

数据库原理

数据库期末项目——酒店管理系统

姓名	学号
XXX	22xxxxxxx
YYY	22yyyyyyy

0 目录

I 引言	3
I.1 背景故事	3
I.2 设计目的	3
I.3 设计环境	3
I.4 人员分工	4
II 设计概要	5
II.1 系统需求分析	5
II.2 系统结构设计	5
II.3 功能模块设计	6
III 详细设计	8
III.1 系统数据库设计	8
III.2 主要功能模块	9
III.2.A 后端	9
III.2.A.a Postgre 类	9
III.2.A.b TableBase 类	10
III.2.A.c 例: Hotel 类	11
III.2.A.d Manager 类	12
III.2.B 前端	13
III.2.B.a ManageForm 类	13
III.2.B.b 例: RoomTableControl 类	14
III.2.B.c 例: ReserveControl 类	16
IV 调试与问题	18
IV.1 问题	18
IV.2 解决方案	18
IV.3 解决效果	19
V 总结	22

VI 附录与指引	22
VII 参考文献	22

I 引言

I.1 背景故事

你将扮演一位坐拥几座酒店的管理者，在电脑桌前邂逅性格各异、能力独特的旅客们，和他们一起创造财富——同时，逐步发掘数据库的真相！

你的账户是 [DataBase](#)，密码是 [password](#)。当你正确输入上述信息时，输入框会变为绿色。

I.2 设计目的

我们将要设计实现的，是一个拥有友好用户界面的、简单易用的、面向酒店管理人员(而不是入住者)的酒店预订管理系统。我们将要实现以下模块：

1. 酒店信息管理负责酒店信息的添加、修改和查询；
2. 房间信息管理负责房间信息的添加、修改和查询；
3. 预订信息管理负责预订信息的录入、修改和查询。

I.3 设计环境

我们首先考虑后端语言的选择。我们希望能够便捷地、现代化地使用程序语言操作数据库，这里便排除掉了繁琐且内存不安全的 C 与 C++ 语言，而在 JAVA, Python, C# 中选择。考虑到大部分的使用环境都是 Windows，且并非所有人都会在计算机中安装 JVM，因此我们又排除掉了 JAVA；但同时考虑小部分情况下的跨平台性，且 Python 不易打包成可执行文件，我们最终选择了 C# 作为后端语言。

另一个有趣的点是，C# 具有独特的 Linq 查询表达式，与数据库语言颇为相似。这就允许我们在本地内存中进行相似的操作。

选择 C# 作为后端语言后，我们惊喜地发现它也能同时胜任前端的构建。对于 C#，WinForm 与 WPF 都是非常成熟的前端框架，其中尤以 WinForm 最具有简易性——它可以使用拖拽的方式构建前端界面！

因此我们最终敲定以 C# 语言作为整个项目的基石，无需分别为了后端与前端专门学习两种语言，更是省去了两个不同语言间的调用过程。项目的环境最终如下所示：

- 数据库：PostgreSQL
- 后端语言：C# [\[1\]](#)
 - 使用官方的 Npgsql 作为连接与操控 PostgreSQL 数据库的驱动程序。
 - [Npgsql 官方教程](#) [\[2\]](#)
- 前端语言：C# [\[1\]](#)
 - 使用 WinForm 作为开发前端的平台，而不是更加复杂的 WPF。
 - 出于美观考量，使用开源的 AntdUI 作为界面库。
 - [AntdUI Gitee 发布页](#) [\[3\]](#)

I.4 人员分工

- XXX: 组长。负责统筹协调, 后端代码编写, 部分前端代码编写, 前端界面美化与审查, 报告书编写。
- YYY: 组员。负责前端代码编写, 前后端的逻辑连接。

II 设计概要

II.1 系统需求分析

1. 酒店信息管理

酒店具有的具体信息有：酒店名称，酒店地址，酒店星级，房间类型及其总数。

2. 房间信息管理

酒店房间的具体信息有：酒店地址，房间编号，房间类型，单天价格，是否已被预订。

3. 预订信息管理

预订订单的具体信息有：订单编号，酒店地址，预订房间号，预订人身份证号，起始日期，旅居天数。

结合上述的具体信息，我们显然可以发现系统中具有实体集：酒店(Hotel)、房间(Room)、预订人(Reserver)。事实上，我们还应当从中分离出一个“房间类型(RoomType)”的实体集出来。

其中的联系集是：(Hotel, RoomType)，酒店持有其独特的房间类型；(Room, RoomType)，房间自然是具有房间类型的；(Reserver, Room)，预订人会预订特定的房间；(Reserver, Hotel)，由于房间归属于酒店，因此还需要附加酒店信息。整理后我们可以绘制 E-R 图如 [图 1 \[酒店预订管理系统的 E-R 图\]](#) 所示。

方便起见，在之后的设计中，我们以酒店编号(hotelNO)来代替地址(Address)。在数据库中，相关属性会以 hotelNO 的名字出现；在用户界面中，尽管列名会显示为“酒店地址”，但其中的内容依然会以不同的数字形式出现。

II.2 系统结构设计

我们设计的系统中，通用的流程是：在未创建表的情况下，首先创建表格并初始化。同时，在本地内存与数据库中都持有表格。之后的一切操作，先在本地图通过算法初步判断是否是合法的更新；如果合法，那么就更新本地表格；之后，再更新数据库中的数据。如果数据库更新成功，后端才返回更新成功的指示。

从结构上来说，我们自底向上设计。我们首先实现 C# 语言对 Postgre 数据库的相关操作；之后在此基础上，实现对酒店预订信息的管理；最后基于上述实现，设计美观优雅易操作的用户界面。这将会在下一节 [章节 II.3 \[功能模块设计\]](#) 中体现。

