《电影信息查询管理系统》概要/详细设计

项目名称：电影信息查询管理系统

当前版本：<1.2.0>

最后修改时间：2023 年 12 月 10 日

修订记录

修订版本号	修订日期	修订描述	作者
V.1.0	2023/12/10	终版	小组共同

目录

1 引言

1.1 系统标识

项目名称：电影信息查询管理系统

项目简称：MIIMS（Movie Information Inquiry and Management System）

项目组成员：

姓名
吴文博
王恩琪
孙鉴
姚冠男

1.2 编写目的

本文是对项目内容的详细说明，给出系统所需全部功能的实现方法与实现要求，进而指导项目的具体开发工作。

1.3 本文的读者群

本文面向的读者群主要为技术管理人员、数据库系统应用开发人员。

理解本文需要了解关系型数据库相关知识，以及常见的前后端框架的原理与使用方法；

1.4 专门术语及缩略词定义

无

1.5 参考文献

1. MySQL 中文文档：https://www.mysqlzh.com/
2. Spring 文档：https://docs.spring.io/spring-framework/
3. Vue 开发框架：https://cn.vuejs.org/
4. Element ui 指南：https://element.eleme.cn/#/zh-CN

2 概要设计

2.1 设计注意事项、原则、目标及相关规范

电影信息查询管理系统遵循“低耦合，高内聚”的设计原则，将系统划分为三个相对独立的模块：前端的信息展示与用户操作、后端的操作请求处理及数据库交互、数据库管理系统的数据增删改查实现。

前端模块使用 Vue 框架进行开发。Vue 框架具有集成度高、对开发人员友好、组件较为完善等优势，能够提高开发效率，并形成能够在各类平台高效运行的前端界面，满足用户操作的各类需求。

后端模块使用 java 语言和 spring 框架进行开发。java 功能丰富，spring 框架提供了统一、规范的接口，使得后端能够高效地与前端和 MySQL 数据库进行数据交互。

MySQL 数据库作为系统的数据库管理系统，具有体积小、速度快等优点，能够满足系统实际运行时的性能需求。

在系统设计阶段，需要充分考虑数据模型的设计。数据模型需要充分考虑未来新增功能的可能性，并且已经根据范式分解理论进行了合适程度的分解，以实现较好的独立性、可扩展性、较小的冗余度和较高的性能。

前后端的分离设计使得系统新增某一功能时，只需分别在前端增加一独立的子组件、在后端增加一独立的处理函数即可。

本系统应具有较为全面的电影信息管理和查询功能，能够实现良好的用户体验，在不同的平台上均能够高效地运行，并实现时延低、内存和 CPU 消耗小等优势。此外，

本系统还应具有身份验证、阻止非法行为等功能，实现较高的安全性。

2.2 总体结构设计概述 (System Architecture Overview)

根据 2.1 节的系统划分方式，我们进一步设计了前端页面展示、前端操作生成、后端请求处理、数据库服务四大类型构件，并对各个构件进行了进一步的功能划分，形成了如图 1 所示的系统总体架构图。

