

数据结构课程设计

实验报告

 **2024/2025(1)**

实验题目 图书馆书籍管理系统模拟

学生姓名

学生学号

学生班级

任课教师

提交日期

**计算机科学与技术学院**

1. 实验题目和要求

1.1问题描述

在图书馆书籍管理系统中，管理员需要经常对图书信息进行查询和验证，如查询某本书是否在库、验证某本书的借阅状态等。由于图书信息的查询和验证频率很高，系统有必要有效地组织这些图书信息，从而快速查找和验证图书。另外，系统也会经常添加新入库的图书、删除损坏的图书和更新图书的借阅状态等操作，因此，系统必须采用动态结构，在添加、删除或更新后，依然能保证验证过程的快速。请采用相应的数据结构模拟图书馆书籍管理系统，其功能要求包括图书查询、图书状态更新、新书添加和旧书删除等。

1.2基本要求

1. 要求自己编程实现二叉树结构及其相关功能，以存储图书信息，不允许使用标准模板类的二叉树结构和函数。同时要求根据二叉树的变化情况，进行相应的平衡操作，即AVL平衡树操作，四种平衡操作都必须考虑。测试时，各种情况都需要测试，并附上测试截图；
2. 要求采用类的设计思路，不允许出现类以外的函数定义，但允许友元函数。
3. 要求采用多文件方式：.h文件存储类的声明，.cpp文件存储类的实现，主函数main存储在另外一个单独的cpp文件中。如果采用类模板，则类的声明和实现都放在.h文件中。
4. 不强制要求采用类模板；不强制要求采用可视化窗口，但若采用可视化窗口可适当提高考核分数；要求源程序中有相应注释；
5. 要求测试例子要比较详尽，各种极限情况也要考虑到，测试的输出信息要详细易懂，表明各个功能的执行正确；
6. 设计思路

2.1系统总体设计

该图书管理系统提供了用户登录功能，支持管理员和普通用户身份验证。系统中的用户和管理员在图书管理方面具有相同的权限，包括查看、添加、删除和修改图书信息。管理员与普通用户的区别主要体现在登录时的身份验证，管理员登录后可以进入用户管理系统，而普通用户只能访问图书管理系统。

图书数据通过自定义的书籍类进行封装，并利用AVL树进行存储和检索，以保证图书信息的高效查找和操作。图书信息包含图书编号、书名、作者、出版日期及借阅状态等字段。借阅状态的更新也在图书管理系统中得到了处理，用户可查看图书的借阅状态，而管理员可进行更改。

数据文件的存储采用文本文件，所有的图书信息、用户信息均保存在本地文件中，系统支持从文件读取和修改图书及用户信息。系统的操作界面简洁明了，通过Qt实现，确保了用户的良好操作体验。

整体设计充分考虑了图书管理的高效性与数据的持久化管理，确保了系统的稳定运行与高效操作。

**2.2系统功能设计**

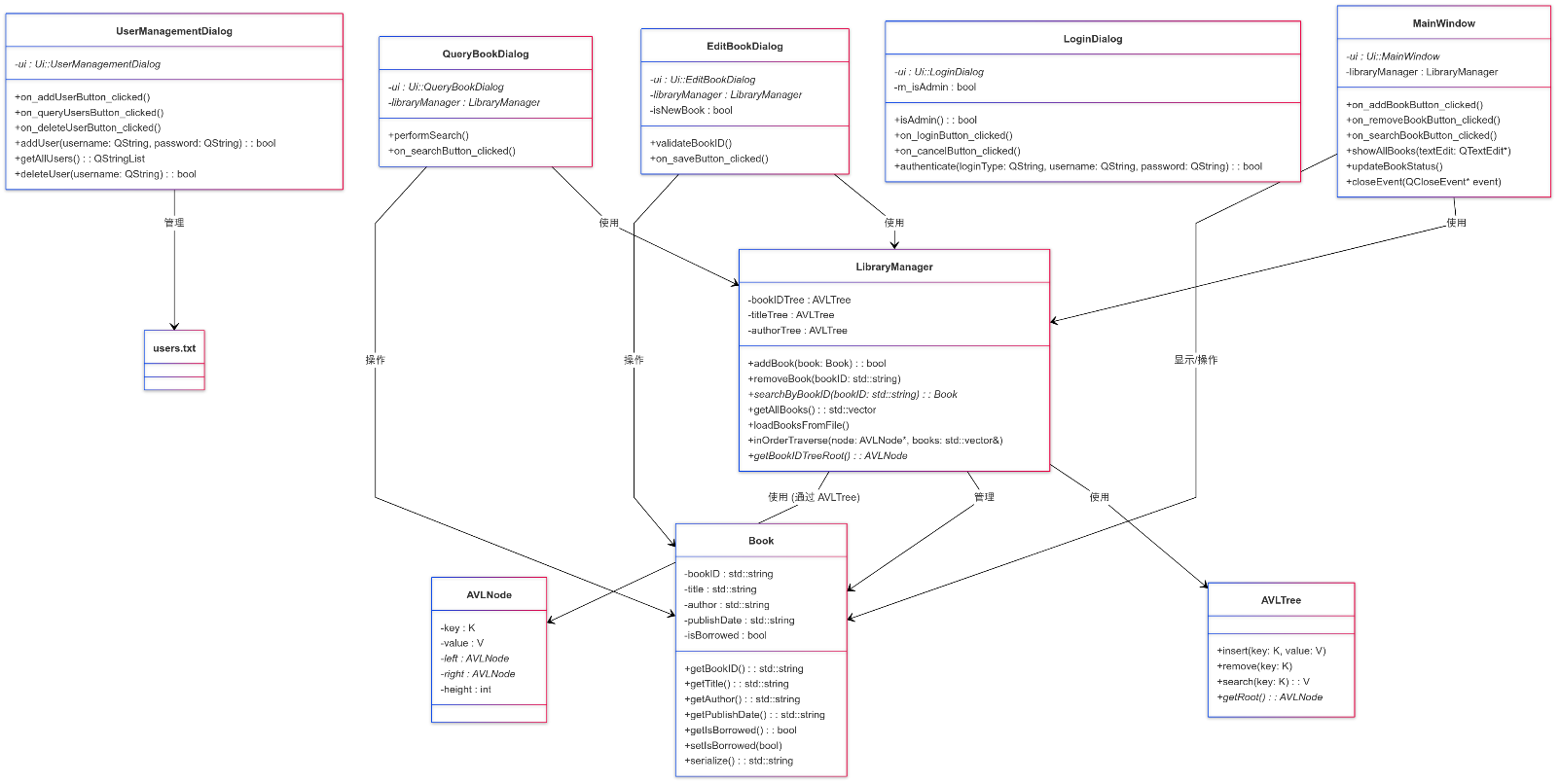
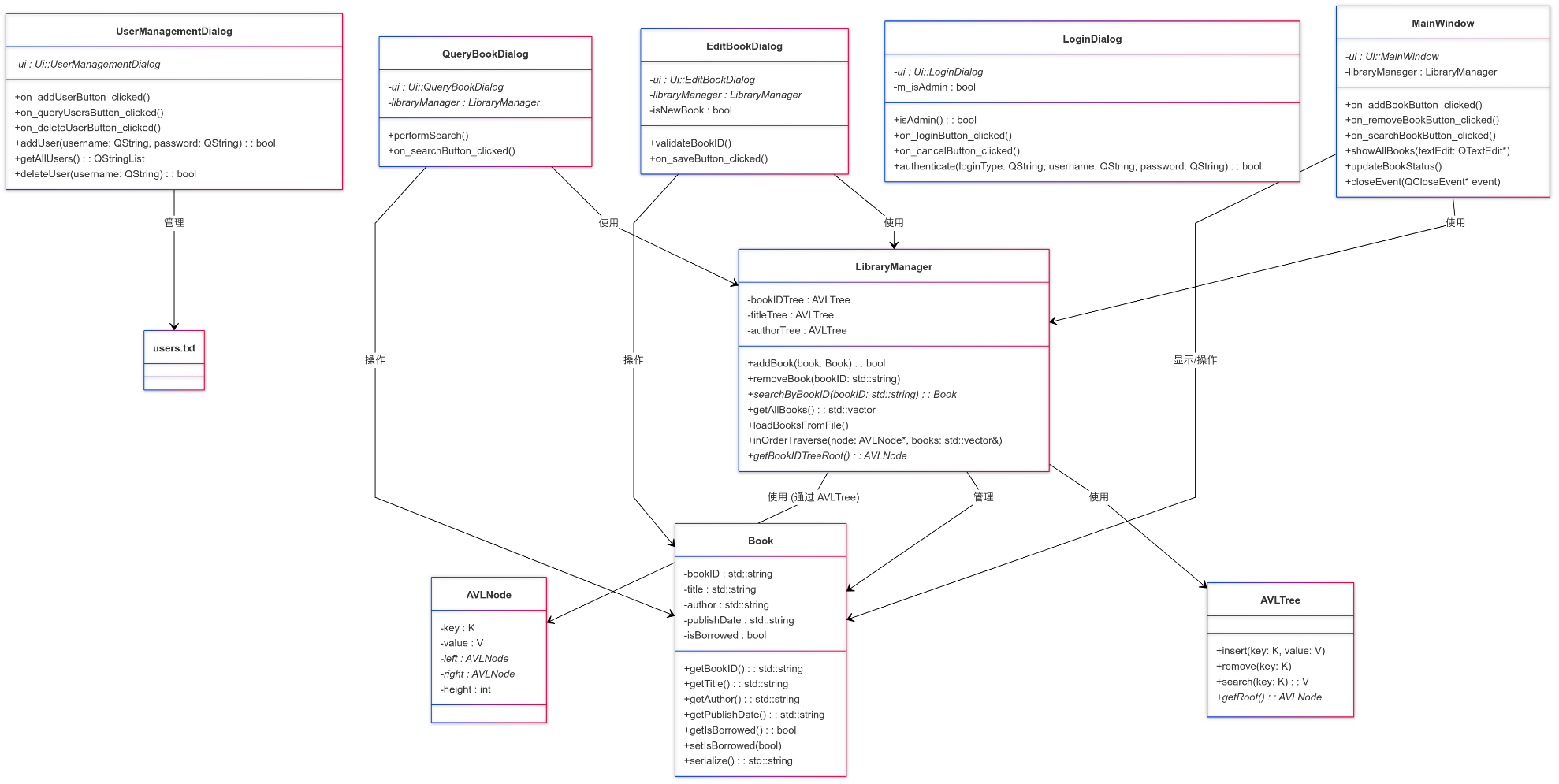
系统的功能设计见下表