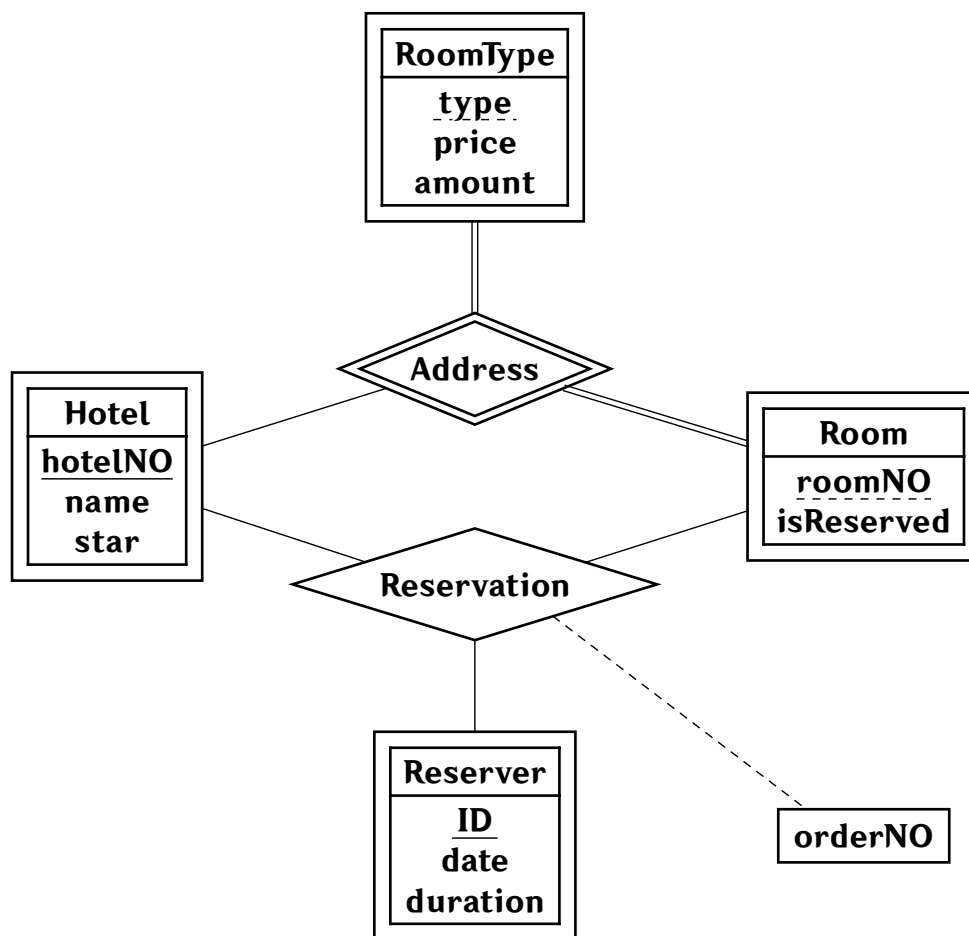


图1 酒店预订管理系统的 E-R 图

II.3 功能模块设计

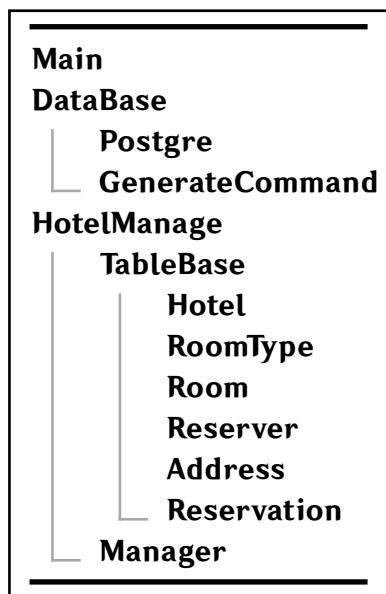


图2 功能模块设计层级结构(后端)

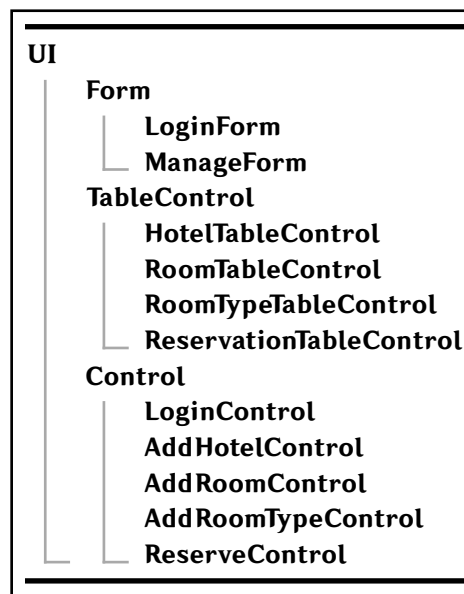


图3 功能模块设计层级结构(前端)

在这个项目中，我们设计的功能模块层级结构如上图所示。由于 C# 语言的语言特性，图中列出的层级结构不一定代表真实文件结构，仅具有逻辑含义。同时，所列项目也不一定是文件名或类名，具体含义请看下面的具体释义。

首先是后端部分：

- **Main** 是整个项目的入口，即主函数
- **DataBase** 命名空间，用于较底层与数据库沟通的功能
 - **Postgre** 类，即对 **Npgsql** 中基础功能的封装
 - **GenerateCommand** 类，用于较为系统地生成 SQL 语句
- **HotelManage** 命名空间用于管理酒店数据库相关表格
 - **TableBase** 类，是六张表格的基类
 - **Hotel, RoomType**...等六张内存中的表格
 - **Manager** 类，管理上述六张表格

接下来是前端部分。下述层级关系中的第二层是逻辑结构，在实际代码中没有这样的区分。带有 **Control** 的是控件，与 **Form** 窗口具有一定区别——控件是窗口中的某个部分：

- **UI** 命名空间，实现项目的前端界面
 - **Form** 是窗口界面，本项目中存在两个窗口：
 - **LoginForm**，登录界面的窗口，同时用于设置数据库连接参数
 - **ManageForm**，登录成功后进入的管理界面窗口，在这里实现对酒店各项的管理
 - **TableControl**，用于显示表格的控件
 - **HotelTableControl**，显示酒店的表格
 - **RoomTableControl**，显示特定酒店的房间的表格
 - **RoomTypeTableControl**，显示特定酒店的房间类型的表格
 - **ReservationTableControl**，显示特定酒店的订单的表格
 - **Control**，用于显示提示用户输入的控件
 - **LoginControl**，用户设置数据库连接参数
 - **AddHotelControl**，用户查看或添加酒店
 - **AddRoomControl**，用户查看或添加房间
 - **AddRoomTypeControl**，用户查看或添加房间类型
 - **ReserveControl**，用户查看或添加订单

III 详细设计

III.1 系统数据库设计

根据 E-R 图，六张表对应的创建表格指令分别是：

1. 酒店表：

```
1 CREATE TABLE Hotel (
2     hotelNO int NOT NULL PRIMARY KEY,
3     name char(10) NOT NULL,
4     star int NOT NULL);
```

謹 SQL

2. 房间类型表：

```
1 CREATE TABLE RoomType(
2     hotelNO int NOT NULL REFERENCES Hotel(hotelNO) ON DELETE CASCADE,
3     type char(10) NOT NULL,
4     price int NOT NULL,
5     amount int NOT NULL,
6     PRIMARY KEY (hotelNO, type));
```