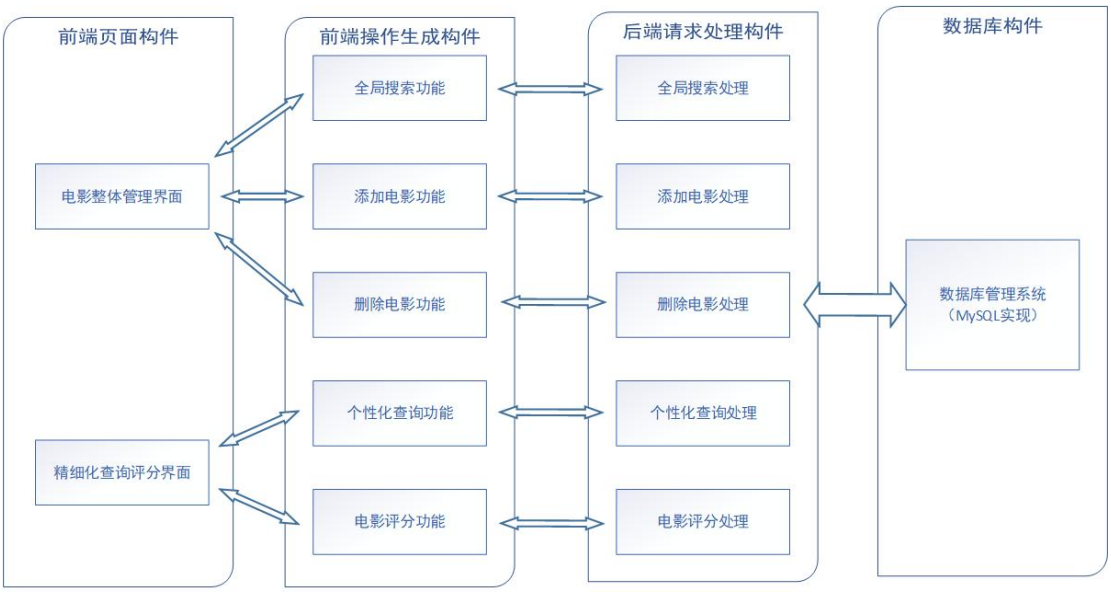

图 1 系统整体架构图

其中，前端页面类型构件主要为用户提供可视化的数据查看和操作界面，主要分为电影整体管理子界面和精细化查询评分子界面。在电影整体管理子界面，用户可以查看当前数据库中所有电影信息，并进行电影的添加和删除操作；在精细化查询评分子界面，用户可以根据个性化的需求，根据不同的类型、语言、国家和评分查询所需的电影信息，并在观影后进行评分操作。该构件主要使用 Vue 代码和 CSS 样式实现。

前端操作生成类型构件主要对用户对电影信息的操作进行处理，根据输入的参数（如查询内容）生成相应的 HTTP 请求；同时将请求的结果实时显示在页面上。该构件主要使用 Vue 代码实现。

后端请求处理类型构件主要对前端发送的各类操作对应的 HTTP 请求进行解析，并由其中的参数构造对应的 SQL 语句，对数据库发起增删改查操作，将结果返回至前端。该构件主要使用 java 的 spring 框架实现。

数据库构件具体执行系统所构造的 SQL 语句，主要由数据库管理系统实现。本系统选

取 MySQL 作为数据库管理系统。

2.3 构件设计 (Components Design)

本小节主要对 2.2 节所提出的各类构件及其功能划分进行详细描述。

2.3.1 构件 1 电影搜索-前端构件

2.3.1.1 功能描述

此构件可以获取数据库中所有电影信息，并且可以通过搜索特定电影名的方式进行全局搜索，在用户界面以分页表格形式展示电影的电影名、评分、类型、国家、语言、上映时间、时长、别名、导演、演员、编剧等信息。

2.3.1.2 对象模型

构件主要采取模块化设计方法，主要数据结构为：

- TableData：由电影信息构成的列表，电影信息以 JSON 对象表示
- tableData：显示在当前呈现出的一页电影信息，从总表中获得
- query：包含分页和搜索相关的参数，如 pageSize、pageIndex、name

主要函数为：

- axios.get(url).then()：获取电影信息列表函数
- axios.post(url, sendpara).then()：获取电影信息请求函数
- getData(TableData)：获取当前页面电影信息函数

2.3.1.3 接口规范

1.电影信息列表获取函数

函数原型：axios.get(url).then()

功能描述：向给定的 url 发送电影信息获取 HTTP 请求，并将结果写入电影信息列表 TableData。

参数说明：输入为 HTTP 请求的目标 url，输出为由 JSON 对象构成的电影信息列表。

前置条件：该函数在电影整体管理界面首次加载时执行，调用该函数前假定后端服务和数据库服务均已开启。

后置条件：电影信息列表变量 TableData 与数据库的电影信息同步。

2. 获取电影信息请求函数

函数原型：axios.post(url, sendpara).then()

功能描述：根据用户输入特定电影名信息，向给定的 url 发送获取电影信息 HTTP 请求。

参数说明：输入为 HTTP 请求的目标 url 和待插入电影元组信息构成的 JSON 对象 sendpara，输出为由 JSON 对象构成的电影信息列表。

前置条件：该函数在电影整体管理界面首次加载时执行，调用该函数前假定后端服务和数据库服务均已开启。

后置条件：电影信息列表变量 TableData 与数据库的电影信息同步。

3.当前页面电影信息函数

函数原型：getData(TableData)

