值设置为 INQUIRE_ALL，表示查询所有航班信息。之后售票端会调用 send 函数，将结构体数据发送至服务器。服务器把处理结果返回给售票端，售票端通过 recv 函数接收信息。这里比较重要的是数据显示部分，其主要代码如下所示。

```
for (i=0;i<ret;i=i+sizeof(message)) {
   memcpy(&message,recv_buf+pos,sizeof(message));
   if(message.msg_type==INQUIRE_SUCCEED){
      sprintf(msg,"查询成功！航班号：%d, 剩余票数：%d, 票价：%d", \
      message.flight_ID,message.ticket_num, message.ticket_total_price);
   }
   else{
```

```
            sprintf(msg,"查询失败！航班号：%d, 剩余票数：未知",message.flight_ID);
        }
        display_info(msg);
        pos+=sizeof(message);
    }
```

在这个数据遍历的循环中，变量 i 和变量 pos 都作为了接收数据的偏移指针，以 sizeof(message)作为每组数据的数据量，分组进行数据输出。

3.5 增加航班信息

增加航班信息、更新航班信息、删除航班信息功能是售票端所特有的功能，首先来介绍增加航班信息的功能。

在 QT 程序中，增加航班信息的功能用到的事件处理函数为 on_action_add_triggered，当用户点击“增加航班信息”按钮时，会触发该函数。该函数首先会弹出一个提示框，要求管理员输入要增加的航班信息，包括航班号、票价和票数。之后售票端将 msg_type 成员变量设置为 ADD_TICKET，并将 flight_ID 等其他成员变量赋值。调用 send 函数向服务器发送请求信息。服务器接收到该请求后，会首先连接数据库，并使用插入语句向数据库中插入新的航班信息。服务线程的处理过程如下所示。

```
case ADD_TICKET:
    read_ticket_list();                    // 读取数据库机票信息
    mysql_init(&mysql);
    mysql_real_connect(&mysql, "localhost", "zhj", "666588", "linux", 0, NULL, 0);
    sprintf(sqlstr, "insert into tickets values(%d, %d, %d)", message.flight_ID, \
    message.ticket_num, message.ticket_total_price);
    mysql_query(&mysql,sqlstr);            // 执行更新语句
    mysql_close(&mysql);
break;
```

目前的代码逻辑中，因为服务器对数据库的操作成功率很高，所以当服务器处理完成后，没有再向售票端发送回复消息，售票端也不必再调用 recv 接收消息。

3.6 更新航班信息

更新航班信息与增加航班信息的代码逻辑基本相同，先在售票端弹出对话框要求用户输入待更新的航班信息，然后将该信息赋值到消息结构体中。消息结构体中的成员变量 msg_type 被设为 UPDATE_TICKET，调用 send 函数向服务器发送更新请求。

服务器判断到 UPDATE_TICKET 后，执行的代码与 ADD_TICKET 类似，只是将插入语句换成了更新语句。同样，服务器处理完成数据库的数据后，也不会向售票端发送回复消息。

3.7 删除航班信息

删除航班信息与增加、更新航班信息功能的代码逻辑类似，只是将服务器处理过程中调用 mysql_query 函数的地方，换成了数据库的删除语句。这里就不再赘述了。

3.8 MySQL 数据库配置

航班信息的存储我们使用了 MySQL 的数据库，下面简单介绍一下数据库的配置过程。为了方便地查看数据库中的内容，我们选择在 Ubuntu 中搭建 LAMP 环境，安装 Apache、MySQL 和 PHPMyAdmin 软件。环境搭建的过程就不在此介绍了，只介绍一下 MySQL 环境配置的过程。

首先我们为数据库创建一个新的用户，用户名设为 zhj，如下图 3.2 所示。

```
zhj@vmware:~$ sudo mysql -u root
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 45
Server version: 5.7.30-0ubuntu0.18.04.1 (Ubuntu)

Copyright (c) 2000, 2020, Oracle and/or its affiliates. All rights reserved.

Oracle is a registered trademark of Oracle Corporation and/or its
affiliates. Other names may be trademarks of their respective
owners.

Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.

mysql>
mysql> CREATE USER 'zhj'@'localhost' IDENTIFIED BY '666588';
Query OK, 0 rows affected (0.00 sec)

mysql>
mysql> GRANT ALL PRIVILEGES ON *.* TO 'zhj'@'localhost' WITH GRANT OPTION;
Query OK, 0 rows affected (0.00 sec)

mysql> FLUSH PRIVILEGES;
Query OK, 0 rows affected (0.01 sec)

mysql> quit
Bye
zhj@vmware:~$
```

图 3.2 创建用户

接下来需要手动创建一个数据库，将其命名为 linux，如下图 3.3 所示。以 root 用户登录进 MySQL，使用 create database linux 语句创建数据库。创建完成后，使用 show databases 语句查看当前所有的数据库。

```
zhj@vmware:~$ sudo mysql -u root
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 56
Server version: 5.7.30-0ubuntu0.18.04.1 (Ubuntu)

Copyright (c) 2000, 2020, Oracle and/or its affiliates. All rights reserved.

Oracle is a registered trademark of Oracle Corporation and/or its
affiliates. Other names may be trademarks of their respective
owners.

Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.

mysql> create database linux;
Query OK, 1 row affected (0.00 sec)

mysql> show databases;
+--------------------+
| Database           |
+--------------------+
| information_schema |
| linux              |
| mysql              |
| performance_schema |
| phpmyadmin         |
| sys                |
+--------------------+
6 rows in set (0.00 sec)

mysql>
```

图 3.3 创建数据库

按照课本上相关章节的内容，编写创建数据库表的 C 语言代码，部分代码如下图 3.4 所示。

```c
int main()
{
    int i;
    MYSQL mysql;
    MYSQL_RES * result;
    mysql_init(&mysql);
    mysql_real_connect(&mysql, "localhost", "zhj", "666588", "linux", 0, NULL, 0);
    mysql_query(&mysql, "create table tickets(flight_ID int AUTO_INCREMENT PRIMARY KEY, ticket_nu
    mysql_query(&mysql, "insert into tickets values(1, 100, 300),(2, 100, 300),(3, 100, 300),(4,
    mysql_query(&mysql, "select * from tickets");
    printf("--增加数据测试--\n");
    printResult(&mysql);
```

图 3.4 初始化数据表

在运行 C 语言数据库程序前，需要首先使用 sudo apt-get install libmysqlclient-dev 语句，安装与数据库相关的 C 语言库。安装完成后，编译运行程序。程序运行结果如下图 3.5 所示。

```
zhj@vmware:~/TicketingSystem/Data$ ls
images  mysql_setup.c
zhj@vmware:~/TicketingSystem/Data$ gcc mysql_setup.c -o mysql_setup -lmysqlclient
zhj@vmware:~/TicketingSystem/Data$ ls
images  mysql_setup  mysql_setup.c
zhj@vmware:~/TicketingSystem/Data$ ./mysql_setup
--增加数据测试--
flight_ID       ticket_num      ticket_price
1       100     300
2       100     300
3       100     300
4       100     300
5       100     300
6       100     300
7       100     300
8       100     300
9       100     300
10      100     300
zhj@vmware:~/TicketingSystem/Data$
```