謹 SQL

3. 房间表：

```
1 CREATE TABLE Room(
2     hotelNO int NOT NULL REFERENCES Hotel(hotelNO) ON DELETE CASCADE,
3     roomNO int NOT NULL,
4     isReserved boolean NOT NULL,
5     PRIMARY KEY (hotelNO, roomNO));
```

謹 SQL

4. 预订人表：

```
1 CREATE TABLE Reserver (
2     hotelNO int NOT NULL REFERENCES Hotel(hotelNO) ON DELETE CASCADE,
3     ID int NOT NULL PRIMARY KEY,
4     date date NOT NULL,
5     duration interval NOT NULL);
```

謹 SQL

5. 地址表：

```
1 CREATE TABLE Address (
2     hotelNO int NOT NULL,
3     roomNO int NOT NULL,
4     type CHAR(10) NOT NULL,
5     PRIMARY KEY (hotelNO, roomNO),
6     FOREIGN KEY (hotelNO, roomNO) REFERENCES Room(hotelNO, roomNO) ON DELETE CASCADE,
```

謹 SQL


```
7 FOREIGN KEY (hotelNO, type) REFERENCES RoomType(hotelNO, type) ON DELETE CASCADE);
```

6. 预订表:

```
1 CREATE TABLE Reservation (
2     orderNO serial NOT NULL PRIMARY KEY,
3     ID      int    NOT NULL REFERENCES Reserver(ID) ON DELETE CASCADE,
4     hotelNO int    NOT NULL,
5     roomNO  int    NOT NULL,
6     FOREIGN KEY (hotelNO, roomNO) REFERENCES Room(hotelNO, roomNO) ON DELETE CASCADE);
```

謹 SQL

在每一张表中，所有属性都完全依赖于主键或主键元组，因此这些表至少是符合 2-NF 范式的；同时，也不存在传递依赖，因此系统数据库是符合 3-NF 范式的。

这里规定了所有属性均不能为 NULL，从根源上避免了数据的不完整性。同时，几乎所有与 hotelNO 和 roomNO 有关的属性都设置为 ON DELETE CASCADE，这样在删除酒店或房间时，相关的信息也都会被删除，避免了因为被引用而无法被删除、也使得删除操作更加简便，减少了后续的心智负担。

唯一的例外是 RoomType 中的属性 type。按道理来说，当某类房间被删除时，对应的所有房间也应删除。假设删除了一种房间类型，此时 Address 表中的房间对应关系也都被删除，但是 Room 表中依然存在相关房间——它们只是不再出现在 Address 表中。而 Room 是整个数据库关系中最重要两个属性之一，不能在其中设置 roomNO 为其他表的外键(也避免循环依赖)。所以要求在删除某个房间类型时，必须再手动删除所有相关房间。

III.2 主要功能模块

本项目中，核心操作都在后端，由后端提供已经封装好的接口，前端只需要调用这些接口即可，也只需要处理前端自己的逻辑。

III.2.A 后端

III.2.A.a Postgre 类

这个类是对 Npgsql 的封装，提供了一些基础的数据库操作，例如连接数据库、执行 SQL 语句等。只需要提供数据库的连接参数，这个类就会自动连接数据库。

类中提供两个通用的操作函数，public NpgsqlDataReader? Query(string query) 和 public int NotQuery(string sentence)，参数即是 SQL 语句。前者用于查询(即 SELECT 语句)，后者用于更新(即 CREATE, DROP, INSERT, UPDATE, DELETE 等语句)。当然，也提供了更具体的函数，例如 public int Insert(string sentence)，但实际上就是调用了 NotQuery(...) 函数。

```
1 public NpgsqlDataReader? Query(string query)
2 {
3     if (string.IsNullOrEmpty(query)) return null;
4     using var command = new NpgsqlCommand(query, Connection);
```

禱 C#

```

5     return command.ExecuteReader();
6 }
7
8 public int NotQuery(string sentence)
9 {
10     if (string.IsNullOrEmpty(sentence)) return 0;
11     using var command = new NpgsqlCommand(sentence, Connection);
12     return command.ExecuteNonQuery();
13 }

```

类中提供返回 `Adapter` 的函数，用于将 `DataTable` 与数据库中的表格进行同步。这个函数会返回 `NpgsqlDataAdapter`，可以将 `DataTable` 中的数据更新到数据库中，也能使数据库中的表格写入到 `DataTable` 中。正如我们即将要提到的 [章节 III.2.A.b \[TableBase 类\]](#) 中所述，每个表格都需要持有一个对应的 `Adapter`，才能使之与数据库同步。

```

1 public NpgsqlDataAdapter? Adapter(string tableName) 禳 C#
2 {
3     if (Connection is null) return null;
4     var adapter = new NpgsqlDataAdapter($"SELECT * FROM {tableName};", Connection);
5     _ = new NpgsqlCommandBuilder(adapter);
6     return adapter;
7 }

```

III.2.A.b TableBase 类

这个类是六张表格管理的基类。在这个类中，已经定义好一个表格类应当具有的属性，例如表格名 `Name`、存储于内存中的表格副本 `Table`，以及最重要的、能够实现与数据库同步的 `Adapter`。

在这个类中，已经定义好 `Table` 同步到数据库中的函数 `public virtual int Update()`，返回值是更新的行数。需要注意：由于各种原因，同步可能失败，因此需要捕获异常；并在发生异常时，拒绝本地已执行的更新，恢复到上一次同步的状态。拒绝使用的函数是 `public void Reject()`。

```

1 public virtual int Update() 禳 C#
2 {
3     try {
4         int changes = Adapter.Update(Table);
5         Table.AcceptChanges();
6         return changes;
7     } catch (Exception) {
8         Table.RejectChanges();
9         return 0;
10    }
11 }
12

```

```
13 public void Reject() { Table.RejectChanges(); }
```

III.2.A.c 例：Hotel 类

正如 章节 III.2.A.b [TableBase 类] 所说，所有表格管理类都继承自 TableBase。在此基础上，每一个表格管理类实现基于自己属性的相关管理逻辑。现在以 Hotel 类为例说明。

在我的实现中，各类会持有一个专属的 Attribute 枚举类型，用以描述表中的属性名。例如本例中：

```
1 public class Hotel : TableBase 禛 C#
2 {
3     /*-----Public Enum-----*/
4
5     public enum Attribute
6     {
7         hotelNO, name, star
8     }
9 }
```