功能描述：根据当前用户选取的页数和每页显示信息的数量，计算目前需要显示在用户界面上的电影信息列表。

参数说明：输入为电影信息总表 TableData，输出为当前需要展示在用户界面的电影信息列表 tableData。

前置条件：TableData 非空（前端已获取所有电影信息内容）

后置条件：tableData 的信息实时展示在用户界面。

2.3.2 构件 2 电影数据删除-前端构件

2.3.2.1 功能描述

此构件根据用户删除的电影的属性值信息，向后端发起删除请求，删除这条电影记录。

2.3.2.2 对象模型

该构件主要采取模块化设计方法，主要数据结构为：

• row：即将被删除的电影信息，为一包含 movieId、movie、grade、directorList、scriptwriterList、performerList、type、country、language、date、duration、alias、gradeNum 属性的结构体。

主要函数为：

• axios.post(url, sendpara).then()：删除元组请求函数。

2.3.2.3 接口规范

1.删除元组请求函数

函数原型：axios.post(url, sendpara).then()

功能描述：根据用户删除电影属性值，向给定的 url 发送删除电影信息的 HTTP 请求。

参数说明：输入为 HTTP 请求的目标 url 和待删除电影元组信息构成的 JSON 对象 sendpara，输出为后端返回的运行结果（成功/失败）。其中 url 应设置为“IP 地址:端口号/delete”。

前置条件：用户删除的电影在电影信息总表 TableData 中存在，且后端服务与数据库服务均开启。

后置条件：该请求得到后端响应，且 movie 表中相关记录被删除，scr_movie、dir_movie、per_movie 三个表中与该记录相关的元组也被级联删除。同时前端展示的该条电影信息也同步消失。

2.3.3 构件 3 电影数据添加-前端构件

2.3.3.1 功能描述

该构件根据用户输入的添加电影的信息，向后端发起添加该条电影记录的请求。

2.3.3.2 对象模型

该构件主要采取模块化设计方法，主要数据结构为：

• Movie：用户填写的新增电影信息，为一包含 movieId、movie、grade、directorList、scriptwriterList、performerList、type、country、language、date、duration、alias、

gradeNum 属性的结构体。

主要函数为：

• axios.post(url, sendpara).then()：插入信息请求函数

2.3.3.3 接口规范

1. 插入信息请求函数

函数原型：axios.post(url, sendpara).then()

功能描述：根据用户输入的新增电影信息，向给定的 url 发送插入电影信息的 HTTP 请求。

参数说明：输入为 HTTP 请求的目标 url 和待插入电影元组信息构成的 JSON 对象 sendpara，输出为后端返回的运行结果（成功/失败）。其中 url 应设置为“IP 地址:端口号/insert”。

前置条件：用户输入的电影各属性值完整且合法；后端服务与数据库服务均开启。

后置条件：该请求得到后端处理，同时数据库的各表均作出相应更新。

2.3.4 构件 4 个性化查询-前端构件

2.3.4.1 功能描述

该构件根据用户选取的类型、国家、语言和评分属性值，发起个性化查询请求并将符合条件的电影展示在用户界面。

2.3.4.2 对象模型

该构件主要采取模块化设计方法，主要数据结构为：

• query：查询内容，为一包含 type、country、language、grade、pageSize 和 pageIndex 属性的结构体。

主要函数为：

• axios.post(url, sendpara).then()：个性化查询请求函数。

2.3.4.3 接口规范

1.个性化查询请求函数

函数原型：axios.post(url, sendpara).then()

功能描述：根据用户输入的电影类型、所属国家、语言和评分四个属性值，向给定的url 发送个性化查询符合要求的电影信息的 HTTP 请求，并将结果展示。

参数说明：输入为 HTTP 请求的目标 url 和由 2.3.4.2 节所述的 query 结构体转化形成的 JSON 对象 sendpara，输出为满足要求的电影信息列表。其中 url 应设置为“IP 地址:端口号/ personalizedSelect”。

前置条件：用户所选取个性化查询信息均合理，且后端服务与数据库服务均开启。

后置条件：前端电影信息列表 TableData 被更新为个性化查询结果，并实时展示在个性化查询评分界面。

2.3.5 构件 5 电影评分-前端构件

2.3.5.1 功能描述

该构件用于支持用户对电影的评分需求，将用户对某一电影的评分发送给后端，并在用户界面展示更新后的平均评分。

2.3.5.2 对象模型

该构件主要采取模块化设计方法，主要数据结构为：

- my_grade：用户评分
- send_grade：由电影名 name 与评分 my_grade 构成的结构体

主要函数为：

- handleEdit(row,grade)：评分绑定函数
- axios.post(url, sendpara).then()：评分更新请求函数。

2.3.5.3 接口规范

1.评分绑定函数

函数原型：handleEdit(row,grade)