|  |  |
| --- | --- |
| **功能** | **说明** |
| 用户登录 | 用户或管理员登录，区分两种账号权限。管理员可进入用户管理系统 |
| 添加用户 | 由管理员给出用户名和密码等信息，完成新用户的添加 |
| 用户查询 | 管理员查询所有用户的信息，展示用户名和其他相关信息 |
| 删除用户 | 管理员输入用户名，删除指定用户信息 |
| 图书查询 | 查看现有的图书信息，包括序号、书名、作者、出版日期等 |
| 图书添加 | 输入图书的相关信息（如书名、作者、出版社等），将新图书添加到系统 |
| 图书删除 | 输入图书编号，删除指定图书 |
| 从文件读入 | 系统启动时自动从文本文件读取所有用户、图书及借阅信息 |
| 写入文件 | 在系统退出时，将所有的用户、图书、借阅状态等信息保存到文本文件中 |
| 借阅图书 | 用户查询并借阅图书，系统更新图书的借阅状态 |
| 归还图书 | 用户归还图书，系统更新图书的借阅状态 |
| 查找图书 | 用户输入图书信息（如书名、作者），系统展示该图书的详细信息 |

表 1 系统功能

* 1. **类的设计**
     1. 类间关系

图 1类间关系



* + 1. AVLNode结构体

|  |  |  |
| --- | --- | --- |
| 成员 | 类型 | 说明 |
| key | TKey | 存储节点的键，用于排序和查找 |
| value | TValue | 存储节点的值 |
| left | AVLNode\* | 指向左子节点的指针 |
| right | AVLNode\* | 指向右子节点的指针 |
| height | int | 节点的高度（以该节点为根的子树的高度） |
| AVLNode构造函数 | AVLNode(const TKey&, const TValue&) | 初始化键、值，左、右子节点为空，高度为1 |

表 2 AVLNode

AVLNode 是一个模板结构，用于定义 AVL 树的节点结构。它使用了 C++ 模板机制，使得该节点结构能够存储不同类型的键（TKey）和值（TValue）。通过模板，AVLNode 可以根据实际需求存储任意类型的数据，从而具有高度的灵活性和可重用性。节点用于表示树中的每一个节点。每个节点包含一个键值对（key 和 value），用于存储和查找数据；两个指针（left 和 right），分别指向左右子节点；以及一个整型值 height，用于记录节点的高度，以保证 AVL 树的自平衡特性。该结构体通过构造函数初始化节点的键值、左右子节点为空指针，并将高度设置为 1。AVL 树通过该结构体构建节点，确保树在插入或删除节点时保持平衡，从而优化查找、插入和删除操作的时间复杂度。

* + 1. AVLTree类

|  |  |
| --- | --- |
| 变量/函数 | 说明 |
| AVLNode\* root | 根节点指针。 |
| 构造函数 | 初始化树的根节点为空。 |
| 析构函数 | 递归销毁所有节点。 |
| getHeight(node) | 获取节点的高度。 |
| getBalanceFactor(node) | 计算节点的平衡因子。 |
| rotateLeft(node) | 左旋操作，修复平衡。 |
| rotateRight(node) | 右旋操作，修复平衡。 |
| rotateLeftRight(node) | 左-右双旋操作。 |
| rotateRightLeft(node) | 右-左双旋操作。 |
| insert(node, key, value) | 插入节点，保持树的平衡。 |
| remove(node, key) | 删除节点，保持树的平衡。 |
| search(key) | 查找特定键对应的值。 |
| findMin(node) | 查找子树中的最小节点。 |
| getRoot() | 获取树的根节点。 |

表 3 AVLTree类

AVLTree 类是一个自平衡的二叉查找树实现，其中包含了多种操作和辅助功能。类中的主要变量是 root，它指向树的根节点，所有的树操作都通过该根节点进行。

**构造与析构函数**：

* AVLTree()：初始化 AVL 树，将 root 设置为 nullptr，创建一个空的树。
* ~AVLTree()：递归销毁树中的所有节点并释放内存，确保树的清理工作。

**核心操作函数**：

* insert(const TKey& key, TValue value)：在树中插入一个新的键值对。此函数会调用递归版本 insert(AVLNode<TKey, TValue>\* node, const TKey& key, TValue value)，确保插入节点后树仍然保持平衡。
* remove(const TKey& key)：删除指定键的节点，并重新平衡树。递归删除操作由 remove(AVLNode<TKey, TValue>\* node, const TKey& key) 实现。
* search(const TKey& key) const：查找并返回指定键的值。递归查找操作通过 search(AVLNode<TKey, TValue>\* node, const TKey& key) 完成。

**平衡调整**：

* getHeight(AVLNode<TKey, TValue>\* node) const：获取节点的高度，用于计算树的平衡因子。
* getBalanceFactor(AVLNode<TKey, TValue>\* node) const：计算并返回节点的平衡因子（左子树高度 - 右子树高度），用于判断是否需要旋转。
* 旋转操作（rotateLeft, rotateRight, rotateLeftRight, rotateRightLeft）通过调整节点位置来确保树的平衡，避免树退化为链表。

**辅助函数**：

* findMin(AVLNode<TKey, TValue>\* node) const：查找并返回树中最小的节点。
* destroy(AVLNode<TKey, TValue>\* node)：递归销毁树的节点，确保内存释放

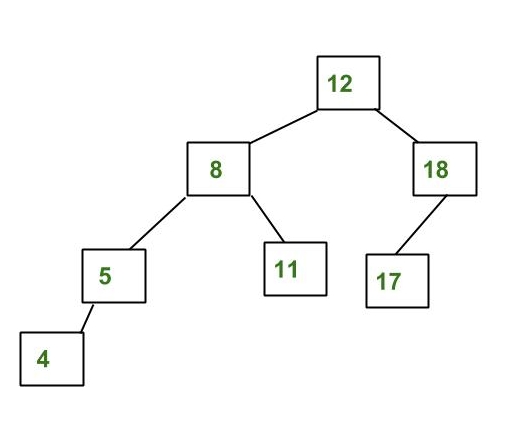


图 2 AVLTree数据结构样例

* + 1. MyUnorderedMap类

|  |  |
| --- | --- |
| **变量/函数** | **说明** |
| buckets | 存储哈希表的桶，每个桶是一个 KeyValuePair 类型的链表（std::vector<KeyValuePair>）。 |
| bucket\_count | 哈希表中桶的数量。 |
| hash\_func | 用于计算键的哈希值的哈希函数，默认使用 std::hash<Key>。 |
| key\_equal | 用于比较键是否相等的函数，默认使用 std::equal\_to<Key>。 |
| element\_count | 当前哈希表中元素的数量（键值对的数量）。 |
| bucket\_index(const Key& key) | 根据 key 计算哈希桶的索引位置。 |
| MyUnorderedMap(size\_t) | 构造函数，初始化哈希表的桶数量和其他成员。 |
| insert(const Key&, const Value&) | 插入或更新键值对。如果键已存在则更新值，否则插入新键值对。 |
| find(const Key&) | 查找键 key，如果找到返回指向对应值的指针，否则返回 nullptr。 |
| erase(const Key&) | 删除指定 key 的键值对，如果删除成功返回 true，否则返回 false。 |
| size() | 返回哈希表中的元素数量。 |
| bucket\_size() | 返回哈希表中桶的数量。 |
| load\_factor() | 计算当前哈希表的装载因子（元素数量 / 桶数量）。 |
| rehash(size\_t) | 扩容哈希表，重新分配桶数量并重新计算每个元素的桶索引。 |
| getAllKeyValuePairs() | 获取所有键值对，以 std::vector<std::pair<Key, Value>> 形式返回。 |
| operator[](const Key&) | 用 [] 操作符访问哈希表。如果 key 存在则返回对应的值，否则插入一个默认构造的值并返回。 |
| at(const Key&) | 查找指定 key 的值，如果键不存在则抛出 std::out\_of\_range 异常。 |
| const at(const Key&) | at 的常量版本，返回不可修改的值引用。如果键不存在则抛出 std::out\_of\_range 异常。 |

表 4 MyUnorderedMap类

MyUnorderedMap 是一个自定义实现的哈希表类，它采用了开放寻址法（通过哈希桶）来存储键值对。这个类提供了一个高效的键值映射容器，支持基本的插入、查找、删除和更新操作，且能够在元素数量过多时自动扩容以保持较好的性能。设计目标是模仿 C++ 标准库中的 std::unordered\_map，并在此基础上提供灵活性以满足不同需求。

**类的设计理念与功能**

在 MyUnorderedMap 中，数据通过哈希函数计算得到的索引分布在一个动态大小的桶数组中，每个桶内部存储一个链表形式的 KeyValuePair，这使得该容器能够在平均常数时间复杂度（O(1)）内完成大部分操作。哈希表的核心设计包括对 key 的哈希值计算、冲突解决以及动态扩容等关键功能。