图 3.5 数据表初始化

此时打开 PHPMyAdmin，可以看到数据表已经被成功初始化了，如下图 3.6 所示。

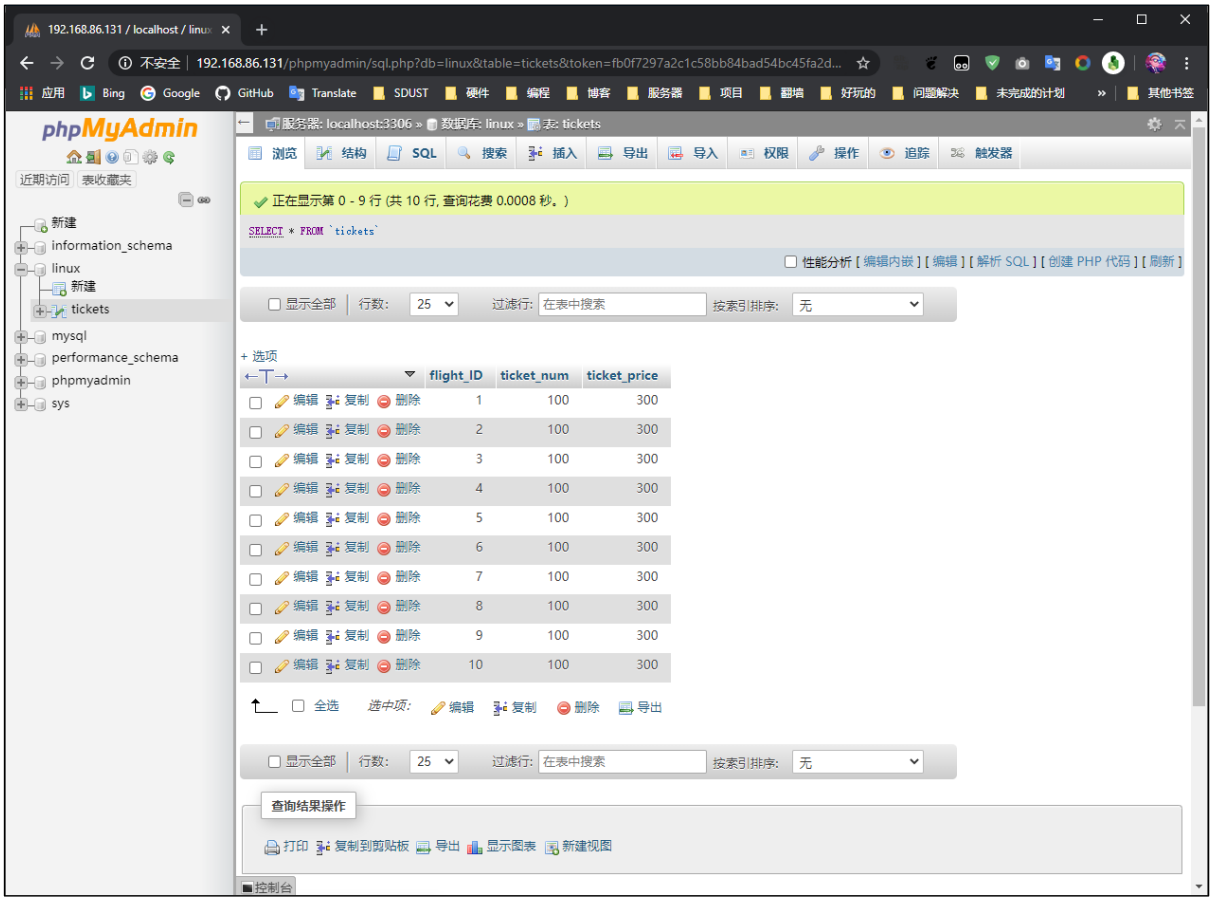

图 3.6 查看数据表

数据表被成功初始化后，接下来的操作就可以直接在 QT 中进行了。我们在 QT 中可以用 C 语言的数据库函数，对数据表中的内容进行增删改查，是非常方便的。

4 实现的效果

由于我们是移植的课本例程代码，将代码移入了 QT 中并进行的续修改，所以测试过程就可以直接在 QT 中进行了。代码编写完成后，运行售票端进行测试。下面仅介绍售票端实现的效果，服务端和购票端效果请查看附录中的测试视频链接。

4.1 登录界面及初始化

因为售票端的权限较高，所以我们为售票端设计了登录界面。启动售票端后，首先会出现管理员的登录界面，如下图 4.1 所示。

图 4.1 登录界面

如果输入错误的用户名或密码，则登录失败，售票端弹出提示信息。

图 4.2 提示信息

如果用户名和密码输入正确，则窗口跳转到售票界面。售票界面初始化时只能点击“连接服务器”和“退出”按钮。其界面如下图 4.3 所示。

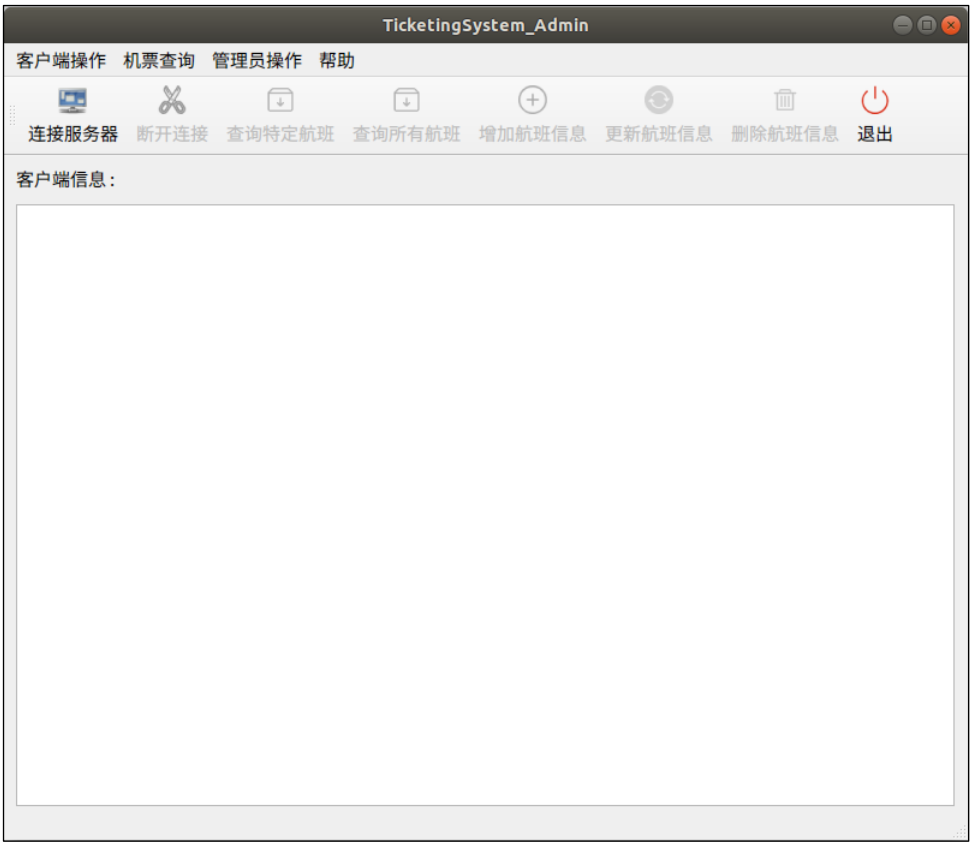

图 4.3 售票界面初始化

此时需要启动服务端程序，开启服务器才能在售票端成功连接服务器。

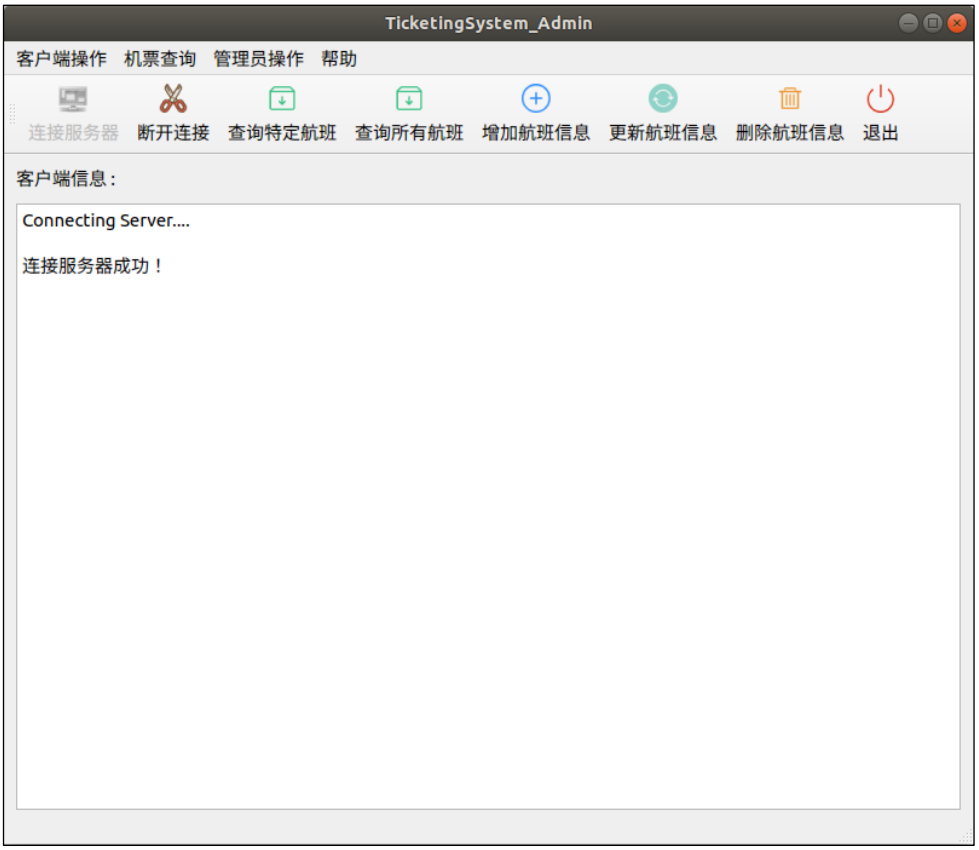

图 4.4 售票界面连接服务器

4.2 查询特定航班功能

用户点击“查询特定航班”按钮，会弹出提示窗口，要求用户输入待查寻的航班号信息，如下图 4.5 所示。用户在弹出的提示窗口中输入航班号，点击 OK 按钮即可进行航班查询。如果用户点击 Cancel 取消按钮，则取消当前的查询功能，返回到程序主界面。