功能描述：将用户的评分与某一电影绑定

参数说明：输入为用户评分电影信息 row 和评分数 grade，输出为结构体 send_grade

前置条件：用户评分数在正常范围，电影名已在电影信息列表中存在。

后置条件：执行个性化查询请求函数。

2.个性化查询请求函数

函数原型：axios.post(url, sendpara).then()

功能描述：根据电影名 name 和评分值 my_grade 构造 HTTP 请求，并向后端相应的 url 发送该请求。

参数说明：输入为 HTTP 请求的目标 url 和评分绑定函数返回的 send_grade 结构体转化形成的 JSON 对象 sendpara，输出为后端返回的执行结果（成功/失败）。其中 url 应设置为“IP 地址:端口号/ update”。

前置条件：评分绑定函数正常返回。

后置条件：所评分电影对应的评分、评分人数属性被更新，并在用户界面中该条记录处体现。

2.3.6 构件 6 电影搜索-后端处理构件

2.3.6.1 功能描述

该构件解析并处理前端发起的电影搜索请求，并将数据库查询结果返回给前端。

2.3.6.2 对象模型

该构件主要采取模块化设计方法，主要数据结构为：

• movie：由名称 name、评分 grade、类型 type、所属国家 country、所使用语言

language、日期 date、片长 duration、别名 alia、导演名称 director、演员列表 performer、编剧列表 scriptwriter 构成的结构体。

主要函数为：

- selectAll()：搜索处理函数
- selectByName()：搜索处理函数(按名称)

2.3.6.3 接口规范

1. 搜索处理函数

函数原型：selectAll()

功能描述：处理前端发起的获取全部电影信息请求，构造相应的查询 SQL 语句并将查询数据库得到的结果返回给前端。

参数说明：输入为前端获取全部电影信息的 GET 请求，面向前端的返回值为电影信息列表。

前置条件：监听到前端向/selectAll 这一 url 发送获取全部电影信息的 HTTP 请求。

后置条件：前端的电影信息列表变量得到同步更新。

1. 搜索处理函数(按名称)

函数原型：selectByName()

功能描述：处理前端发起的获取特定电影信息请求，构造相应的查询 SQL 语句并将查询数据库得到的结果返回给前端。

参数说明：输入为前端获取全部电影信息的 GET 请求，面向前端的返回值为电影信息列表。

前置条件：监听到前端向/selectByName 这一 url 发送获取全部电影信息的 HTTP 请求。

后置条件：前端的电影信息列表变量得到同步更新。

2.3.7 构件 7 电影数据添加-后端处理构件

2.3.7.1 功能描述

该构件解析并处理前端发起的电影添加请求，并将数据库插入操作的成功/失败结果返回给前端。

2.3.7.2 对象模型

该构件主要采取模块化设计方法，主要数据结构为：

• name、grade、type、country、language、date、duration、alias、director、performer、scriptwriter 等变量，分别代表电影名称、评分、类型、所属国家、语言、发布日期、时长、别名、导演、编剧人员和演员。

主要函数为：

• save()：电影添加处理函数

2.3.7.3 接口规范

1. 电影添加处理函数

函数原型：save()

功能描述：处理前端发起的添加电影信息请求，根据请求中的电影各属性值构造相应的数据插入 SQL 语句并将插入结果返回给前端。

参数说明：输入为前端生成的、包含待添加的电影记录的各属性值的 HTTP 请求，面向前端的返回值为插入操作执行结果（成功/失败）。

前置条件：监听到前端向/ insert 这一 url 发送包含属性值的添加电影 HTTP 请求。

后置条件：数据库中的 movie 表新增了一条记录，同时 director、performer、scriptwriter 表以及相应的关联实体表均进行了相应更新。

2.3.8 构件 8 电影数据删除-后端处理构件

2.3.8.1 功能描述

该构件解析并处理前端发起的电影信息删除请求，并将数据库删除操作的成功/失败结果返回给前端。

2.3.8.2 对象模型

该构件主要采取模块化设计方法，主要数据结构为：

• movieID：待删除的电影名称

主要函数为：

• delete()：电影删除处理函数

2.3.8.3 接口规范

1. 电影删除处理函数

函数原型：delete()