类的核心操作是 insert、find、erase 和 operator[]。其中，insert 用于插入新的键值对，或者在键已存在时更新其对应的值；find 用于查找给定键的值；erase 用于删除指定键的键值对；operator[] 提供了通过 [] 运算符访问值的能力，如果键不存在，则会自动插入一个默认构造的值。

**哈希表管理与扩容**

哈希表中的桶数量是可调的，初始化时可以指定桶的数量，默认情况下使用 8 个桶。随着元素数量的增加，负载因子（元素数与桶数的比值）逐渐增大，当负载因子超过 1 时，哈希表会自动扩容。这通过 rehash 函数实现，rehash 会增加桶的数量，并重新将元素散列到新的桶中。这一设计确保了即使在数据量大幅增加时，哈希表仍然能够保持较高的操作效率。

**键值对存储与访问**

哈希表中的每个元素是一个 KeyValuePair 对象，它包含一个键（key）和一个与之关联的值（value）。哈希表通过 bucket\_index 函数将键映射到桶的索引位置。每个桶内部是一个链表，当发生哈希冲突时，多个键值对会被存储在同一个桶内。通过 key\_equal 函数，哈希表能够判断两个键是否相等，从而处理冲突。

在访问元素时，operator[] 允许用户通过键直接访问对应的值。如果键不存在，operator[] 会插入一个默认构造的值，确保操作的无缝进行。而 at 函数则用于获取指定键的值，如果键不存在，会抛出 std::out\_of\_range 异常。

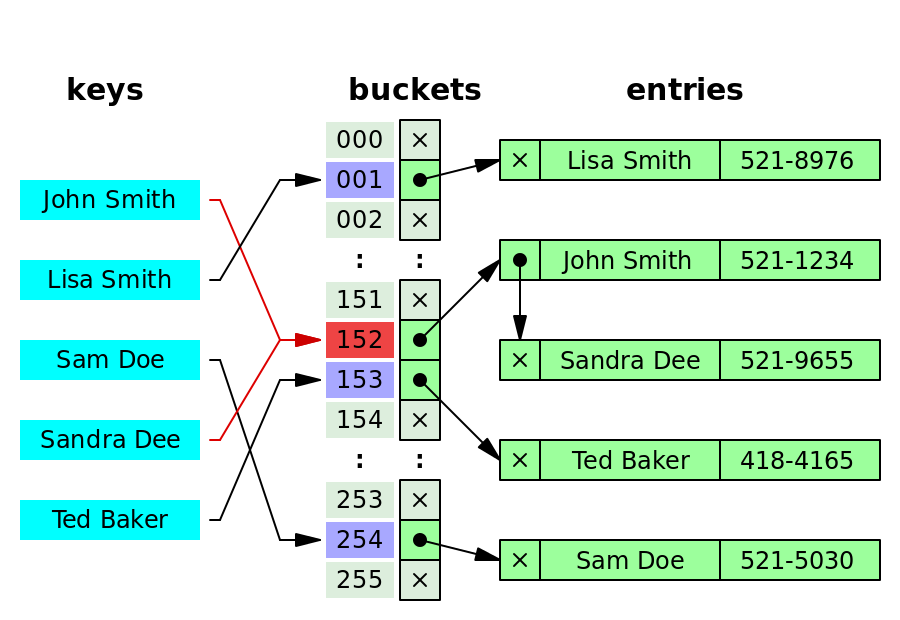


图 3 2.3.4 MyUnorderedMap数据结构逻辑，与std::unodered\_map相似

* + 1. Book类

|  |  |
| --- | --- |
| 变量/函数 | 说明 |
| std::string bookID | 图书编号。 |
| std::string title | 图书标题。 |
| std::string author | 图书作者。 |
| std::string publishDate | 出版日期。 |
| bool isBorrowed | 借阅状态（已借出/可借）。 |
| 构造函数 | 初始化图书的基本信息。 |
| serialize() | 将图书信息转为字符串。 |
| deserialize(data) | 从字符串还原图书对象。 |
| getBookID() | 获取图书编号。 |
| getTitle() | 获取图书标题。 |
| getAuthor() | 获取图书作者。 |
| getPublishDate() | 获取出版日期。 |
| getIsBorrowed() | 获取借阅状态。 |
| setIsBorrowed(bool) | 设置借阅状态。 |

表 5 Book 类

Book 类是图书管理系统中的核心类，用于表示每本图书的基本信息。它包含书籍的唯一标识符 bookID、书名 title、作者 author、出版日期 publishDate 和借阅状态 isBorrowed 等属性。该类提供了用于序列化和反序列化的功能，以及与 AVL 树的兼容操作。

**主要功能说明**：

1. **序列化** (serialize)： serialize 方法将 Book 类的实例转换为字符串形式，方便存储和传输。返回的字符串包括各个字段的信息，字段之间用逗号分隔。isBorrowed 状态用 "1" 或 "0" 表示，分别代表已借出或未借出。例如："1234,The Great Gatsby,F. Scott Fitzgerald,1925-04-10,0"该方法将对象的各个字段按照一定的格式转化为字符串，并返回该字符串。
2. **反序列化** (deserialize)： deserialize 方法用于将一个格式化的字符串转换为 Book 对象。方法通过查找逗号分隔符，将字符串切分为不同字段，并为每个字段赋值。最后，根据字符串的最后一个字段确定图书是否被借出，生成并返回一个 Book 实例。例如：

Book book = Book::deserialize("1234,The Great Gatsby,F. Scott Fitzgerald,1925-04-10,0");

该方法通过解析字符串并将数据赋值给 Book 类的成员变量，恢复出完整的 Book 对象。

1. **操作符重载**： operator< 和 operator== 是对 Book 类的比较操作符重载，用于在 AVL 树等数据结构中进行排序和查找。
   * operator<：根据 bookID 来比较两本书的大小关系。AVL 树使用这个操作符来决定节点的顺序。
   * operator==：根据 bookID 来判断两本书是否相等。
2. **成员变量**：
   * bookID：书籍的唯一标识符。
   * title：书籍的标题。
   * author：书籍的作者。
   * publishDate：书籍的出版日期。
   * isBorrowed：标记书籍是否被借出，布尔值。
     1. LibraryManager 类

|  |  |
| --- | --- |
| 变量/函数 | 说明 |
| AVLTree bookIDTree | 按图书编号存储的 AVL 树。 |
| AVLTree titleTree | 按图书标题存储的 AVL 树。 |
| AVLTree authorTree | 按图书作者存储的 AVL 树。 |
| 构造函数 | 初始化图书管理器并加载数据。 |
| 析构函数 | 销毁所有图书对象。 |
| addBook(book) | 添加图书到管理系统。 |
| removeBook(bookID) | 删除指定编号的图书。 |
| searchBooks(keys) | 模糊查找图书。 |
| searchByAuthor(author) | 根据作者模糊查找图书。 |
| getAllBooks() | 获取所有图书列表。 |
| loadBooksFromFile(filename) | 从文件加载图书数据。 |
| inOrderTraverse() | 中序遍历 AVL 树。 |
| getBookIDTreeRoot() | 获取图书编号 AVL 树的根节点。 |

表 6 LibraryManager类

LibraryManager 类是图书管理系统的核心组件之一，负责管理图书的增、删、查等操作。它通过 AVL 树来实现图书信息的高效存储和查找，并支持图书数据的持久化处理（从文件加载和写入）。该类不仅能够管理图书的基本信息，还能够进行基于书籍 ID、书名、作者的查找操作。

**主要功能说明：**

1. **构造函数和析构函数**：
   * LibraryManager()：构造函数负责初始化图书管理器，并通过 loadBooksFromFile() 方法加载已有的图书信息（如果存在）。
   * ~LibraryManager()：析构函数负责清理图书对象，避免内存泄漏。它通过调用 getAllBooks() 方法获取所有图书并逐个删除。
2. **添加图书**：
   * addBook(const Book& book)：该方法首先检查图书编号（bookID）是否已存在，若存在则返回 false。如果不存在，创建一个 Book 对象并将其插入到三个 AVL 树中：按 bookID、title 和 author 排序。此处为预留功能，预留AVL精准查找标题和作者。
3. **删除图书**：
   * removeBook(const std::string& bookID)：通过 bookID 查找图书，如果图书存在，则从所有 AVL 树中删除对应的节点，并释放 Book 对象的内存。
4. **按书籍 ID 查找图书**：
   * searchByBookID(const std::string& bookID) const：该方法通过 bookIDTree 查找图书，返回匹配的 Book 对象。
5. **查找图书**：
   * std::vector<Book\*> searchBooks(const std::string& query) const：该方法通过倒排索引，进行模糊匹配，查找书名、作者中包含给定关键词的图书，并返回匹配结果。
6. **获取所有图书**：
   * getAllBooks() const：该方法通过中序遍历 bookIDTree 获取所有图书，并将其存储在一个 std::vector<Book\*> 容器中。
7. **中序遍历**：
   * inOrderTraverseHelper()：这是一个递归的辅助函数，用于进行中序遍历，并将遍历到的每本图书添加到 std::vector<Book\*> 中。这个方法的核心作用是确保图书按照 bookID 排序。
8. **加载图书信息**：
   * loadBooksFromFile(const std::string& filename)：该方法从指定的文件中加载图书信息。文件中的每行包含图书的 bookID、title、author、publishDate 和借阅状态（isBorrowed）。如果文件无法打开或是第一次运行程序，它会输出提示信息。对于每一行，方法会创建一个 Book 对象并调用 addBook() 方法将其添加到图书库。
     1. LoginDialog类

|  |  |
| --- | --- |
| 变量/函数 | 说明 |
| bool m\_isAdmin | 是否为管理员用户标志。 |
| 构造函数 | 初始化登录窗口。 |
| on\_loginButton\_clicked() | 登录按钮的槽函数，验证用户信息。 |
| on\_cancelButton\_clicked() | 取消按钮的槽函数，关闭窗口。 |
| authenticate(loginType, username, password) | 验证用户名和密码。 |
| isAdmin() | 返回当前用户是否为管理员。 |