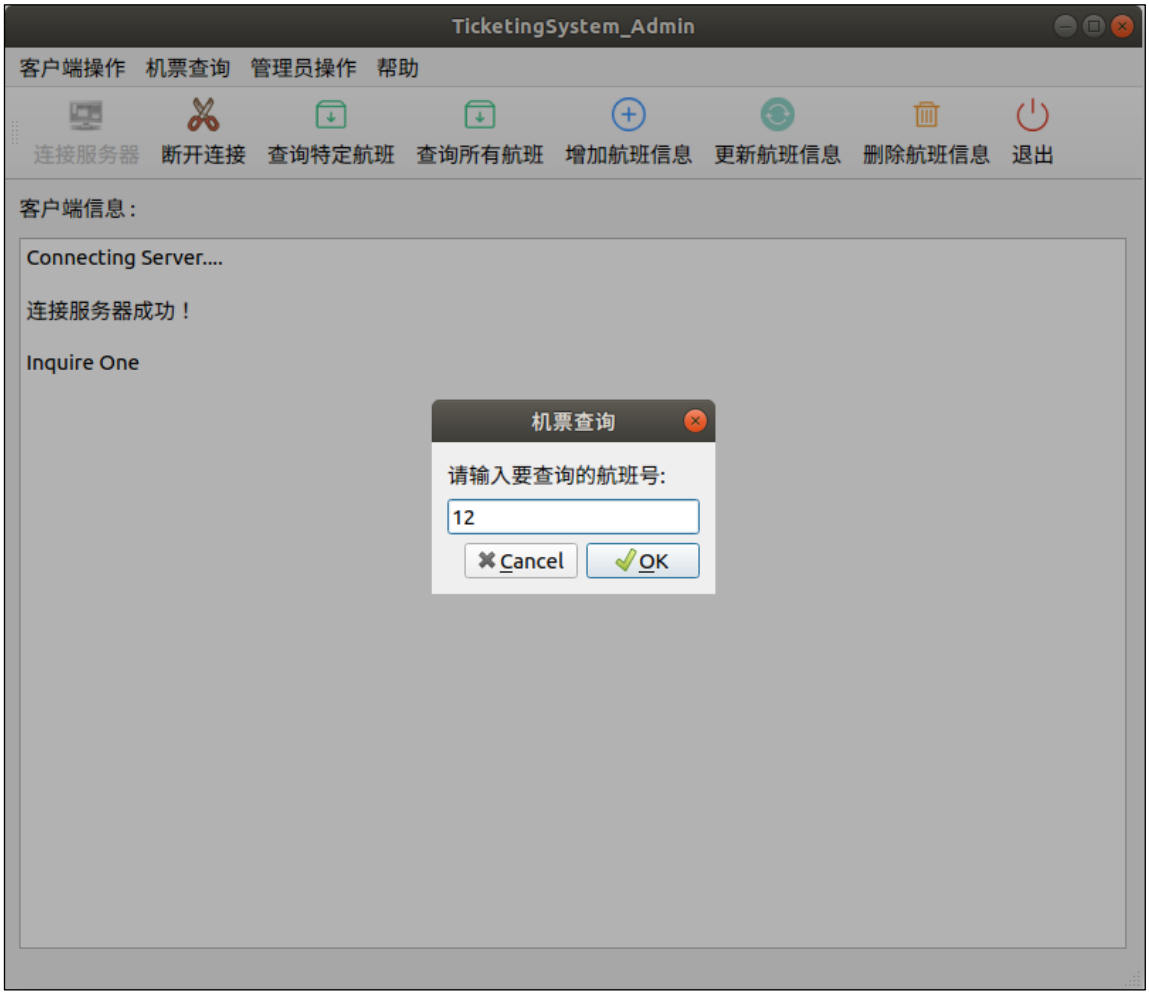

图 4.5 输入航班号

当该航班存在，即用户输入的航班信息无误时，售票端会显示航班信息。

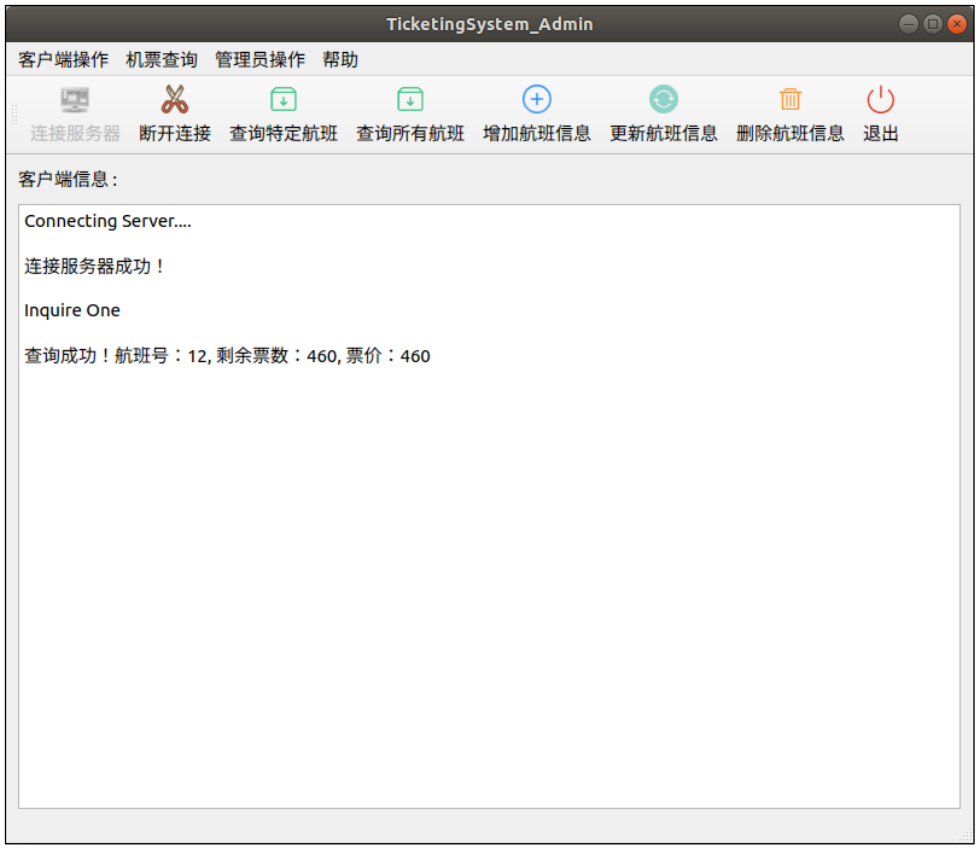

图 4.6 显示航班信息

当用户输入的航班号有误时，售票端显示警告信息。

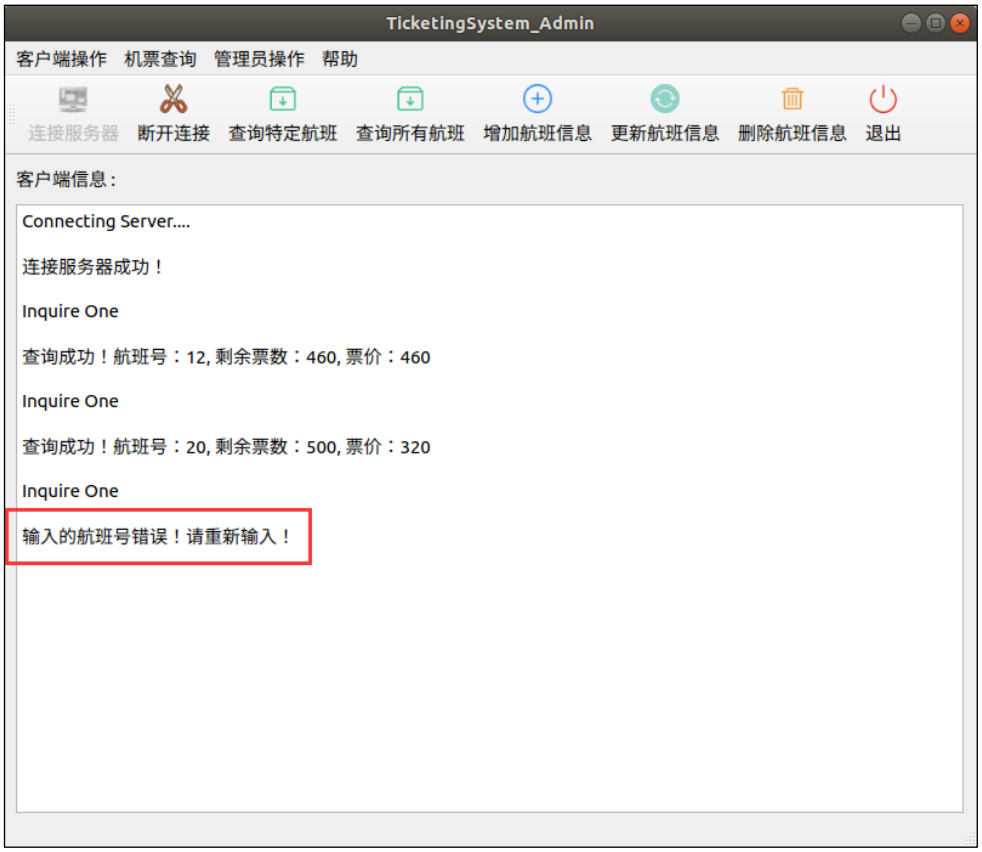

图 4.7 显示警告信息

4.3 查询所有航班功能