功能描述：处理前端发起的删除电影记录请求，根据请求中的电影名称构造删除操作 SQL 语句，并将删除操作执行结果返回给前端。

参数说明：输入为前端生成的、包含待删除的电影名称 movieID 的 HTTP 请求，面向前端的返回值为删除操作执行结果（成功/失败)。

前置条件：监听到前端向/ delete 这一 url 发送包含电影名称属性值的删除电影 HTTP 请求。

后置条件：数据库中的 movie 表中相关记录被删除，同时 director、performer、scriptwriter 三个关联实体表中与该记录有关的内容也被级联删除。

2.3.9 构件 9 个性化查询-后端处理构件

2.3.9.1 功能描述

该构件解析并处理前端发起的关于电影类型、语言、国家和评分的个性化查询请求，并将查询数据库所得结果返回给前端。

2.3.9.2 对象模型

该构件主要采取模块化设计方法，主要数据结构为：

• 评分 grade、电影类型 type、电影语言 language、所属国家 country

主要函数为：

• personlizedSelect()：个性化查询处理函数

2.3.9.3 接口规范

1. 个性化查询处理函数

函数原型：personlizedSelect()

功能描述：处理前端发起的个性化查询请求，解析请求中的评分、电影类型、语言、国家四个属性值并构造查询 SQL 语句，将属性值匹配的查询结果返回给前端。

参数说明：输入为前端生成的、包含电影评分 grade、电影类型 type、电影语言 language、所属国家 country 的 HTTP 请求，面向前端的返回值为属性值匹配的电影信息列表。

前置条件：监听到前端向/ personlizedSelect 这一 url 发送包含评分、电影类型、语言、国家四个属性值的个性化电影查询 HTTP 请求。

后置条件：用户界面显示满足个性化查询要求的电影信息列表。

2.3.10 构件 10 电影评分-后端处理构件

2.3.10.1 功能描述

该构件解析并处理前端发起的包含用户对某一电影的评分值的数据更新请求，将数据库更新操作结果返回给前端，并使用户界面显示评分更新后的电影列表。对象模型

该构件主要采取模块化设计方法，主要数据结构为：

• 评分 grade、电影 movieID

主要函数为：

• update()：电影评分处理函数

2.3.10.2 接口规范

1. 电影评分处理函数

函数原型：update()

功能描述：处理前端发起的数据更新请求，解析请求中包含的电影名称和评分值信息，先构造查询 SQL 语句并查询该电影当前的评分人数和平均评分，再计算更新后的平均评分并通过数据更新 SQL 语句写入该条电影记录，最终将更新结果（成功/失败）返回至前端。

参数说明：输入为前端生成的、包含用户评分 grade 和电影名称 movieID 的 HTTP 请求，面向前端的返回值为数据更新的操作结果（成功/失败）。

前置条件：监听到前端向/ update 这一 url 发送包含用户评分和所评分的电影名称的电影评分 HTTP 请求。

后置条件：数据库中该条电影记录元组的评分人数增加 1，同时平均评分属性得到更新。

2.3.11 构件 11 数据库管理系统构件

该构件主要由数据库管理系统 MySQL 构成，负责具体执行后端程序生成的 SQL 语句、对数据库进行 SQL 语句规定的增删改查操作。

2.3.11.1 功能描述

该构件主要采取模块化设计方法，主要函数为：

• 查询电影（SELECT 语句实现）、插入电影信息（INSERT 语句实现）、删除电影信息（DELETE 语句实现）、个性化电影查询（SELECT 语句实现）、电影评分更新（UPDATE 语句实现）

2.3.11.2 接口规范

1. 查询电影 SQL 语句

语句内容：select * from movie;

功能描述：查询所有的电影记录

参数说明：该语句无需外部输入；输出为所有电影信息元组。

前置条件：后端处理程序调用该 SQL 语句

后置条件：后端处理程序获得所有电影信息元组

2. 查询电影 SQL 语句

语句内容：select * from movie where movie like CONCAT('%', #{movie}, '%');

功能描述：查询特定电影记录

参数说明：该语句无需外部输入；输出为所有电影信息元组。

前置条件：后端处理程序调用该 SQL 语句

后置条件：后端处理程序获得所有电影信息元组

3. 插入电影信息 SQL 语句

语句内容：

insert into dir_movie(movie_id, director_id) values (#{movieId},#{directorId})

insert into per_movie(movie_id, performer_id) values (#{movieId},#{performerId})

insert into scr_movie(movie_id, scriptwriter_id) values (#{movieId},#{scriptwriterId})