同时，应当都具有 public bool Add(...) 和 public void Delete(...) 方法。由于各表格属性不同，所以参数不同，因此并没有作为 TableBase 中规定的接口。

```
1 public bool Add(int hotelNO, string hotelName, int hotelStar) 禛 C#
2 {
3     if (Table.Rows.Find(hotelNO) is not null) return false;
4     Table.Rows.Add([hotelNO, hotelName, hotelStar]);
5     return true;
6 }
7
8 public void Delete(int hotelNO)
9 {
10     Table.Rows.Find(hotelNO)?.Delete();
11 }
```

以上实现了增删，还需要针对每个属性实现改操作。应当注意：虽然各表中都实现了改主键的函数，但实际上由于主键和外键约束的存在，并不推荐调用这些改主键的函数。例如在本例中，可以修改酒店名字：

```
1 public bool Rename(int hotelNO, string hotelName) 禛 C#
2 {
3     var hotelInfo = Table.Rows.Find(hotelNO);
4     if (hotelInfo is null) return false;
5     try {
6         hotelInfo["name"] = hotelName;
7     } catch (Exception) { }
```

```

8         return false;
9     }
10    return true;
11 }

```

III.2.A.d Manager 类

Manager 类是后端的重头大戏。一些操作，例如有客户预订房间，这必然会造成多张表的变动，这些逻辑如果不封装起来，那么前端不仅要持有这六张表，还要协调这些表之间的联动关系，这无疑增加了代码的耦合性。因此，由 **Manager** 类持有六张表并实现操作逻辑，前端只需要持有一个这个类的实例，调用它的接口即可。

回到刚才的例子，例如在前端界面中点击了某个房间的预订按钮，那么只需要调用 `public int ReserveRoom(int reserverID, int hotelNO, int roomNO, DateTime date, int duration)` 即可：

```

1  public int ReserveRoom(int reserverID, int hotelNO, int roomNO, DateTime date,
2  int duration)
3  {
4      if (Room.Reserve(hotelNO, roomNO)) {
5          Reserver.Delete(reserverID);
6          Reserver.Add(reserverID, date, duration);
7          Reservation.Add(reserverID, hotelNO, roomNO);
8      }
9      return Room.Update() + Reserver.Update() + Reservation.Update();
10 }
11 else return 0;
12 }

```

Manager 类中操作相关的返回值，都是数据库中总共更新的行数。当然，也存在一些查询相关的函数，其返回值就由函数行为决定。例如，注意到表中并没有存储还剩多少房间已被预订，但这是可以由 `amount` 和 `isReveived` 属性计算出来的，查询剩余房间数的函数就如下所示：

```

1  public int RoomRemain(int hotelNO, string type)
2  {
3      return (from room in Room.Table.AsEnumerable()
4              join address in Address.Table.AsEnumerable()
5              on room.Field<int>("roomNO") equals address.Field<int>("roomNO")
6              where (room.Field<int>("hotelNO") == hotelNO) &&
7              (address.Field<int>("hotelNO") == hotelNO)
8              && (address.Field<string>("type").TrimEnd() == type) &&
9              (room.Field<bool>("isReserved") == false)
10             select room).Count();
11 }