点击“查询所有航班”按钮可以查询所有存在的航班信息。如下图 4.8 所示。当航班信息较多时，窗口会自动显示滚动条。可以通过拖动滚动条查看所有航班的具体信息。

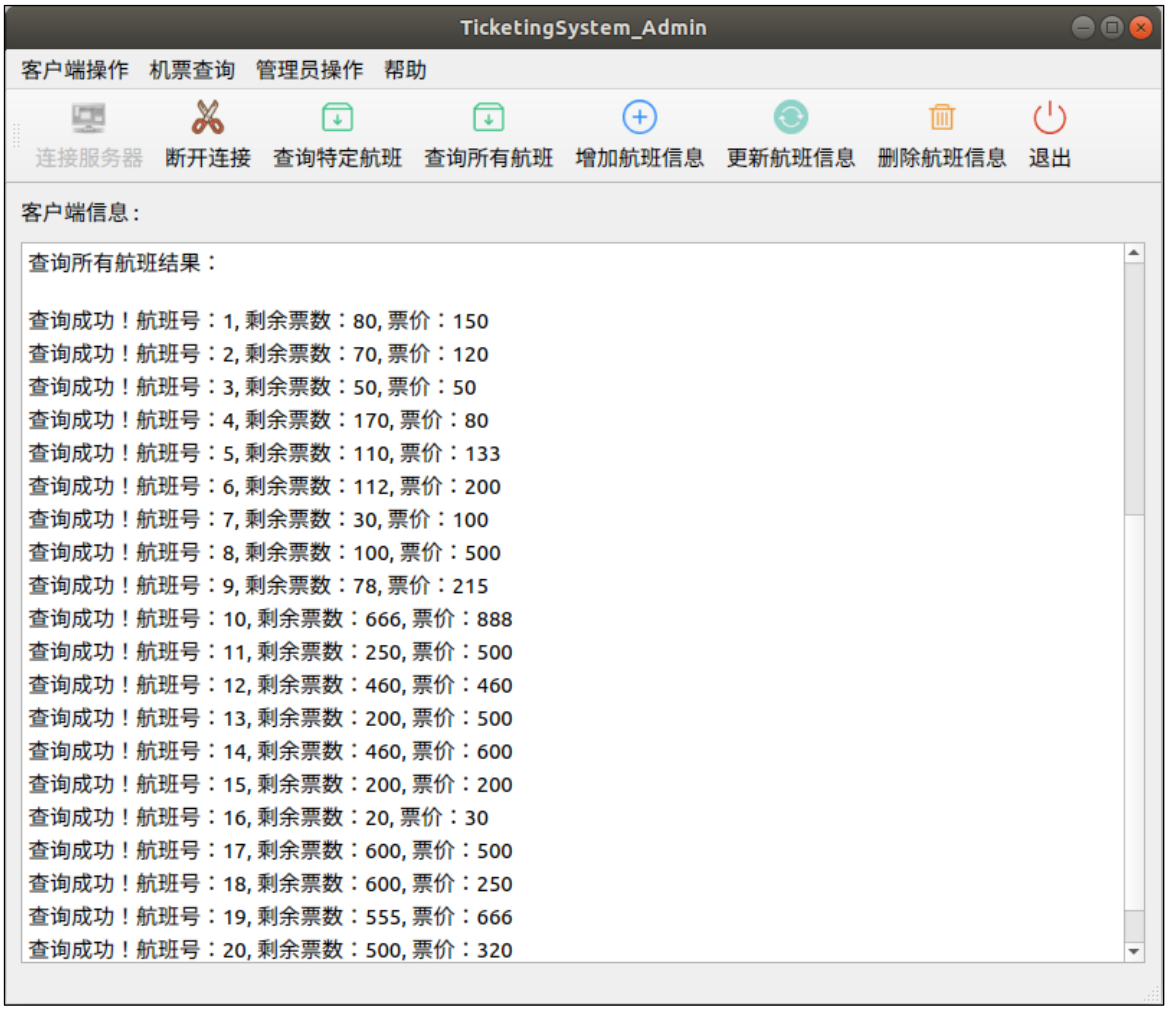

图 4.8 查询航班结果

4.4 增加航班信息功能

点击“增加航班信息”按钮，会弹出提示窗口，要求管理员输入待增加的航班信息。当管理员输入有效的航班信息时，点击 OK 即可确认提交。当管理员点击取消按钮，可以取消当前的操作。