表 7 LoginDialog类

LoginDialog实现了一个登录对话框的功能，允许用户通过用户名和密码进行身份验证。该对话框支持两种登录类型：管理员和用户。

**流程简述：**

1. **构造函数**：
   * 在构造函数中，ui->setupUi(this) 初始化了界面，并设置窗口标题为 "登录"。
   * 默认选择登录类型为“用户”，即 ui->loginTypeComboBox->setCurrentIndex(1)。
2. **登录按钮处理** (on\_loginButton\_clicked)：
   * 用户输入用户名和密码后，点击登录按钮时，程序首先检查输入框是否为空。
   * 如果用户名或密码为空，弹出警告框提示用户输入完整信息。
   * 然后调用 authenticate 函数验证用户名和密码是否正确。
3. **身份验证** (authenticate):
   * 根据选择的登录类型（管理员或用户），程序选择不同的账户文件（admins.txt 或 users.txt）。
   * 如果文件不存在，程序会自动创建默认的管理员账户（admin, admin123）或空的用户文件。
   * 打开文件后，逐行读取文件内容，进行用户名和密码的比对。如果找到了匹配项，则返回验证通过。
   * 如果未找到匹配项，则登录失败。
4. **登录成功或失败**：
   * 如果身份验证通过，弹出登录成功的提示框，并设置 m\_isAdmin 标志（区分管理员和用户身份），然后关闭对话框。
   * 如果验证失败，弹出警告框提示用户名或密码错误。
5. **取消按钮处理** (on\_cancelButton\_clicked)：
   * 点击取消按钮时，关闭对话框并返回 QDialog::Rejected。
6. **管理员身份判断**：
   * 提供了 isAdmin() 函数，用户可以通过它判断当前登录的是否为管理员。

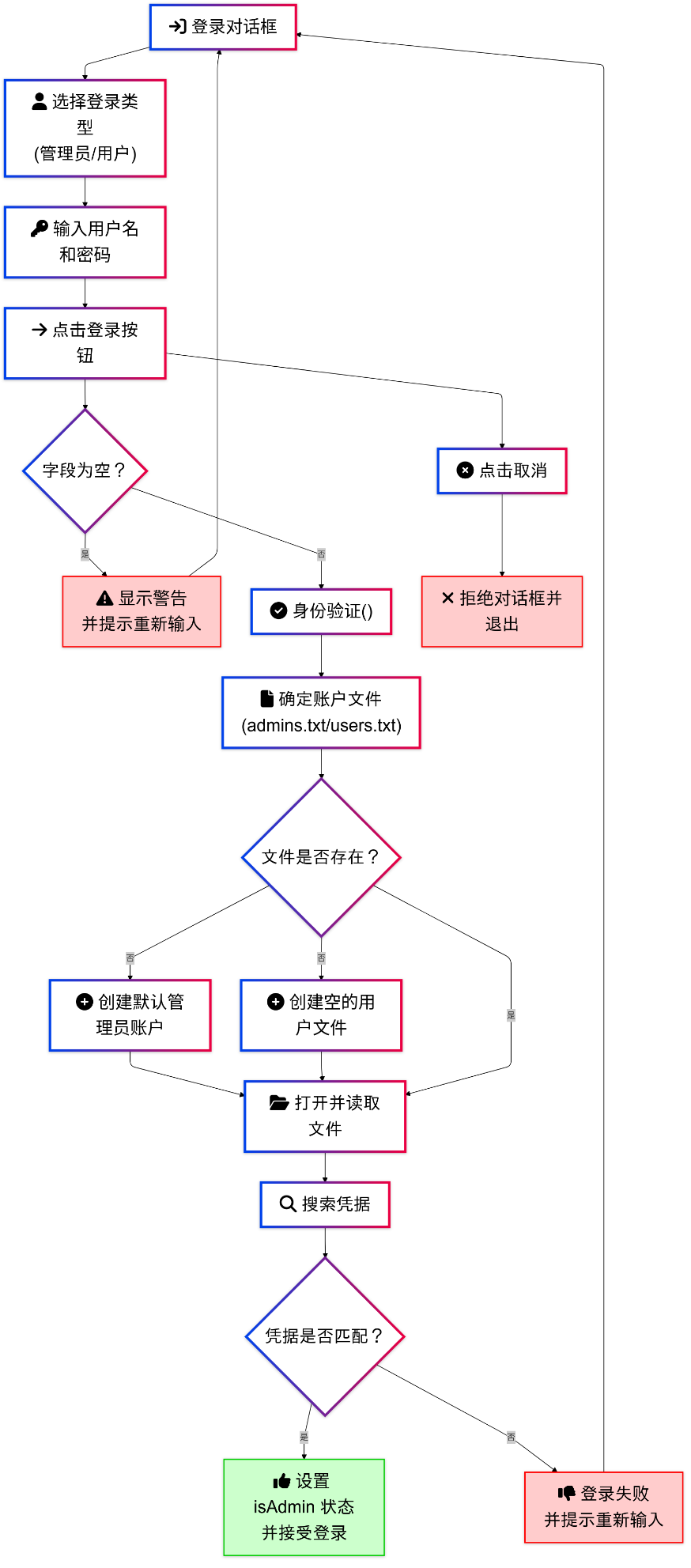


图 4 LoginDialog类流程图

**关键代码：**

if (username.isEmpty() || password.isEmpty()) {

QMessageBox::warning(this, "输入错误", "用户名和密码不能为空！");

return;

}

if (authenticate(loginType, username, password)) {

QMessageBox::information(this, "登录成功", "欢迎，" + username + "！");

m\_isAdmin = (loginType == "管理员");

accept(); // 登录成功，关闭对话框

} else {

QMessageBox::warning(this, "登录失败", "用户名或密码错误！");

}

bool LoginDialog::authenticate(const QString &loginType, const QString &username, const QString &password)

{

QString fileName = (loginType == "管理员") ? "admins.txt" : "users.txt";

QFile file(fileName);

if (!file.exists()) {

// 如果文件不存在，则创建默认账户

createDefaultAccount(loginType, file);

}

// 文件打开和用户名密码验证

}

该代码实现了一个简单而有效的登录系统，支持管理员和普通用户的身份验证，并能在文件不存在时自动创建相关的账户文件。

* + 1. MainWindow类

|  |  |
| --- | --- |
| 变量/函数 | 说明 |
| LibraryManager\* libraryManager | 图书管理器指针。 |
| 构造函数 | 初始化主窗口，设置界面。 |
| on\_addBookButton\_clicked() | 添加图书按钮的槽函数。 |
| on\_removeBookButton\_clicked() | 删除图书按钮的槽函数。 |
| on\_searchBookButton\_clicked() | 搜索图书按钮的槽函数。 |
| showAllBooks(textEdit) | 显示所有图书信息。 |
| updateBookStatus() | 更新图书的借阅状态。 |
| saveBooksToFile() | 保存图书信息到文件。 |
| closeEvent(event) | 窗口关闭事件，自动保存数据。 |

表 8 MainWindow类

MainWindow 类是图书管理系统的主窗口，负责与用户进行交互并协调各种图书管理功能。该类继承自 QMainWindow，并通过与 LibraryManager 对象的协作，实现图书的添加、删除、查询、显示及借阅状态的更新等操作。

**主要功能**

1. **初始化与连接信号槽**： 在构造函数中，MainWindow 初始化用户界面并连接各个按钮的点击信号到相应的槽函数。例如，当用户点击“显示所有图书”按钮时，会调用 showAllBooks 方法展示所有图书信息。

connect(ui->showAllBooksButton, &QPushButton::clicked, [this]() { showAllBooks(ui->allBooksTextEdit); });

1. **添加图书**： 当用户点击“添加图书”按钮时，on\_addBookButton\_clicked 槽函数会弹出一个 EditBookDialog 对话框，允许用户输入新图书的信息并将其添加到系统中。

void MainWindow::on\_addBookButton\_clicked() {

EditBookDialog dialog(this, libraryManager, true);

dialog.exec();

}

1. **删除图书**： 用户点击“删除图书”按钮后，系统会弹出一个输入框，要求用户输入要删除的图书编号。确认删除后，调用 LibraryManager 的 removeBook 方法执行删除操作，并反馈删除结果。
2. **搜索图书**： 通过“搜索图书”按钮，用户可以打开 QueryBookDialog 对话框，根据书名或作者等条件查询图书信息。
3. **显示所有图书**： showAllBooks 方法通过调用 LibraryManager 的 getAllBooks 获取所有图书，并将其格式化后显示在界面的文本编辑区。
4. **更新借阅状态**： 用户可以通过“更新借阅状态”按钮，输入图书编号并选择新的借阅状态（已借出或可借），系统随后更新相应图书的状态。
5. **保存图书信息**： 在窗口关闭时，closeEvent 方法会自动调用 saveBooksToFile，将当前所有图书的信息保存到文本文件中，确保数据的持久化。