```

注意到这个函数使用了 C# 中的 Linq 表达式，与 SQL 语句非常像，这也是我们选择 C# 语言的原因之一。

III.2.B 前端

III.2.B.a ManageForm 类

ManageForm 类是登录之后，用于管理的窗口。这个窗口能在侧边栏选择当前要管理的项目，如下所示：

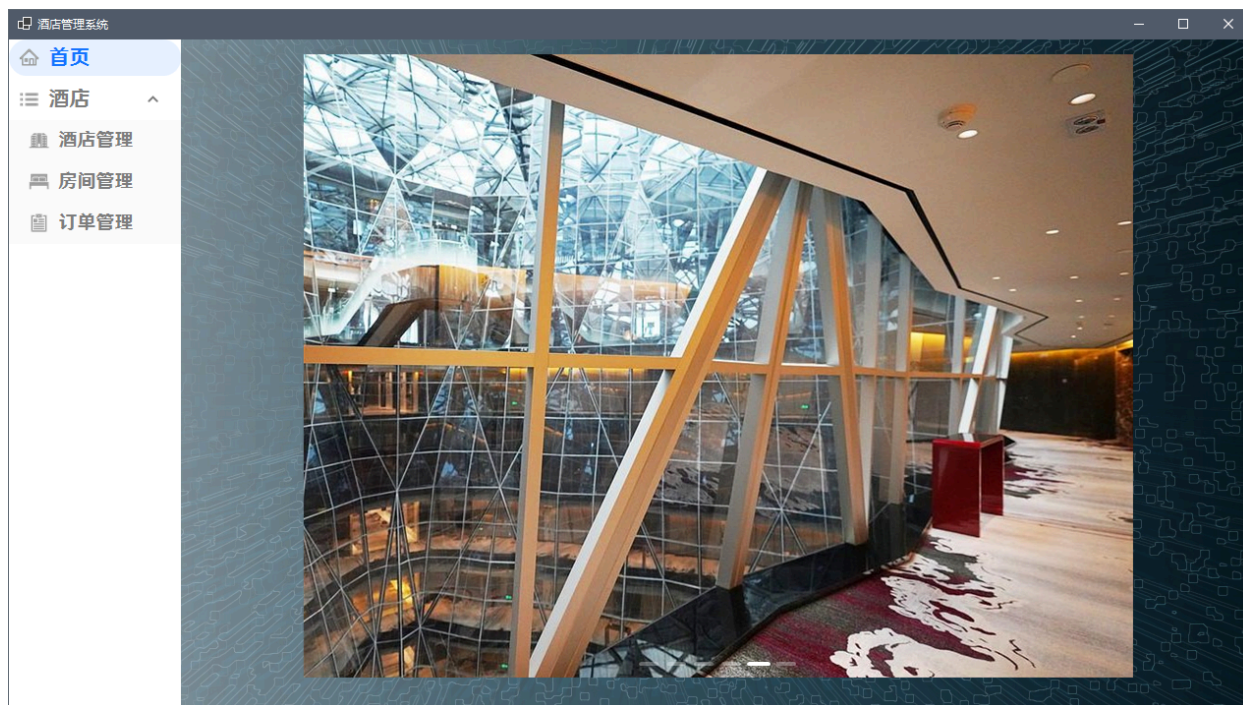


图 4 管理窗口界面

选择 酒店 栏中任意一项，都会展现对应的表格。例如，选择 酒店-房间管理，界面如下所示：

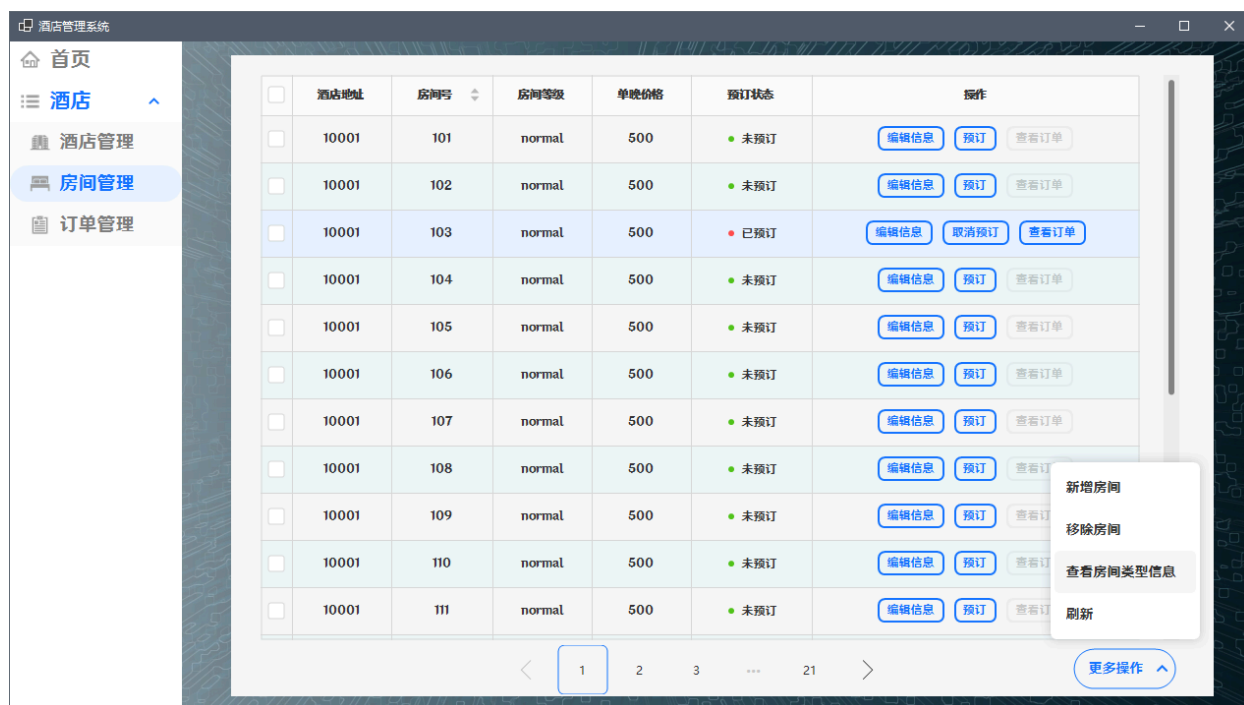


图 5 酒店-房间管理界面

这些表格均由...TableControl 控件实现。接下来就以 图 5 [酒店-房间管理界面] 中的表格为例，说明...TableControl 的实现。

III.2.B.b 例：RoomTableControl 类

显示房间的表格类被命名为 RoomTableControl。带有 Control 一词意味着它是一个控件，将要嵌入到窗口中、而不是独立为一个窗口。TableControl 意味着控件的主要功能是显示表格。

...TableControl 具有以下共性。首先，它们都需要持有同一个 Manager 类的实例，这样才能实现前端操作与后台数据的同步。其次，它们都要持有一个 AntdUI.Table 控件用于显示表格，这自不必多说；重要的是，它们都应实现一个 `private object GetPageData(int current, int pageSize)` 函数，用于分页显示表格。这是因为表格中的数据可能非常多，如果一次性全部显示，一则导致界面卡顿，二则也会对使用者造成心智负担。

例如，RoomTableControl 类中的 `GetPageData(...)` 函数如下：

```

1  private object GetPageData(int current, int pageSize)  禎 C#
2  {
3      var list = new AntList<AntItem[]>(pageSize);
4      var table = manager.GetHotelRooms(hotelNO);
5      int start = Math.Abs(current - 1) * pageSize;
6      int end = Math.Min(start + pageSize, table.Rows.Count);
7      RoomTablePagination.Total = table.Rows.Count;
8
9      for (int i = start; i < end; i++) {

```



```

10      bool isReserved = (bool)table.Rows[i]["isReserved"];
11      var state = isReserved ? TState.Error : TState.Success;
12      string strState = isReserved ? "已预订" : "未预订";
13      string reserveBtnText = isReserved ? "取消预订" : "预订";
14
15      list.Add(new AntItem[] {
16          new AntItem("check", false),
17          new AntItem("hotelno", table.Rows[i]["hotelNO"]),
18          new AntItem("roomno", table.Rows[i]["roomNO"]),
19          new AntItem("type", table.Rows[i]["type"]),
20          new AntItem("price", table.Rows[i]["price"]),
21          new AntItem("isreserved", new CellBadge(state, strState)),
22          new AntItem("operate", new CellLink[] {
23              new CellButton($"EditRoom{i}", "编辑信息") { Type = TTypeMini.Primary,
24                  Ghost = true, BorderWidth = 2F },
25              new CellButton($"Reserve{i}", reserveBtnText) { Type = TTypeMini.Primary,
26                  Ghost = true, BorderWidth = 2F },
27              new CellButton($"ViewReservation{i}", "查看订单") { Type =
28                  TTypeMini.Primary, Ghost = true, BorderWidth = 2F, Enabled = isReserved },
29          })
30      });
31      return list;
32  }

```

注意到我们在列的最后添加了三个按钮，分别是编辑信息、预订、查看订单。因为这三种操作是针对单个房间的，所以每个房间都有这些按钮是一种合理的设计。

同时注意到 [图 5 \[酒店-房间管理界面\]](#) 的右下角有一个 更多功能 选择栏，可以在其中实现新增单个房间、批量删除房间、以及刷新表格的功能。其中，刷新表格是一个安慰剂设计，虽然它确实能够刷新界面，但由于在我们的设计中、只要有数据更改就会自动刷新，所以视觉上并不会发生什么变化。