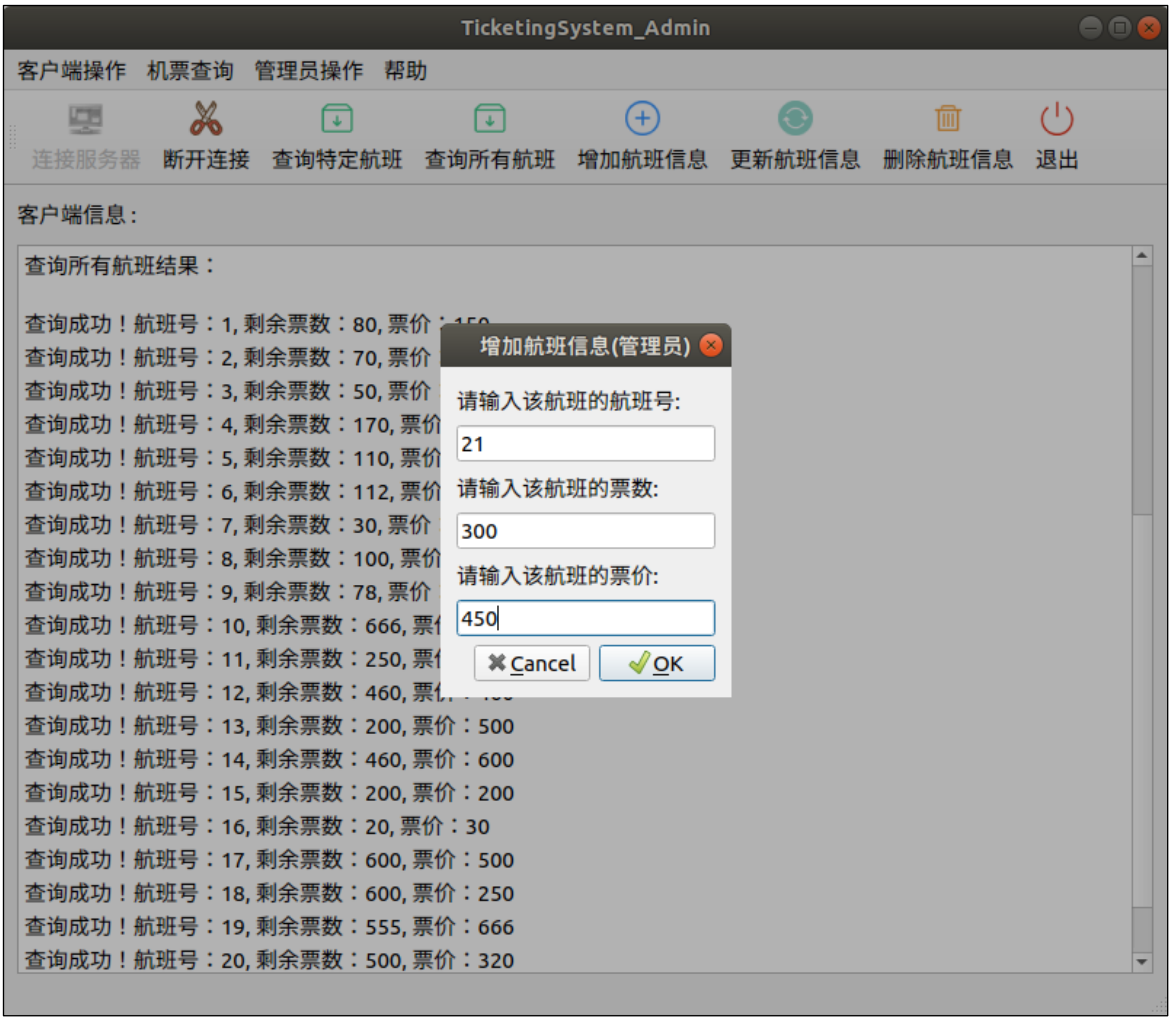

图 4.9 增加航班信息

现测试增加第 21 号航班，票数为 300，票价为 450，点击 OK 确认提交信息。此时售票端会显示增加航班信息成功的提示信息。效果如下图 4.10 所示。

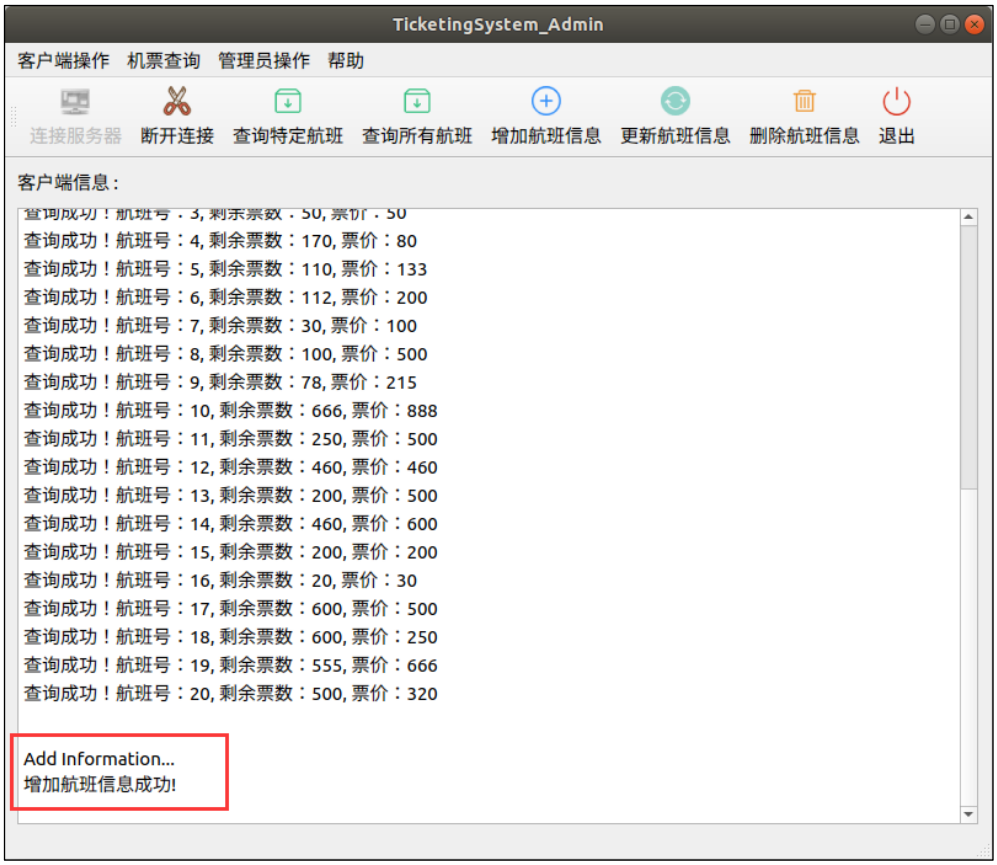

图 4.10 增加航班提示

再次查询所有航班信息，可以看到第 21 号航班的信息已经被添加进来了。

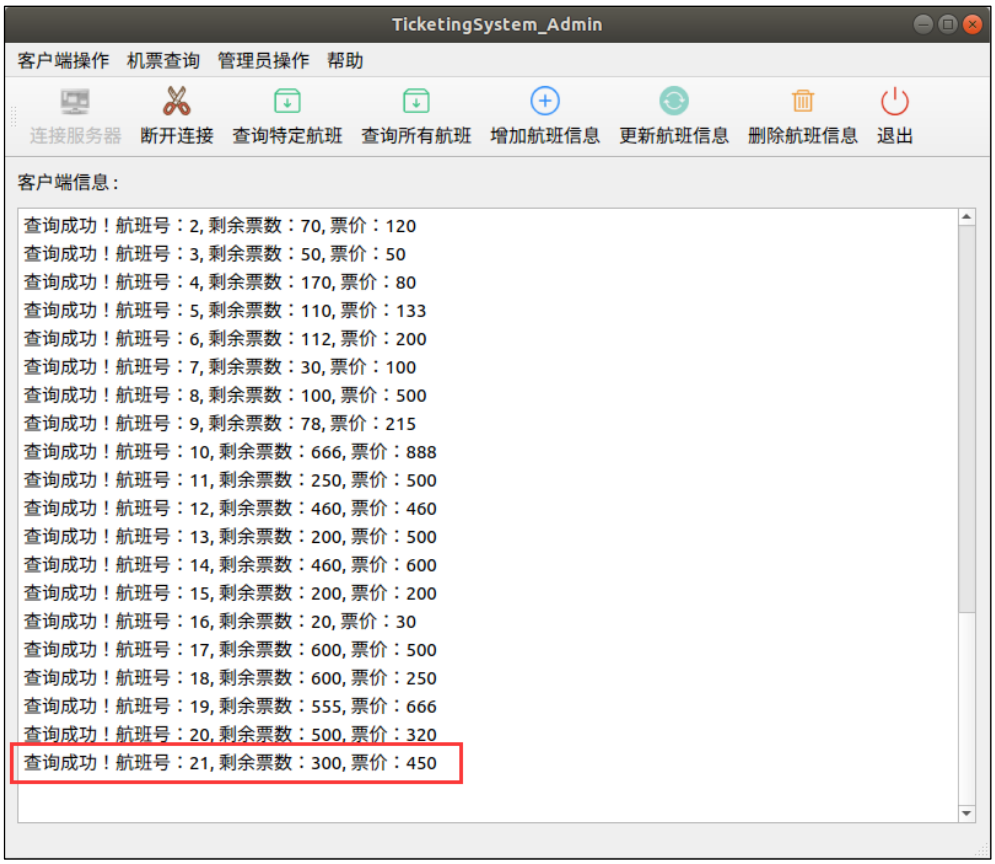

图 4.11 查询航班信息

4.5 更新航班信息功能

管理员点击“更新航班信息”按钮后，同样会弹出提示窗口，要求管理员输入待更新的航班信息，如下图 4.12 所示。当管理员输入有效的航班信息时，点击 OK 即可确认提交。当管理员点击取消按钮，可以取消当前的操作。