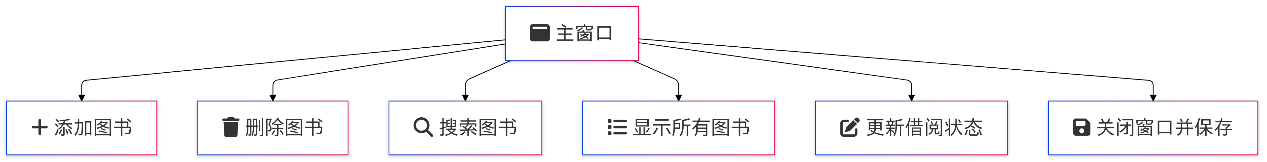


图 5 MainWindow类流程图

* + 1. EditBookDialog类

EditBookDialog 类是图书管理系统中的一个对话框类，负责处理图书的添加与编辑操作。该类继承自 QDialog，通过与 LibraryManager 的交互，实现图书信息的录入、验证和保存。EditBookDialog 支持两种模式：添加新图书和编辑现有图书，根据不同的操作模式设置窗口标题和行为。

**主要功能流程**

1. **初始化对话框**： 在构造函数中，EditBookDialog 初始化用户界面并根据操作模式设置窗口标题。例如，若是添加新图书，则标题为“添加图书”；若是编辑图书，则标题为“编辑图书”。

this->setWindowTitle(isNewBook ? "添加图书" : "编辑图书");

1. **实时验证图书编号**： 当用户在图书编号输入框中输入内容时，validateBookID 函数会被调用，用于检查该编号是否已存在（仅在添加新书时）。如果编号重复，输入框边框会变为红色，以提示用户。

connect(ui->bookIDLineEdit, &QLineEdit::textChanged, this, &EditBookDialog::validateBookID);

1. **保存图书信息**： 用户填写完图书信息后，点击“保存”按钮会触发 on\_saveButton\_clicked 槽函数。该函数首先验证所有必填字段是否已填写，随后根据操作模式（添加或编辑）执行相应的逻辑：
   * **添加新书**：检查图书编号是否存在，若不存在则调用 LibraryManager 的 addBook 方法添加新书。
   * **编辑图书**：调用 LibraryManager 的 removeBook 和 addBook 方法更新图书信息。（预留功能）

if (isNewBook) {

libraryManager->addBook(updatedBook);

} else {

libraryManager->removeBook(bookID.toStdString());

libraryManager->addBook(updatedBook);

}//预留功能

1. **关闭对话框**： 保存成功后，弹出信息提示框，并关闭对话框。

QMessageBox::information(this, "成功", "图书信息已保存！");

accep

* + 1. QueryBookDialog类

|  |  |
| --- | --- |
| 变量/函数 | 说明 |
| LibraryManager\* libraryManager | 图书管理器指针。 |
| 构造函数 | 初始化查询窗口。 |
| 析构函数 | 释放查询窗口资源。 |
| on\_searchButton\_clicked() | 搜索按钮的槽函数，执行查询。 |
| performSearch() | 执行图书搜索操作。 |

表 9 QueryBookDialog 类

QueryBookDialog 类是图书管理系统中的一个对话框类，负责处理图书的查询操作。该类继承自 QDialog，通过与 LibraryManager 的交互，实现基于图书编号、书名、作者及借阅状态的多条件搜索。用户可以输入任意组合的搜索条件，系统将根据输入条件筛选并显示符合条件的图书信息。

**主要功能流程**

1. **初始化对话框**：
   * 在构造函数中，QueryBookDialog 初始化用户界面，并设置窗口标题为“查询图书信息”。
   * 连接“搜索”按钮的点击事件到槽函数 on\_searchButton\_clicked，当用户点击搜索按钮时触发搜索操作。
2. **执行搜索操作**：
   * 用户在对话框中输入一个或多个搜索条件（图书编号、书名、作者）并选择借阅状态（全部、已借出、可借）。
   * 点击“搜索”按钮后，performSearch 方法被调用，系统根据输入的条件筛选图书：
     + **图书编号搜索**：如果输入了图书编号，系统直接通过 LibraryManager 的 searchByBookID 方法查找对应图书。这里使用AVL树进行搜索，时间复杂度低。
     + **书名和作者模糊搜索**：如果输入了书名或作者，系统通过 searchByTitle 和 searchByAuthor 方法进行模糊匹配，筛选包含关键字的图书。这里使用遍历方法查询，较慢。预留了使用AVL树精准查询的方法。未来也可以拓展更多查询方法。
     + **借阅状态筛选**：根据选择的借阅状态进一步过滤结果。
   * 最终，系统将符合条件的图书信息显示在对话框的文本编辑区域中。如果没有找到符合条件的图书，系统会提示用户“未找到符合条件的图书”。
3. **用户反馈**：
   * 如果用户未输入任何搜索条件，系统会提示“请至少输入一个搜索条件！”。
   * 搜索成功后，显示符合条件的图书信息；搜索失败则提示“未找到符合条件的图书。”。

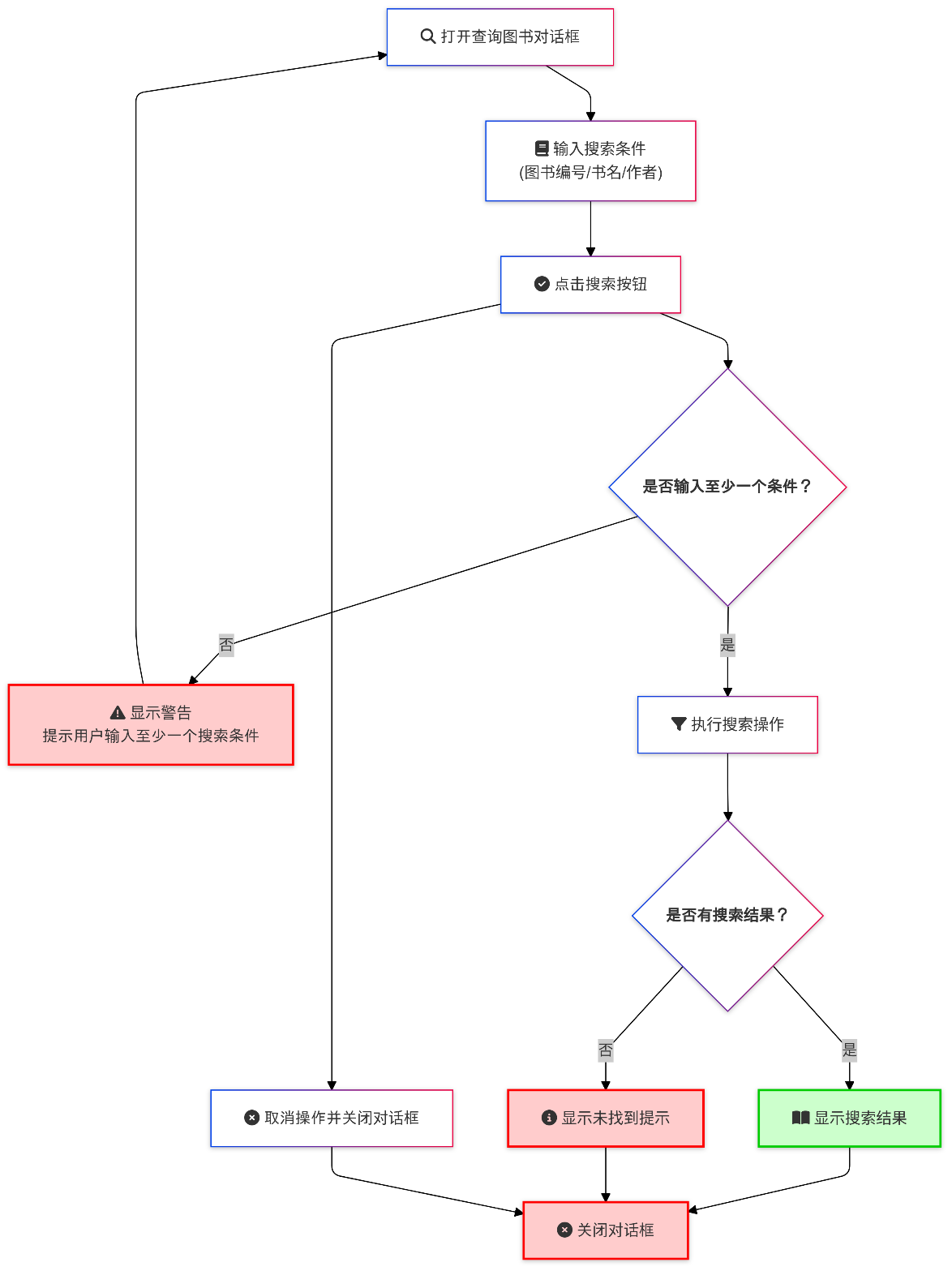


图 6 QueryBookDialog 类流程图

* + 1. UserManagementDialog类

|  |  |
| --- | --- |
| 变量/函数 | 说明 |
| 构造函数 | 初始化用户管理窗口。 |
| 析构函数 | 释放窗口资源。 |
| on\_addUserButton\_clicked() | 添加用户按钮的槽函数。 |
| on\_deleteUserButton\_clicked() | 删除用户按钮的槽函数。 |
| on\_queryUsersButton\_clicked() | 查询用户列表按钮的槽函数。 |
| addUser(username, password) | 添加新用户到文件。 |
| deleteUser(username) | 删除指定用户名的用户。 |
| getAllUsers() | 获取所有用户列表。 |

表 10 UserManagementDialog 类

UserManagementDialog 是一个用户管理对话框类，主要提供以下功能：

1. **添加用户（addUser）**： 管理员输入用户名和密码后，系统验证输入是否为空，并检查用户名是否已存在。若验证通过，用户信息将保存到 users.txt 文件中。
2. **查询用户（getAllUsers）**： 点击“查询用户”按钮，系统从 users.txt 文件中读取所有用户信息，并在界面中显示。若无用户记录，提示“没有用户记录”。
3. **删除用户（deleteUser）**： 用户输入要删除的用户名后，系统验证用户名是否存在。若存在，则从 users.txt 文件中移除该用户。