如果点击“查看房间类型信息”，那么会出现以下界面：



图 6 房间类型表格

这张表也是通过 RoomTypeTableControl 类实现的。具体细节不再赘述。

III.2.B.c 例: ReserveControl 类

如果在 图 5 [酒店-房间管理界面] 中点击任意一个“预订”按钮，那么会出现以下界面：

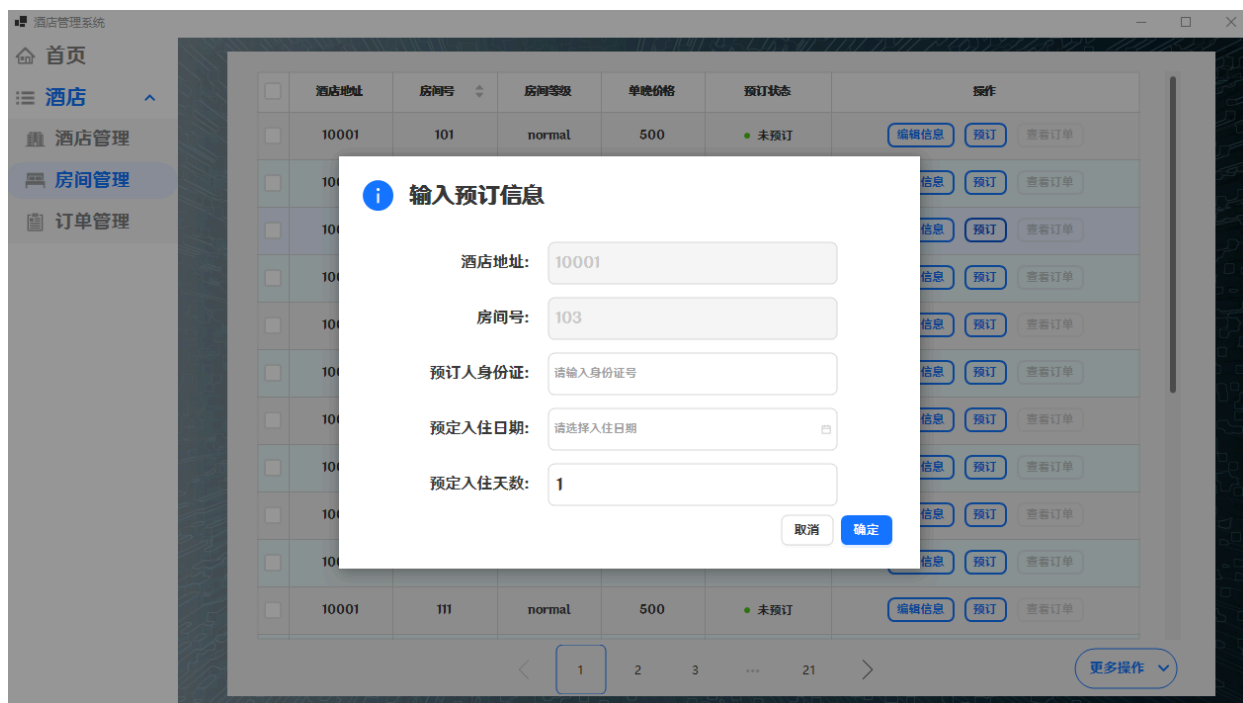


图 7 预订界面

像这样提示用户输入的子窗口控件，都被命名为...Control，例如用于预订信息输入的就叫ReserveControl。在这些窗口中，需要对用户输入内容进行一定的限制，如果输入不合法，就应当拒绝此次输入。例如，如果“预订入住天数”中填入了小于等于0的数字；输入了不存在的酒店或房间；没有任何输入，等等。这些判断逻辑汇聚在一个函数中——不如说，...Control类都应当实现一个public bool IsValid()接口，外界调用者使用此函数测试其中的输入内容是否有效之后，才从中(通过public (...) GetValues()函数)获取其中的输入值。例如，在本例ReserveControl类中，以上两个函数的实现如下：

```

1  public bool IsValid()
2  {
3      if (HotelNOInput.Text.Length != 5) return false;
4      if (!int.TryParse(HotelNOInput.Text, out int _)) return false;
5      if (RoomNOInput.Text == "请输入房间号") return false;
6      if (!int.TryParse(RoomNOInput.Text, out int _)) return false;
7      if (IDInput.Text == "请输入身份证号") return false;
8      if (!int.TryParse(IDInput.Text, out int _)) return false;
9      if (DatePicker.Value is null) return false;
10     return true;
11 }
12
13 public (int hotelNO, int roomNO, int ID,
14     DateTime date, int duration) GetValues()
15 {
16     int hotelNO = 0, roomNO = 0, ID = 0;
17     int.TryParse(HotelNOInput.Text, out hotelNO);
18     int.TryParse(RoomNOInput.Text, out roomNO);
19     int.TryParse(IDInput.Text, out ID);
20     return (hotelNO, roomNO, ID, (DateTime)DatePicker.Value,
21         (int)DurationInputNumber.Value);
22 }

```

在这里，我们使用了C#中的一种语法糖，即GetValues()函数的返回值是一个匿名元组。只需要将所需的数据类型简单地填入小括号中，而无需为它专门设计一个结构体或者类。

IV 调试与问题

IV.1 问题

在项目中，我们遇到过以下问题：

1. 在修改内存中的表格时，有时会与数据库中的数据产生某些出入，导致本地表格 **Update** 到数据库失败——进而导致整个项目崩溃。
2. 由于项目使用 WinForm 构建前端界面，它会自动生成一些代码，而这些代码是编程者不可控的。当后端抛出异常时，即便在自定义函数中存在异常处理，但由于中间存在 WinForm 自动生成的代码，异常不会继续向上抛出，从而导致项目崩溃。
3. 如在 [章节 III.1 \[系统数据库设计\]](#) 中所述，删除房间类型时，需要手动删除所有相关房间。这在实际操作前是没有考虑到的。
4. 有些操作，例如删除某个房间类型时，会导致多个表发生变动——在这个例子中，发生变动的有 RoomType, Room, Address 三张表。正如 [章节 II.2 \[系统结构设计\]](#) 开头已经阐明的那样，算法首先作用到本地内存中，之后才同步到数据库中。如果首先删除 RoomType 中的 type，而由第三个问题 Room 中的房间依然存在；但是 Address 表中既存在引用 roomNO 的外键，也存在引用 type 的外键，这就导致了继续删除 Address 表时会发生冲突并崩溃。

IV.2 解决方案

在 [章节 IV.1 \[问题\]](#) 中提到的问题，我们分别采取了以下解决方案：

1. 问题的根源是：C# 中的 DataTable 数据结构不会自动生成约束关系。虽然数据库的数据初始化是直接初始化到数据库中的，但是从数据库中读取数据到 DataTable 中时，并不会也将数据库中的约束关系读取到 DataTable 中。因此，在创建内存 DataTable 时，还需要手动设置于数据库中相同的约束关系。例如，Address 表中有两个外键约束，那么使用下述代码来模拟这些约束：

```

1 Address.Table.Constraints.Add(new ForeignKeyConstraint(
2     [RoomType.Table.Columns["hotelNO"], RoomType.Table.Columns["type"]],
3     [Address.Table.Columns["hotelNO"], Address.Table.Columns["type"]])
4     { DeleteRule = Rule.Cascade });
5 Address.Table.Constraints.Add(new ForeignKeyConstraint(
6     [Room.Table.Columns["hotelNO"], Room.Table.Columns["roomNO"]],
7     [Address.Table.Columns["hotelNO"], Address.Table.Columns["roomNO"]])
8     { DeleteRule = Rule.Cascade });
  