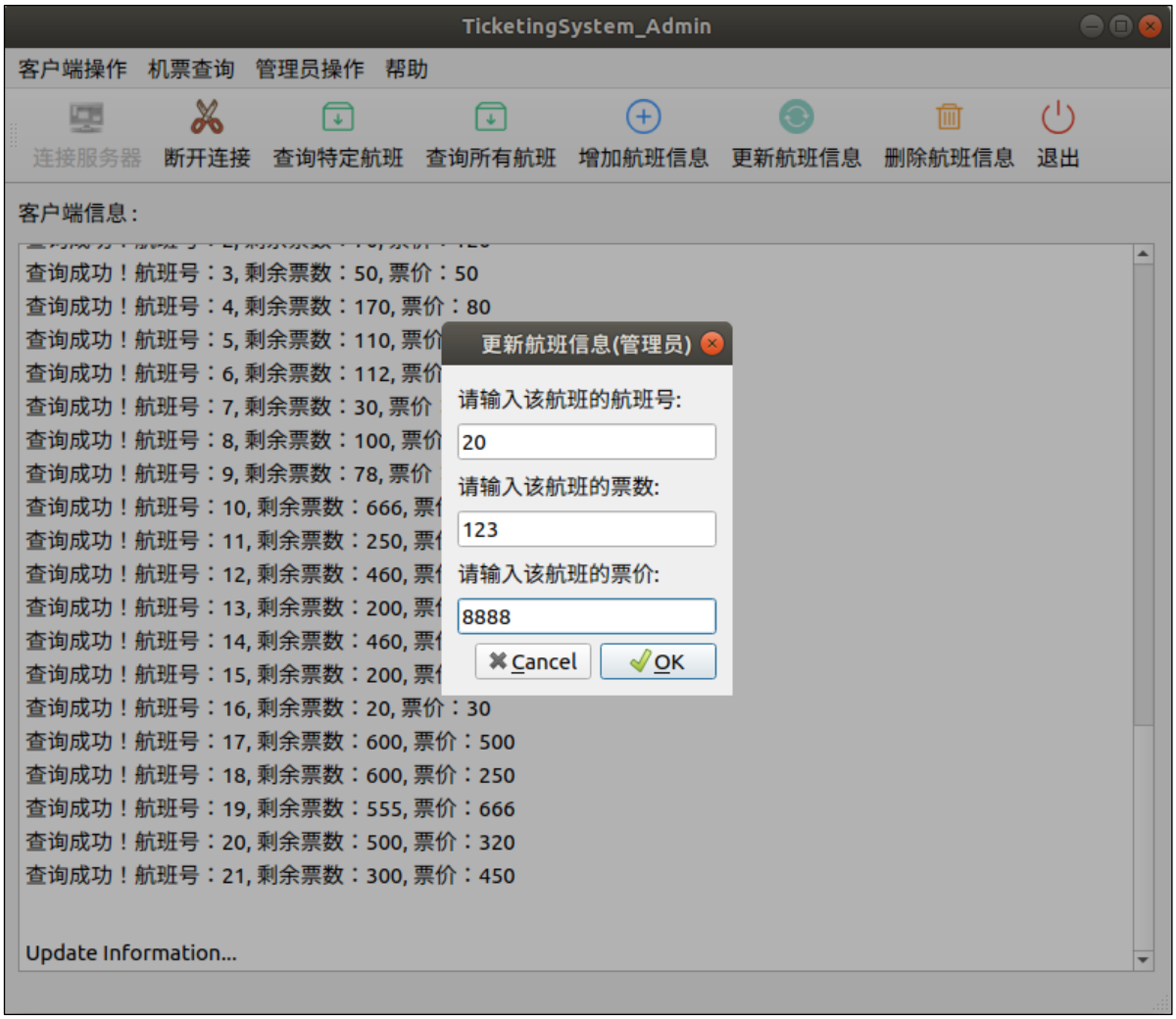

图 4.12 更新航班信息

测试更新一下第 20 号航班，将票数改为 123，票价改为 8888，点击 OK 提交。再次查询所有航班信息，可以看到第 20 号航班的信息已经被更新，如下图 4.13 所示。

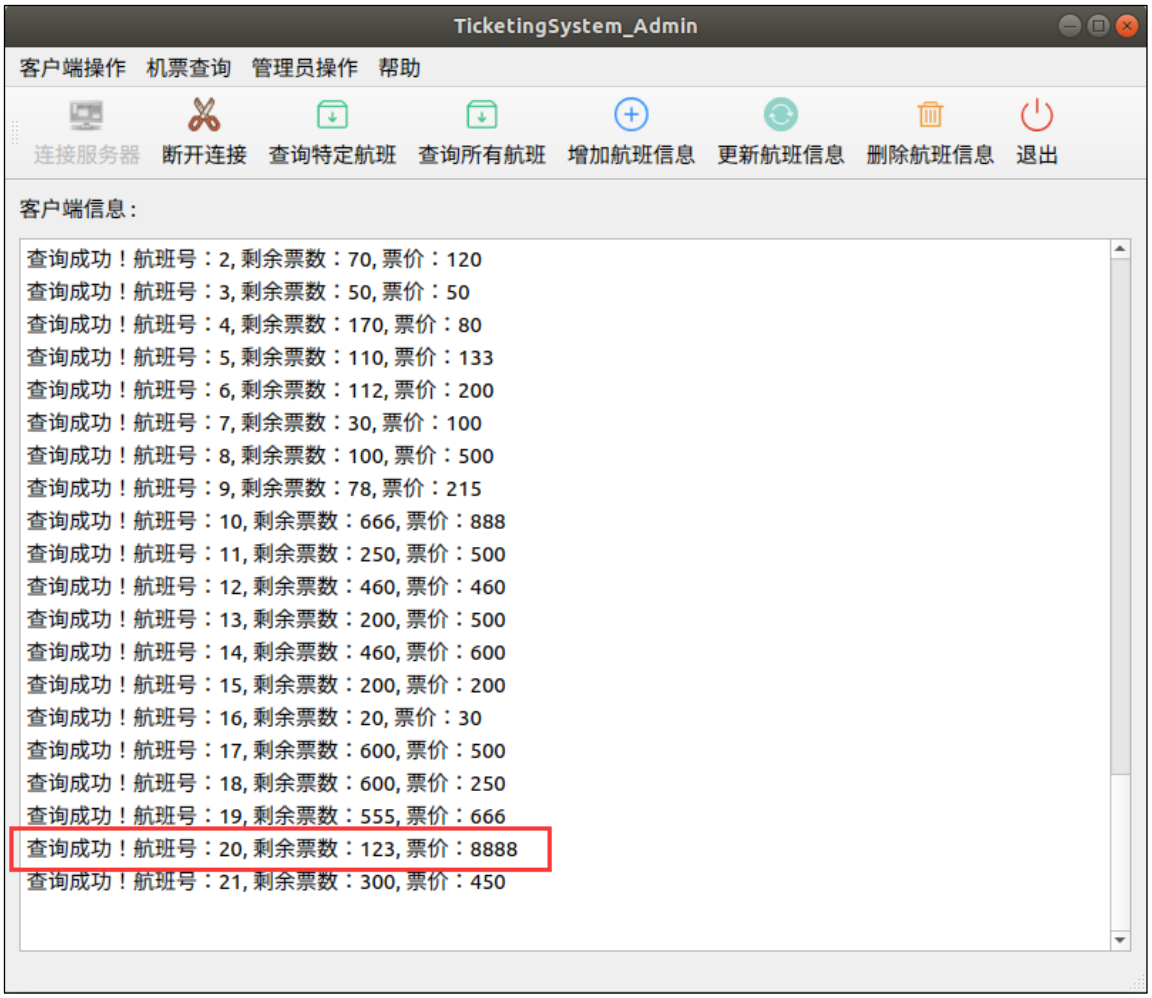

图 4.13 查询航班信息

4.6 删除航班信息功能

管理员点击“删除航班信息”按钮，会弹出提示框，要求管理员填写待删除的航班号。如下图 4.14 所示。当管理员输入有效的航班号时，点击 OK 即可确认提交，删除对应的航班信息。当管理员点击取消按钮，可以取消当前的操作。