**文件交互**

* 用户信息持久化存储在 users.txt 文件中。
* 添加用户时检查文件中是否存在相同用户名。
* 删除用户时重新写回所有未删除的用户信息。

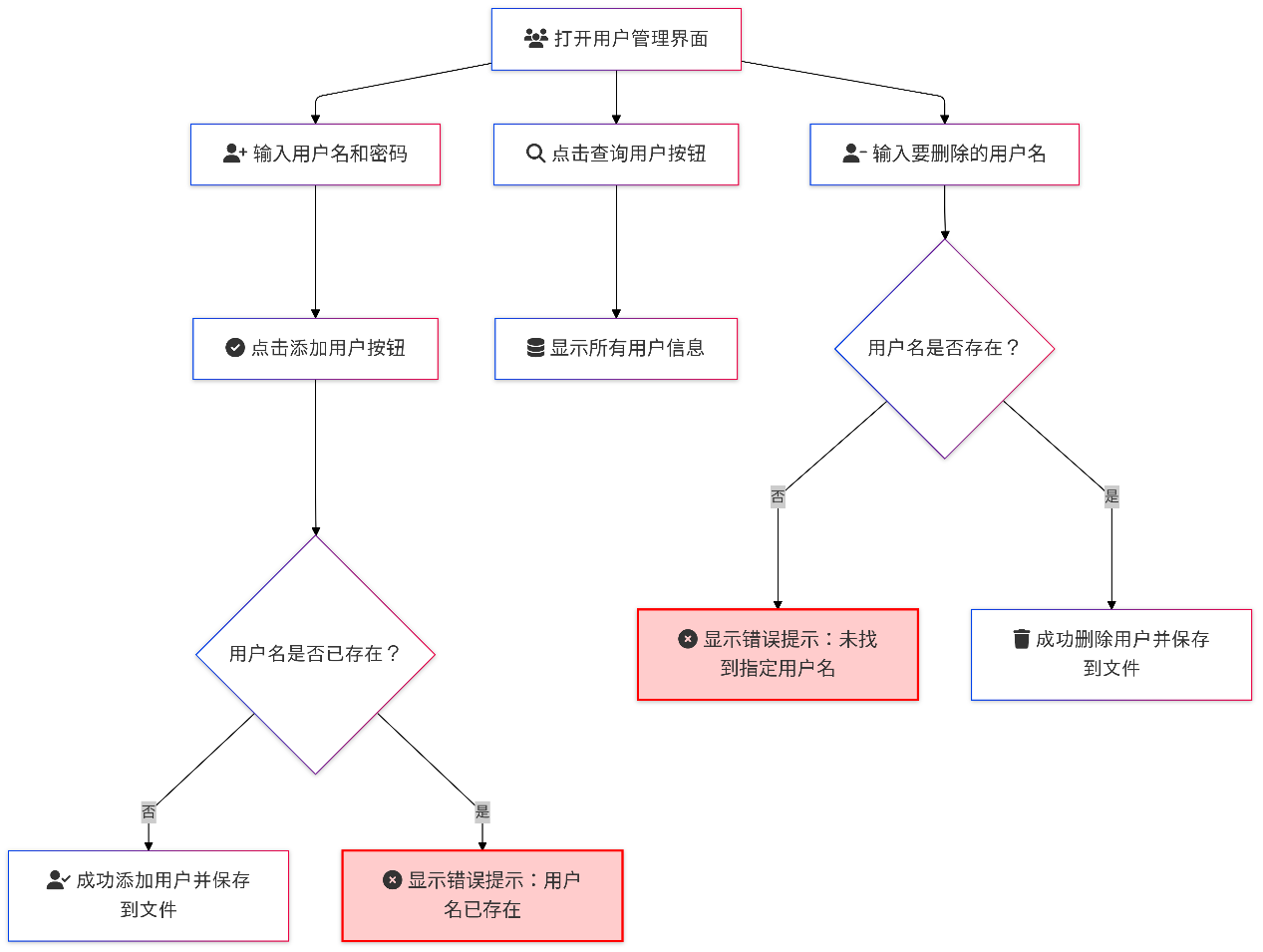


图 7 UserManagementDialog 类流程图

* 1. 主程序设计

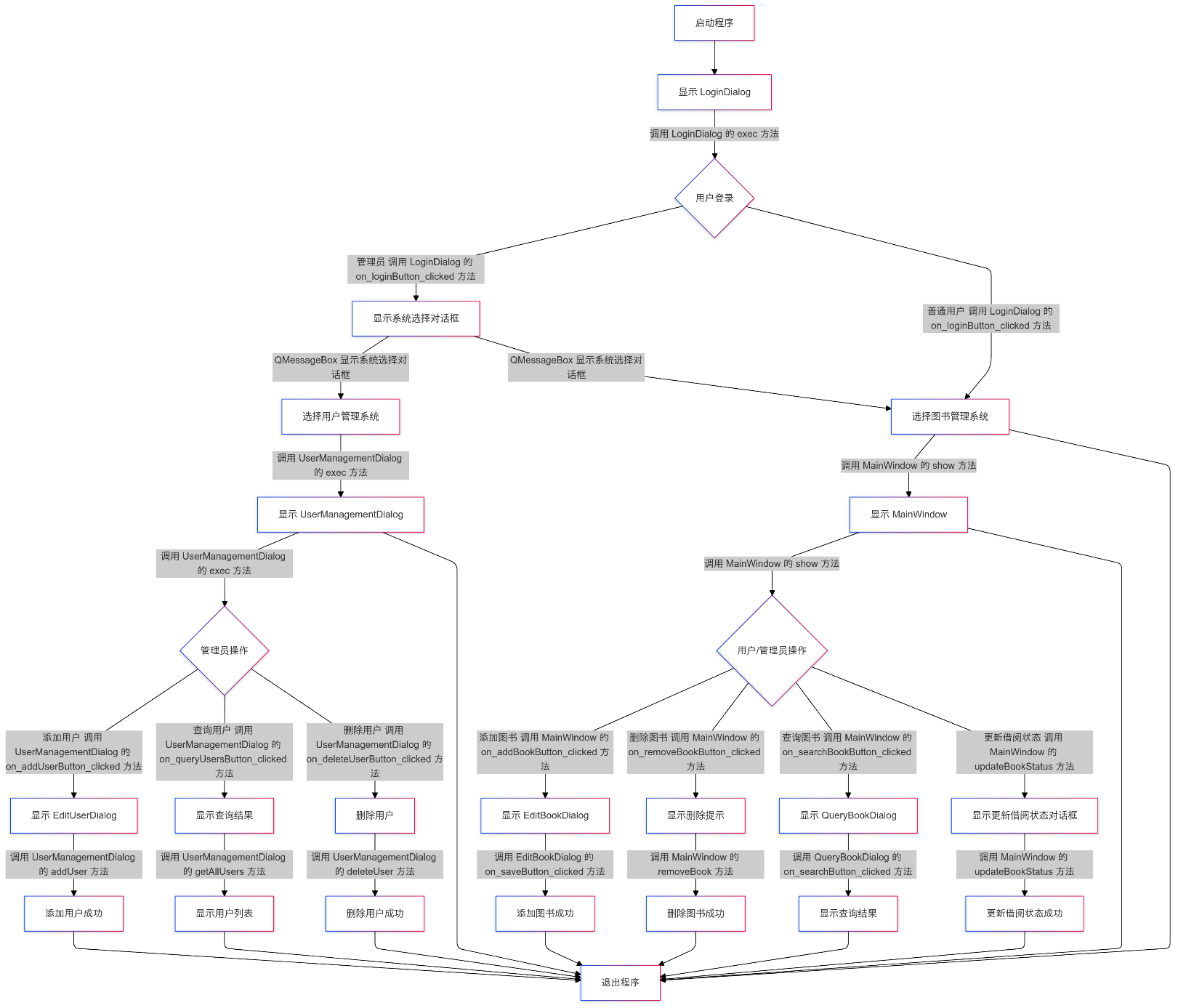


图 8 主程序总体流程图

1. **调试分析**
   1. 技术难点分析
      1. 模糊查找书籍信息

在实现模糊查找书籍信息时，技术难点主要集中在 MyUnorderedMap 数据结构的应用上。系统使用倒排索引存储关键词与书籍的映射关系，确保高效查找与查询。在添加书籍时，书名和作者的每个词都会被提取并存储在 MyUnorderedMap 中，使得查询时能够迅速找到相关书籍。每个关键词对应一个 std::vector<Book\*>，因此多个书籍可以共享同一个关键词映射。

模糊查询时，系统遍历查询中的每个词，在 MyUnorderedMap 中查找匹配的书籍，并记录每本书的匹配次数。利用额外的 MyUnorderedMap<Book\*, int> 结构，系统能够高效地统计每本书的匹配次数，并根据这些次数对书籍进行排序，从而实现精准的查询结果。

随着数据量的增加，MyUnorderedMap 需要动态扩展以维持良好的查询性能。为了防止负载因子过大，系统在插入新元素时会检查当前负载，并在必要时通过 rehash 扩容，确保查询操作不受性能瓶颈影响。

在删除书籍时，系统需要从倒排索引中移除相关关键词，确保数据一致性。通过 erase 方法，MyUnorderedMap 删除特定元素，更新倒排索引中的映射关系。

MyUnorderedMap高效存储和查询关键词与书籍的映射，如果匹配的书籍数量近似则时间复杂度近似为O(1)。更加详细的是O(n),n为匹配的书籍数目。

* + 1. 可视化

需要连接各个数据处理函数与对应的槽函数，确保界面与数据逻辑的有效交互。涉及到正确配置信号与槽的连接，避免重复连接或遗漏连接，从而确保用户操作能够准确触发相应的功能。