```

2. 问题的根源是：WinForm 自动生成的代码中没有异常处理。所以，在注意处理异常逻辑时，将这些逻辑尽量放置在自定义函数中。
3. 正如问题所说，虽然在设计之初没有考虑到 type 相关的级联删除，但回过头来看，在以 hotelNO 和 roomNO 为数据库主要属性的情况下，也确实不可能做到级联删除。因此，在实现删除 type 的相关逻辑时，我们会手动删除所有相关房间。

4. 要解决问题很简单：只需要注意操作的顺序即可。例如，在删除 **type** 之前，先删除所有相关房间，再删除 **type** 即可。另外也需要注意：内存表格同步到数据库中时也需要遵循一定的顺序。

IV.3 解决效果

解决以上较为严重的问题之后，并完善一些操作逻辑之后，我们的成品效果如下：

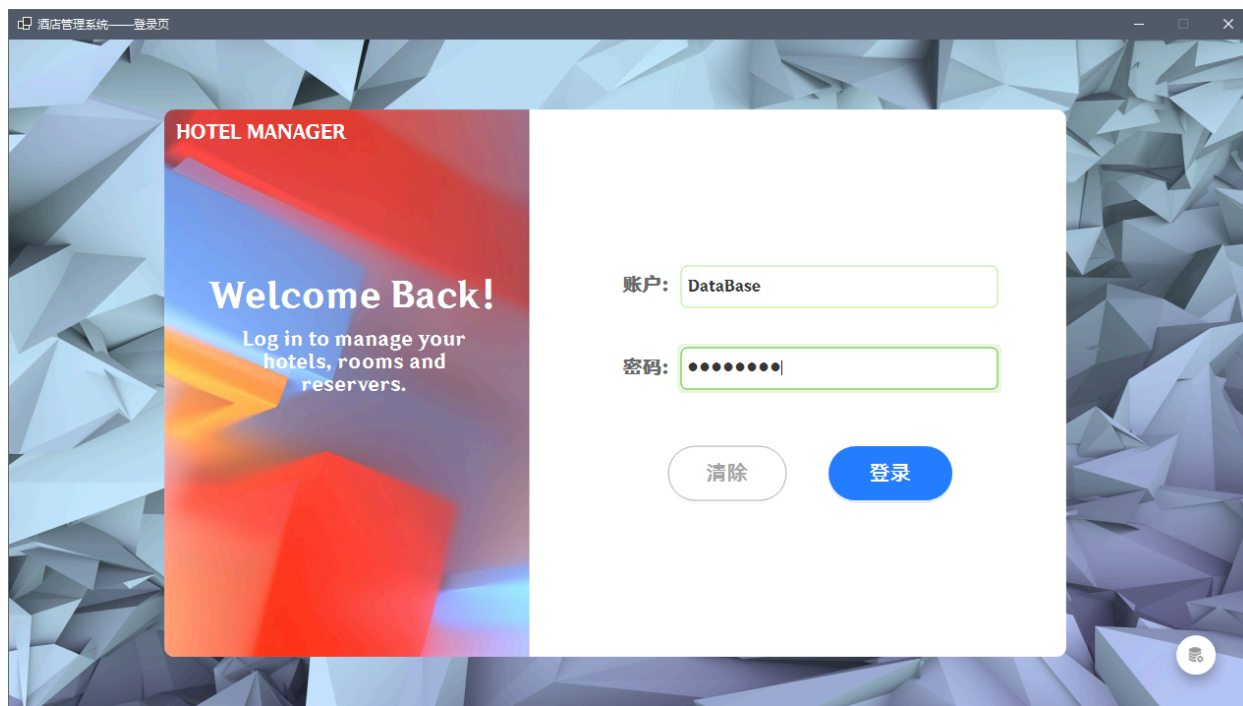


图 8 登录界面

在右下角悬浮按钮处设置数据库连接参数

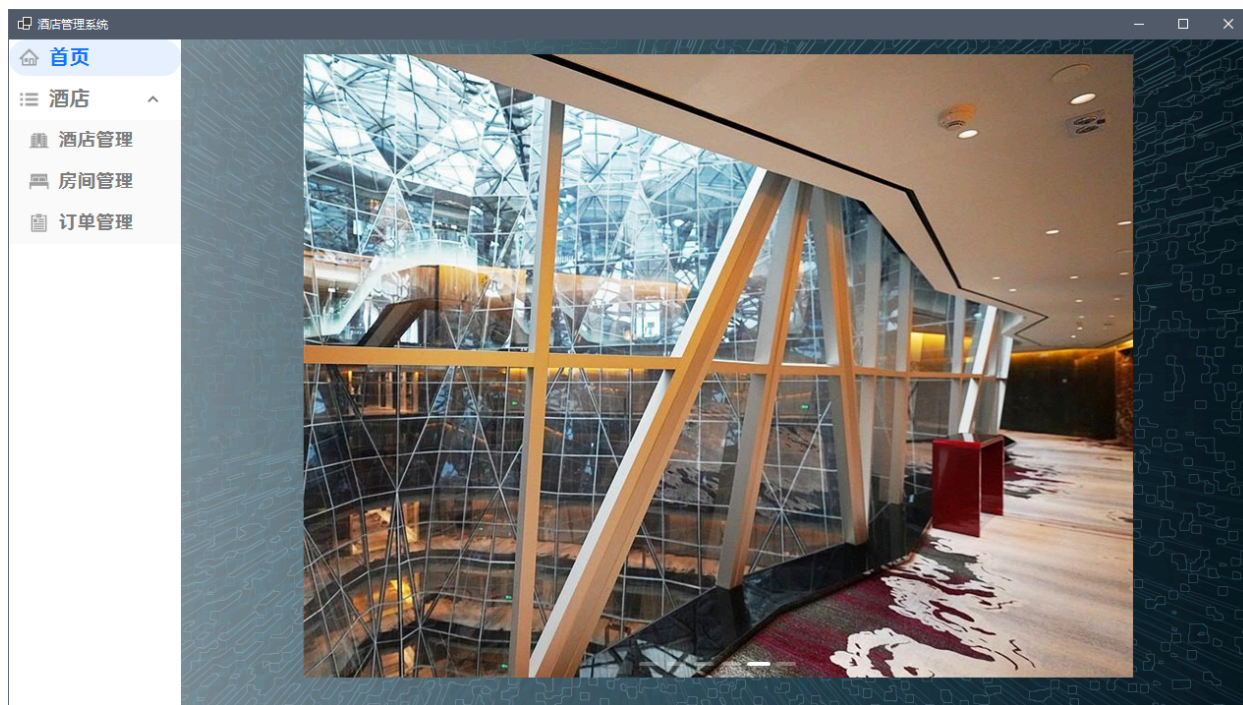


图 9 管理窗口首页



图 10 房间管理页



图 11 输入非法数据时出现提示

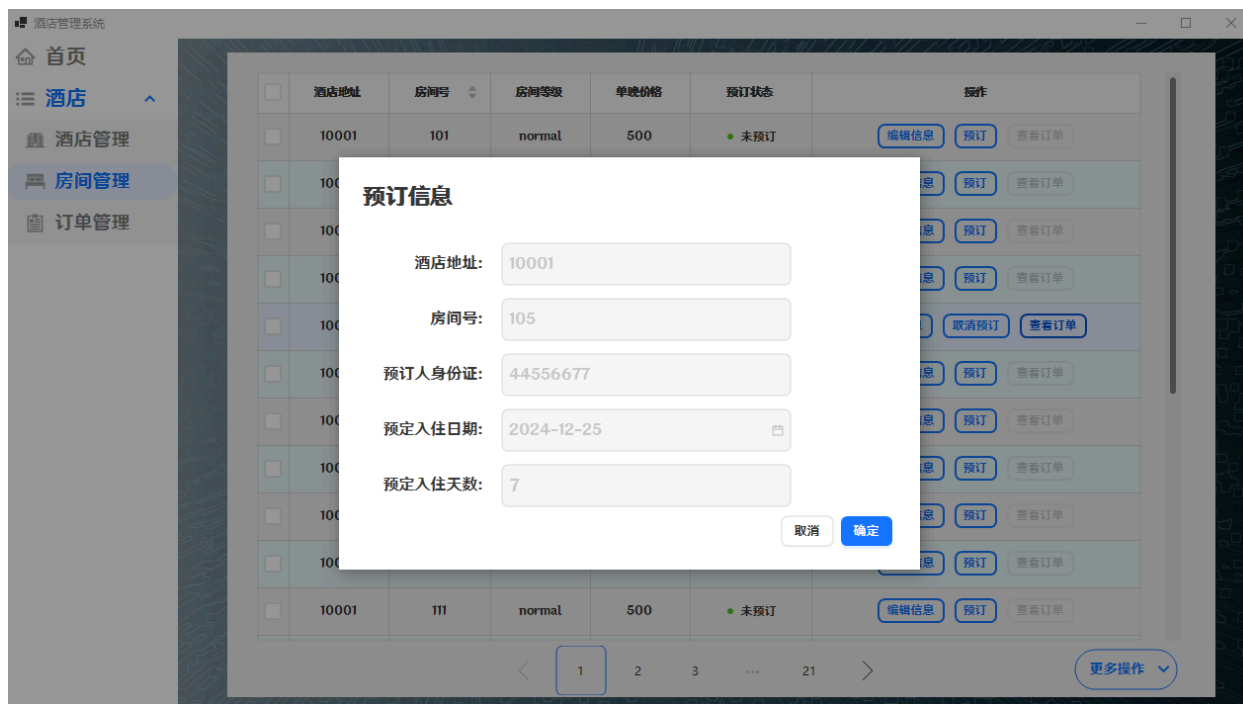


图 12 查看单个房间的订单信息

V 总结

在这个数据库期末项目中，我们权衡了各语言的利弊，最终选定了能够一站式解决前端后端的 C# 语言，并从零开始学习了这门语言。

结合现实中酒店管理具有的要素，确定了管理系统所需要的基本属性，并通过绘制如 [图 1 \[酒店预订管理系统的 E-R 图\]](#) 的 E-R 图，使数据库设计符合 3-NF 范式。

我们采取前端与后端分离的设计方法，按 [章节 II.3 \[功能模块设计\]](#) 所示构建了项目的结构。并按照 [章节 III \[详细设计\]](#) 中示例的项目代码规范，逐步使用 C# 语言从零开始完成整个项目。

最终，我们的成品如 [章节 IV.3 \[解决效果\]](#) 所展现的那样，是一个界面美观大气，空间布局合理，使用体验舒适的桌面应用。同时，具有健壮的异常处理系统，不会因为误操作而崩溃。由于所有更新操作算法都首先修改内存中的副本，成功之后才写回到数据库中([章节 II.2 \[系统结构设计\]](#))，因此本项目也对数据库安全具有一定的保障。