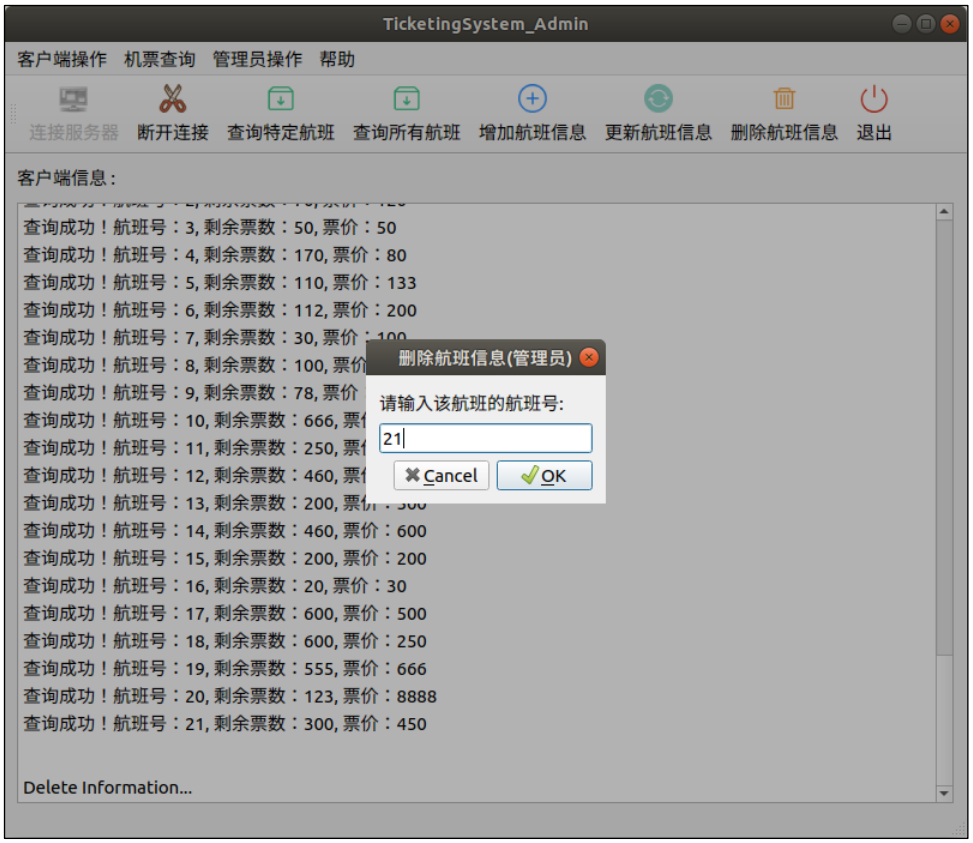

图 4.14 删除航班信息

我们测试输入第 21 号航班，再次查询所有航班，航班信息如下图 4.15 所示。

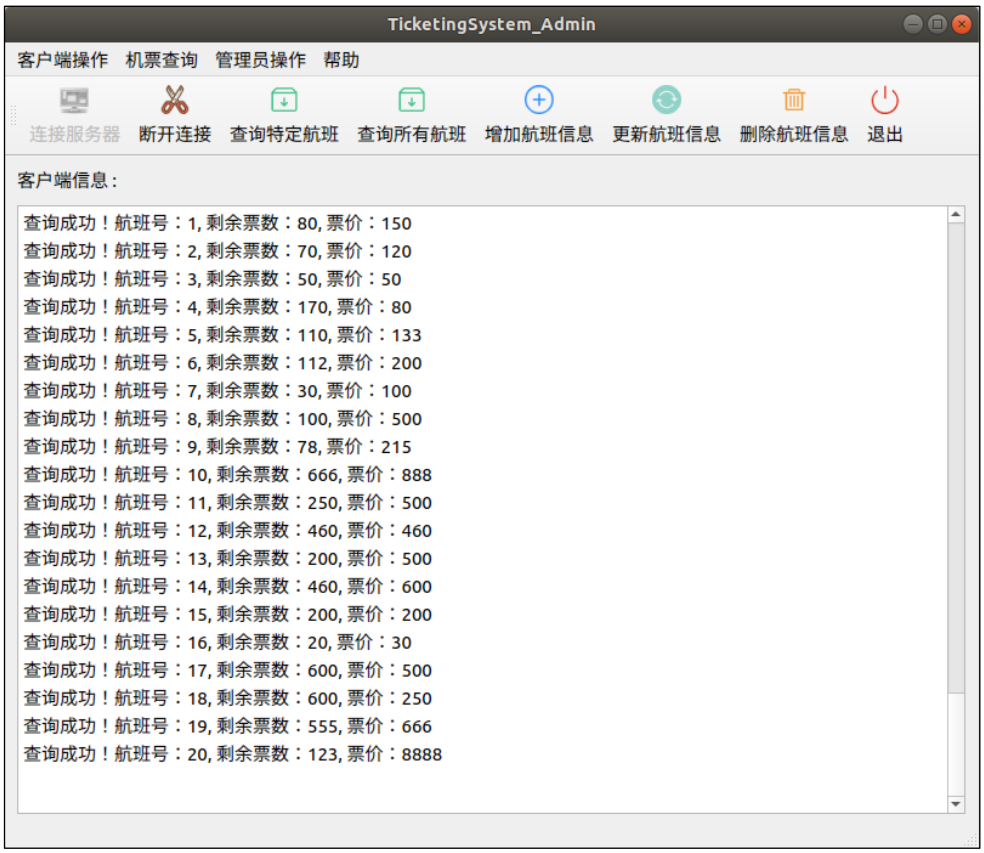

图 4.15 查询航班信息

可以看到，现在第 21 号航班的信息已经被删除了。

4.7 帮助信息和退出程序功能

点击菜单栏的帮助信息按钮，在下拉菜单中找到“显示内容”选项，可以看到程序功能说明，如下图 4.16 所示。

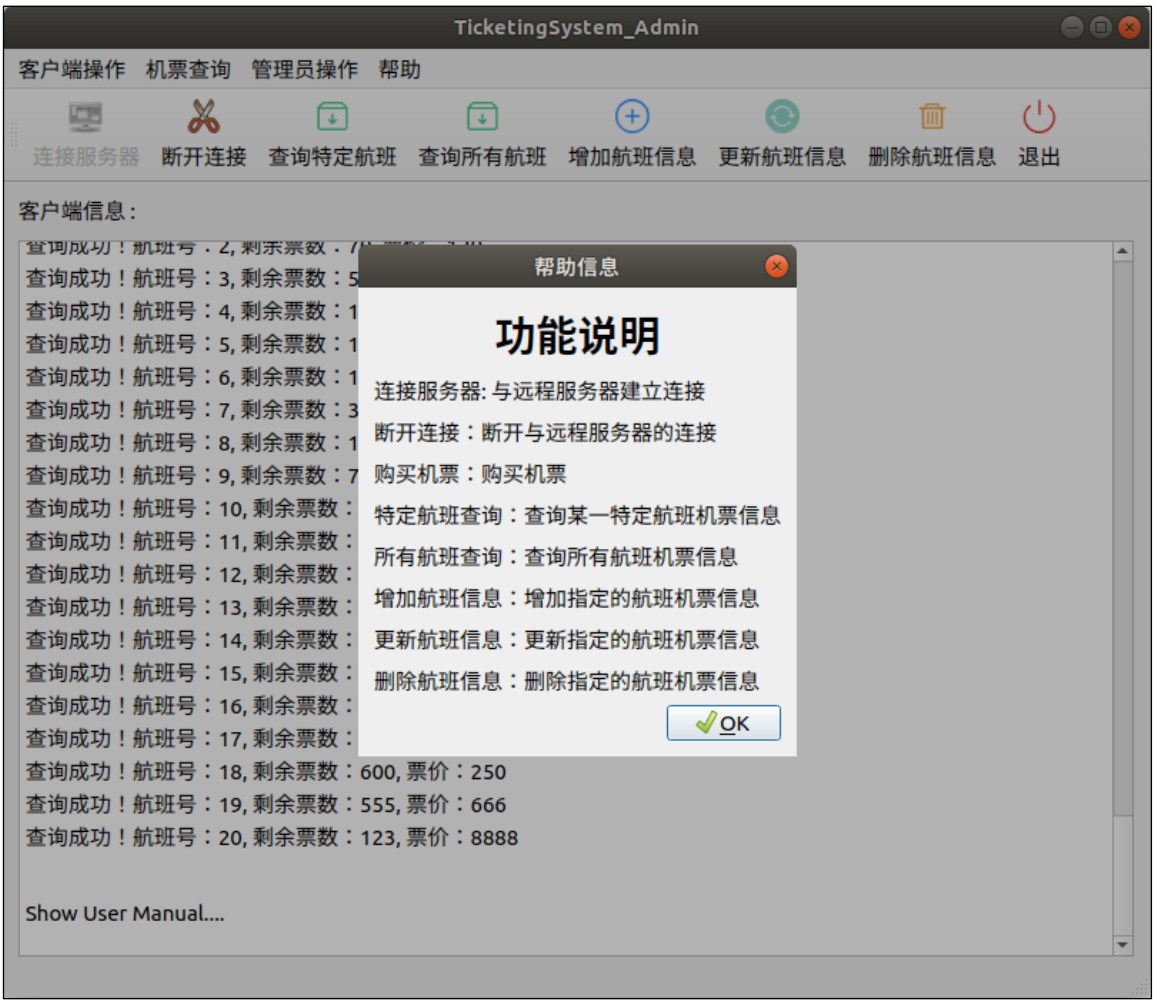

图 4.16 显示功能说明

在“帮助”的下拉菜单中，找到“关于”选项，可以找到程序的版本信息和我们团队的信息，如下图 4.17 所示。

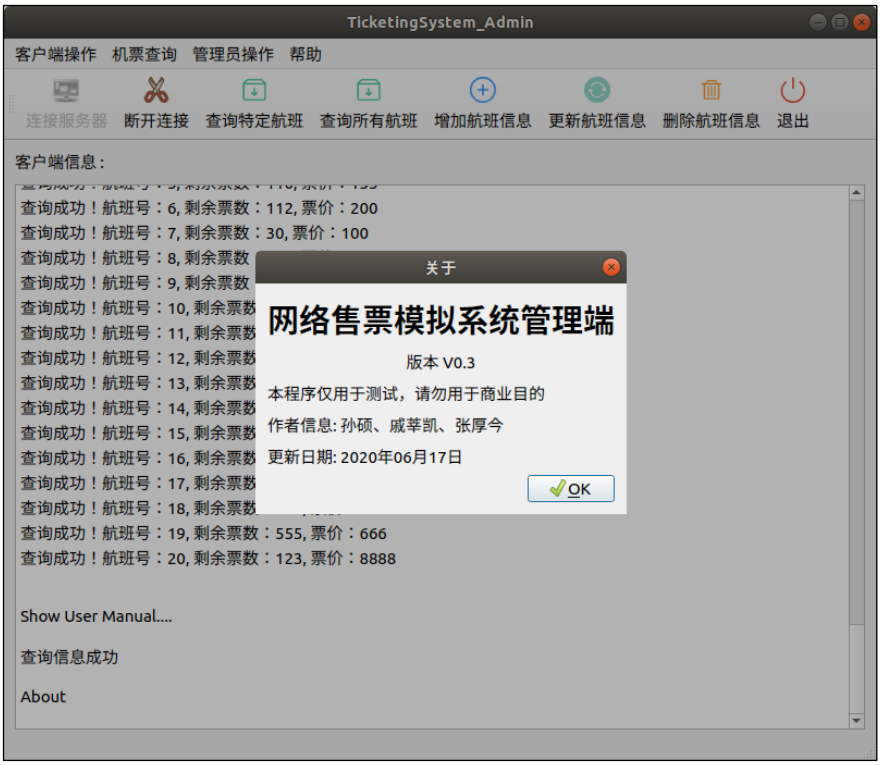

图 4.17 显示团队信息

点击“断开连接”按钮，可以断开售票端与服务器直接的连接。此时售票端界面显示提示信息，同时按钮使能发挥作用，只有“连接服务器”和“退出”按钮使能，其余按钮变为失能状态。

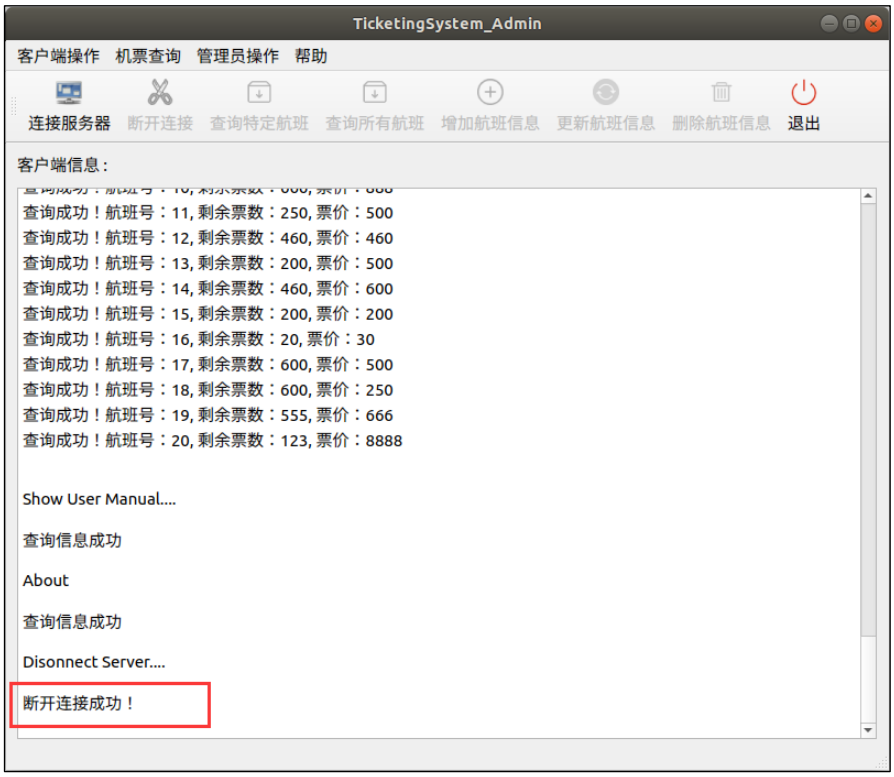

图 4.18 断开连接

点击工具栏最右侧的“退出”按钮，可以退出当前窗口。

5 遇到的问题及解决方案

实训中遇到了很多问题，大部分都是细枝末节的小问题。通过团队之间的讨论，都已经被解决了。其中我印象较深的是有关数据库的一些问题。因为我们是直接将课本上的例程移植进 QT，所以基本的功能都是可以实现的，包括航班查询、机票购买等等。后来我们商量要使用数据库对航班信息进行存储，这就需要在原有的函数基础上增加更多的功能。