* + 1. AVLTree数据搜索

BookID精准查找用AVL树构建，进行搜索。

**3.2调试错误分析**

3.2.1交互窗口弹出两次

错误分析：

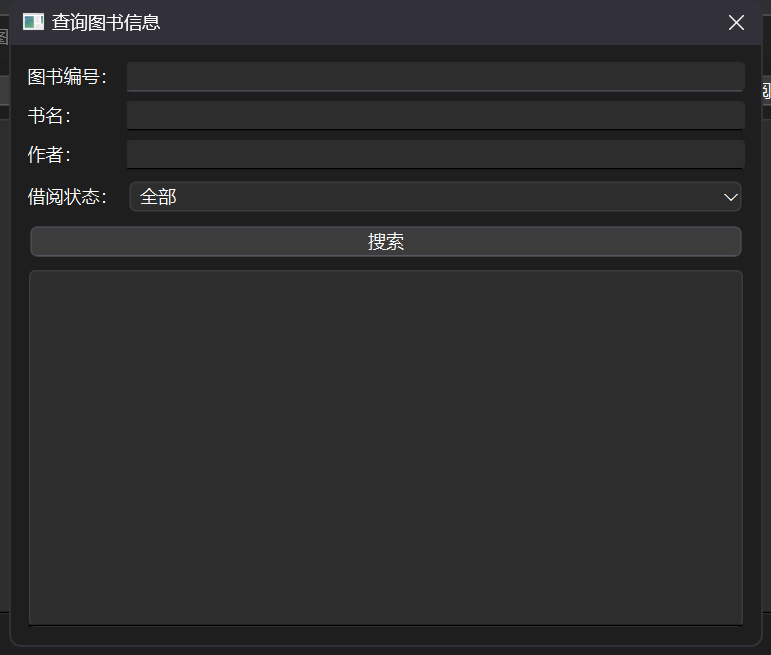


图 9两次弹出窗口

上述窗口反复弹出，经过检验，发现是



图 10错误代码

Qt会自动连接搜索按钮的点击事件到槽函数，即connect(ui->searchButton, &QPushButton::clicked, this, &QueryBookDialog::on\_searchButton\_clicked);会在Qt提供的基类中自动执行。