当然，项目依然有值得改进的地方。比如：

1. 一个酒店具有大量房间，作为管理者有时希望能够直达某一房间的管理界面，因此应当设计一个关于房间号的搜索框。订单管理也是如此。
2. 管理者可能并不喜欢被规定的账户 `DataBase` 和密码 `password`，更希望能够自己注册一个账号和密码。为此，可以在登录界面追加一个注册按钮，并在本地创建一个密文加密的文件用以记录新注册的账户密码。
3. 每次登录时都需要配置数据库连接参数。这对于管理者稍显麻烦，可以追加一个记忆功能，记忆管理者上次填写的参数。
4. 为了测试的方便，一些属性的语义与数据类型并不搭配。例如，酒店地址应是一条字符串，但在数据库中具有名字 `hotelNO`，数据类型为 `int`；预订人身份证号使用 `int` 作为数据类型(这导致在预订界面，过长的身份证号会提示非法数据)。此问题的订正是简单的。

VI 附录与指引

该项目的源代码已上传至 [GitHub 仓库](#) (点击蓝字以访问)。同时，应用安装包可以在右侧 `Release` 栏中下载。

- 为了界面的美观，建议在安装应用之后，将安装目录中的 `/Resource/zh-cn.ttf` 字体文件一并手动安装。这个字体文件亦可在仓库的 `HotelSQL/Resources/` 文件夹中寻得。

VII 参考文献

- [1] ©. Microsoft 2024, “C# Language Documentation.” [Online]. Available: <https://learn.microsoft.com/zh-cn/dotnet/csharp/>
- [2] F. F. Jr., “Npgsql - .NET Access to PostgreSQL.” [Online]. Available: <https://www.npgsql.org/doc/index.html>

- [3] T. @EVA-SS, “基于 Ant Design 设计语言的 Winform 界面库.” [Online]. Available: <https://gitee.com/antdui/AntdUI>