我负责制作的售票端程序，需要加入对航班信息的增删改查功能，这些都涉及到对数据库的基础操作。一开始没有思路，后来经过团队讨论，发现其实实现的方法很简单，只需要在原有的通信协议基础上，对更多情况进行处理就可以了。售票端和服务端添加相同的结构体协议内容，服务器通过判断由售票端发来的结构体成员变量，就可以进行相应的数据库操作。而且因为这种操作方式的成功率很高，所以我们没有让服务器返回处理状态信息。

6 实训总结

本次实训制作了一个简易的航班购票模拟系统，使用 Linux C 编写后台程序，由 QT for Linux 软件编写前端界面，实现了后台功能与前端界面的完美结合。在嵌入式编程方面，由我负责制作的售票端模块涉及到了网络套接字通信、数据请求的格式设计等多方面的工作，涉及到很多日常学习中经常遇到的知识点。

通过本次实训学习，不仅使我对网络套接字、多线程编程等技术有了更深入的理解，也为我在编写图形界面的技术思路上积累了宝贵的经验。我认为这次的嵌入式加 QT 的实训是一次非常成功的尝试，日后的项目开发必定离不开图形界面编程，本次实训使我真正感受到了图形界面编程的强大。

在团队合作方面，本次实训也使我受益匪浅。我们团队三人分工合作，项目进行地有条不紊，整个项目实施过程也非常顺利。实训前期时我们的项目整体进展比较缓慢，后期加快了项目进度，在规定时间内顺利完成了实训。遇到问题时我们会及时沟通、积极讨论，从而产生了很多有价值的新思路和新创意，像“客户端的欢迎页面”、“售票端的注册页面”、“航班信息采用数据库存储”等等技术思路，我觉得这些都是非常有价值的团队成果。感谢团队成员孙硕和戚莘凯为项目做出的宝贵贡献，相信在以后的学习和生活中，本次实训一定会对我们产生积极的影响。