只需删除手动连接即可。

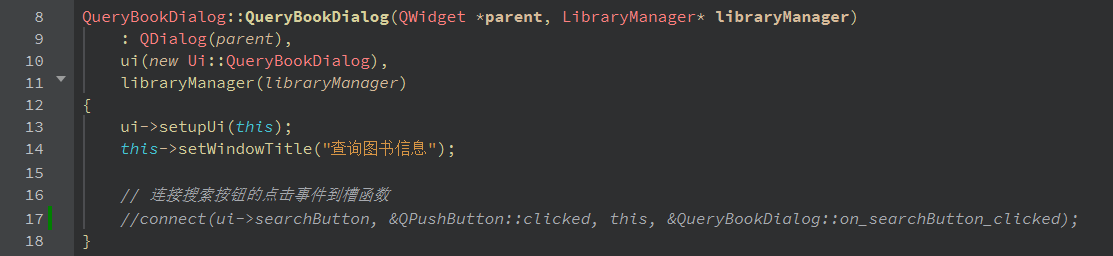


图 11修复代码

3.2.2 访问私有变量

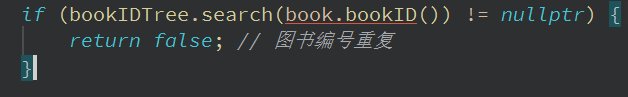


图 12错误代码

其它类中访问Book类实例的私有变量，出错



图 13报错

Book中添加get访问函数

文本

描述已自动生成

图 14添加访问器

访问修改为

文本

描述已自动生成

图 15修改代码

消除错误

3.2.3 管理员不能管理图书

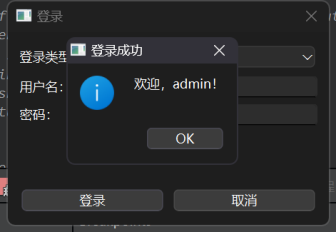
文本

描述已自动生成

图 16原代码

管理员登录后直接进入用户管理系统，无法进入图书管理系统

图 17管理员登录



为管理员加入选择，选择用户管理还是图书管理

文本

描述已自动生成

图 18修改代码

图形用户界面

描述已自动生成

图 19新加选择功能

**4.测试结果分析**

4.1登录系统

图形用户界面, 应用程序

描述已自动生成

图 20用户登录

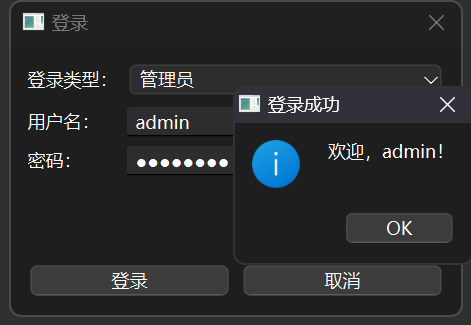


图 21管理员登录

图形用户界面

描述已自动生成

图 22管理员登录选择

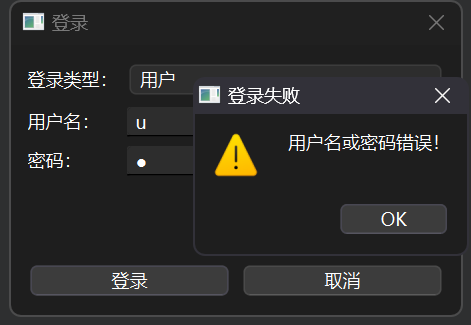


图 23登录失败

图形用户界面, 应用程序

描述已自动生成

图 24管理员账号无法选择用户登录

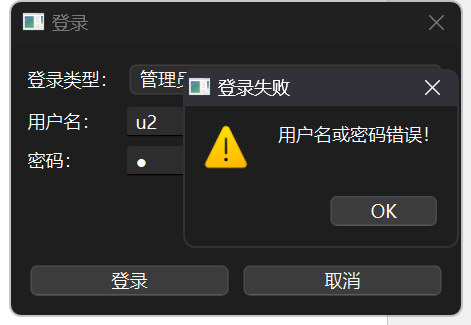


图 25用户无法登录管理员

图形用户界面, 应用程序

描述已自动生成

图 26密码空

4.2用户管理系统

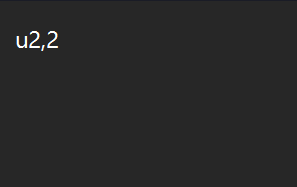


图 27 users.txt

文本

描述已自动生成

图 28 admins.txt

手机的屏幕

描述已自动生成

图 29 用户管理系统

电脑游戏的截图

描述已自动生成

图 30 添加用户



图 31添加成功



图 32重复添加

电脑萤幕的截图

中度可信度描述已自动生成图 33空输入

电脑萤幕的截图

中度可信度描述已自动生成

图 34删除用户

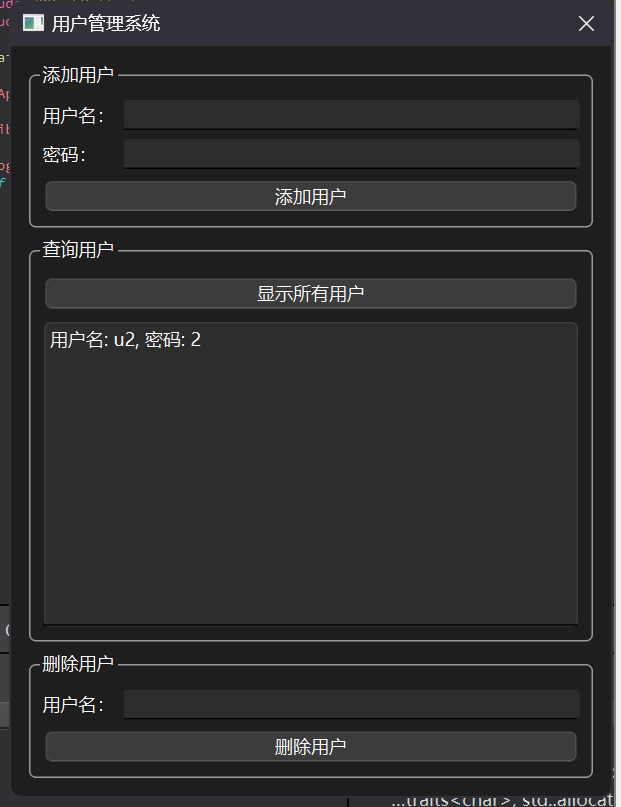


图 35删除成功

图形用户界面

描述已自动生成

图 36管理员退出用户管理后选择下一步操作

4.3图书管理系统



图 37显示所有图书

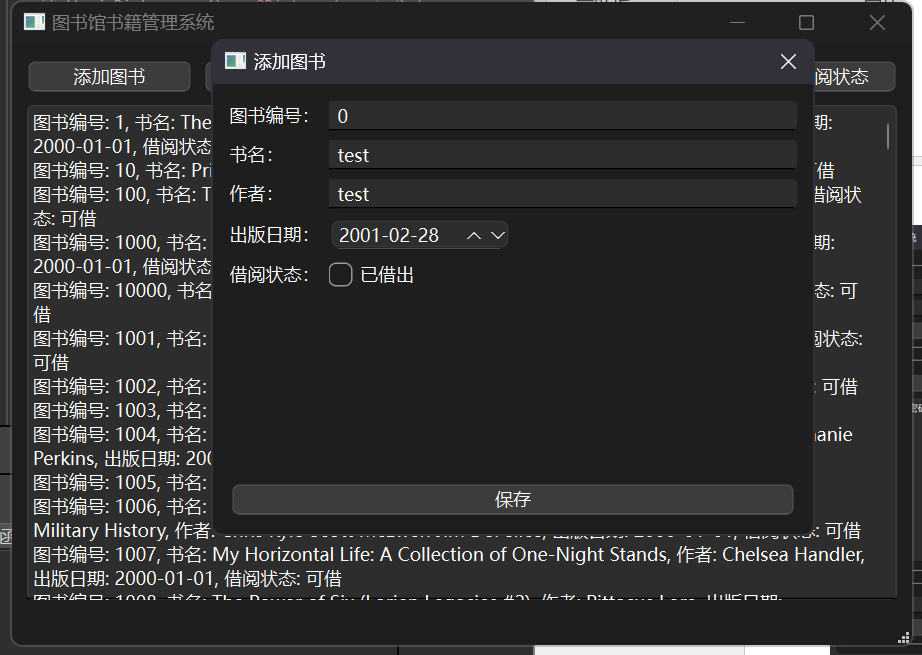


图 38添加图书，输入非闰年2-29自动变成2-28

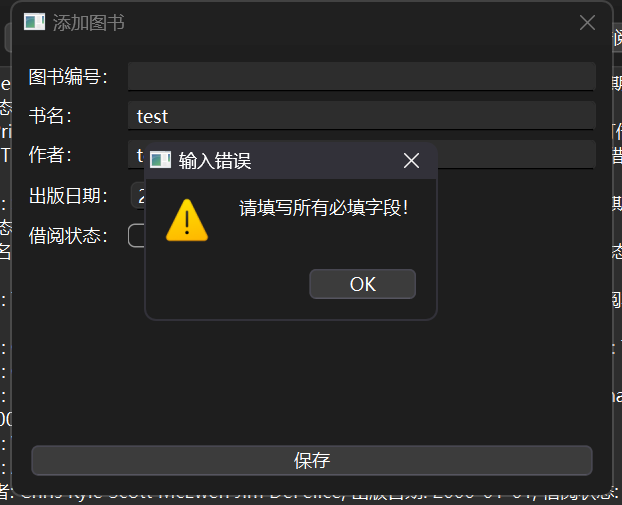


图 39空输入

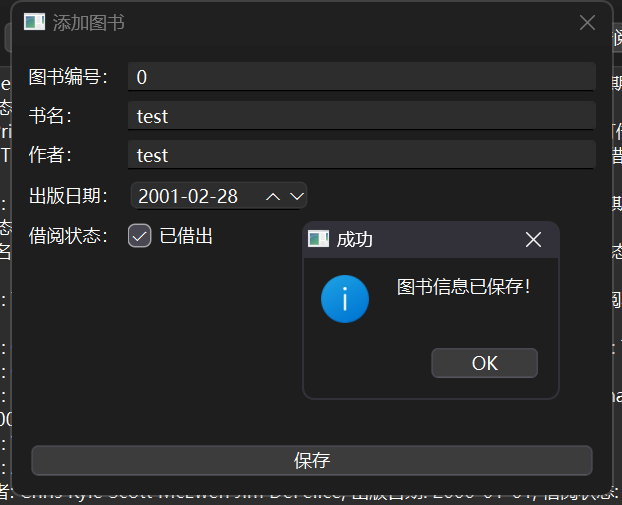


图 40保存成功

文本

描述已自动生成

图 41添加成功

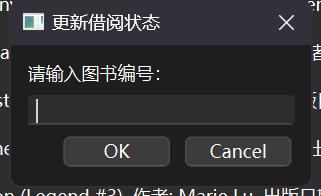


图 42更新借阅状态

图形用户界面, 应用程序

描述已自动生成

图 43空输入



图 44输入正确后出现选项

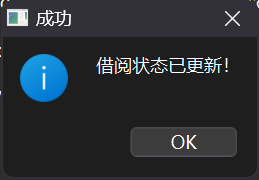


图 45更新成功



图 46更新成功

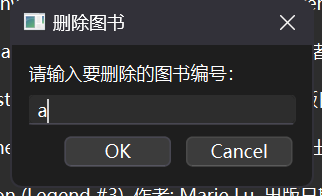


图 47删除图书错误输入

图形用户界面, 应用程序

描述已自动生成

图 48编号不存在

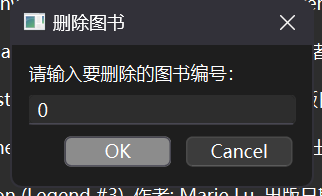


图 49正确输入

图形用户界面, 文本, 应用程序

描述已自动生成

图 50删除确定

手机屏幕的截图

描述已自动生成

图 51删除成功

文本

描述已自动生成

图 52删除成功



图 53编号AVL查询

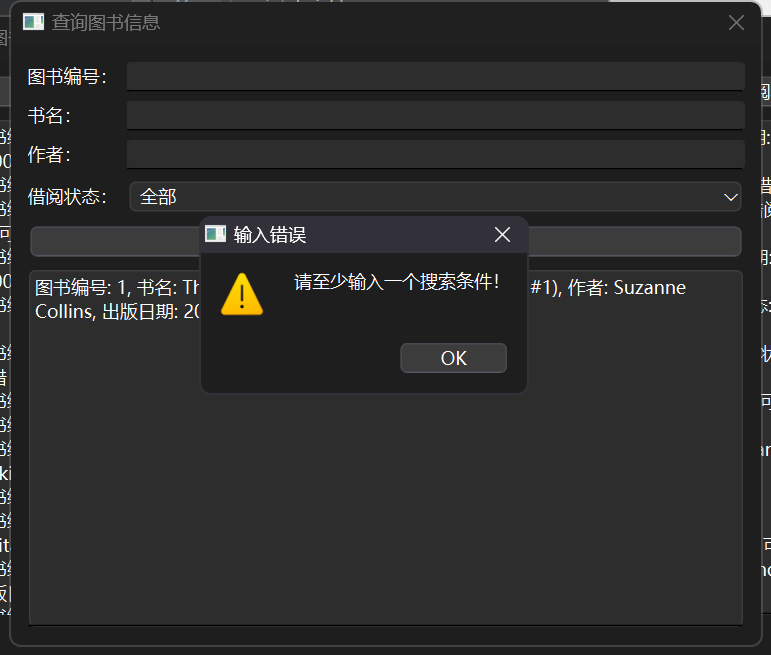


图 54空输入

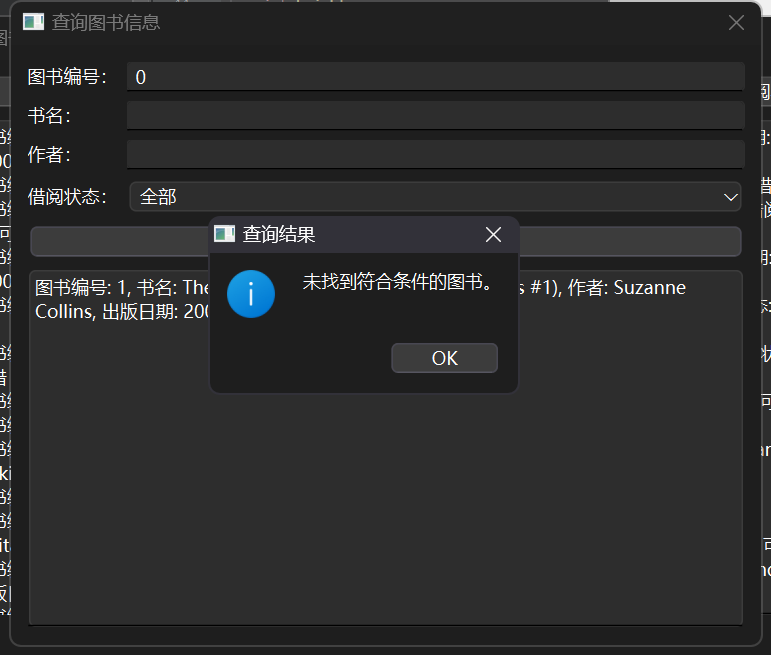


图 55未找到

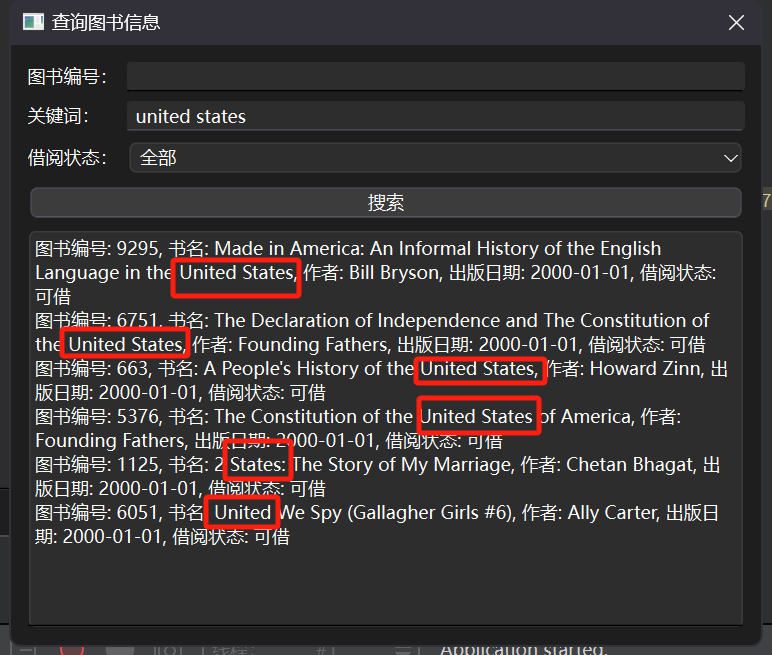


图 56书名模糊查询

图形用户界面, 文本

描述已自动生成

图 57作者模糊查询

图形用户界面, 文本, 应用程序

描述已自动生成

图 58借阅状态查询



图 59关闭程序自动保存



图 60重新打开成功加载