功能描述：根据新增的电影信息，在表中新增记录，同时在三个关联实体中增加相应记录。

参数说明：输入为 movie、performer、director、scriptwriter 表涉及的全部属性值；输出为语句执行状态。

前置条件：后端处理程序调用该 SQL 语句

后置条件：后端处理程序获得插入操作执行结果。

4. 删除电影信息 SQL 语句

语句内容：delete from movie where movie_id = #{movieId};

功能描述：删除某一电影记录

参数说明：输入为待删除电影的 movieID 属性；输出为语句执行状态。

前置条件：后端处理程序调用该 SQL 语句

后置条件：后端处理程序获得删除操作执行结果。

5. 电影信息个性化查询 SQL 语句

语句内容：select * from movie where grade >= #{gradeLow} and grade <= #{gradeHigh}${Sql}

功能描述：根据国家、类型、语言的值以及分数范围查询符合要求的电影元组

参数说明：输入为国家、类型、语言和评分属性值；输出为满足要求的电影元组。

前置条件：后端处理程序调用该 SQL 语句

后置条件：后端处理程序获得个性化查询结果。

6. 评分信息更新 SQL 语句

语句内容：

update movie set grade=#{gradeNew},grade_num=#{gradeNumNew} where movie_id = #{movieId}

功能描述：向被评分电影元组分别写入用户评分后的新平均评分和新评分人数。

参数说明：输入为被评分电影 id 以及更新后平均评分、更新后评分人数；输出为更新操作执行状态。

前置条件：后端处理程序调用该 SQL 语句

后置条件：后端处理程序获得更新操作执行状态。

2.4.构件合作模型 (Components Collaboration)

本系统中存在页面构件与前端操作生成类构件、前端操作生成类构件和后端请求处理类构件、后端请求处理类构件和数据库构件之间的合作关系，下面对这三类合作关系分别进行阐述。

2.4.1 页面构件与前端操作生成类构件合作模型

页面构件与前端操作生成类构件主要在两方面构成合作：一是数据绑定，用户的输入需要实时地引起前端操作生成类构件的变量变化，同时前端在收到后端的响应数据后也需实时地使这些内容在用户页面上展示。二是事件触发，前端操作生成类构件需要对用户在页面构件进行的点击、提交等操作进行响应。

其中，数据绑定可以通过 Vue 框架提供的 v-model 功能实现，例如在个性化查询中，使用如下的代码实现页面中“电影类型”选择框与前端个性化查询构件中“query”变量的绑定：

<el-select v-model="query.type" placeholder="类型" class="handle-select mr10">

而事件触发可以通过 Vue 框架的@click{}语句实现，例如在个性化查询中，使用如下代码实现用户点击“搜索”按钮后，前端个性化查询构件即执行 handleSearch()函数发起查询请求并更新前端数据：

<el-button type="primary" icon="el-icon-search" @click="handleSearch">搜索</el-button>

2.4.2 前端操作生成类构件与后端请求处理类构件合作模型

前端操作生成类构件与后端请求处理类构件主要通过 HTTP 请求进行数据交互，其中全局电影搜索请求使用 GET 请求，目的是为了实现更高的效率；而其余功能的实现使用 POST 请求，目的是为了更高的安全性。在用户发起操作后，前端操作生成类构件将用户的输入参数转化为 JSON 格式，并封装在 HTTP 请求中，发送至后端处理相应功能的 url；而后端监听到发送至某 url 的 HTTP 请求后，即解析该请求获得相关参数，并将数据库操作的结果封装后返回给前端构件。二者的合作主要通过前端的 axios 包和后端的 spring 框架实现。

2.4.3 后端请求处理类构件与数据库构件合作模型

后端请求处理类构件与数据库构件的合作主要体现在：后端根据功能需求以及所得参数构造查询、插入、更新、删除 SQL 语句并交由数据库管理系统 MySQL 执行，MySQL 对数据库进行数据操作后将结果返回给后端程序。

2.5.图形用户接口设计（GUI Design）

本小节给出系统的图形用户接口设计，包括页面外观、展示内容、用户可操作内容和操作方式等，并进行相应的处理过程分析。

2.5.1 电影整体管理界面

登陆页面如图 2 所示。

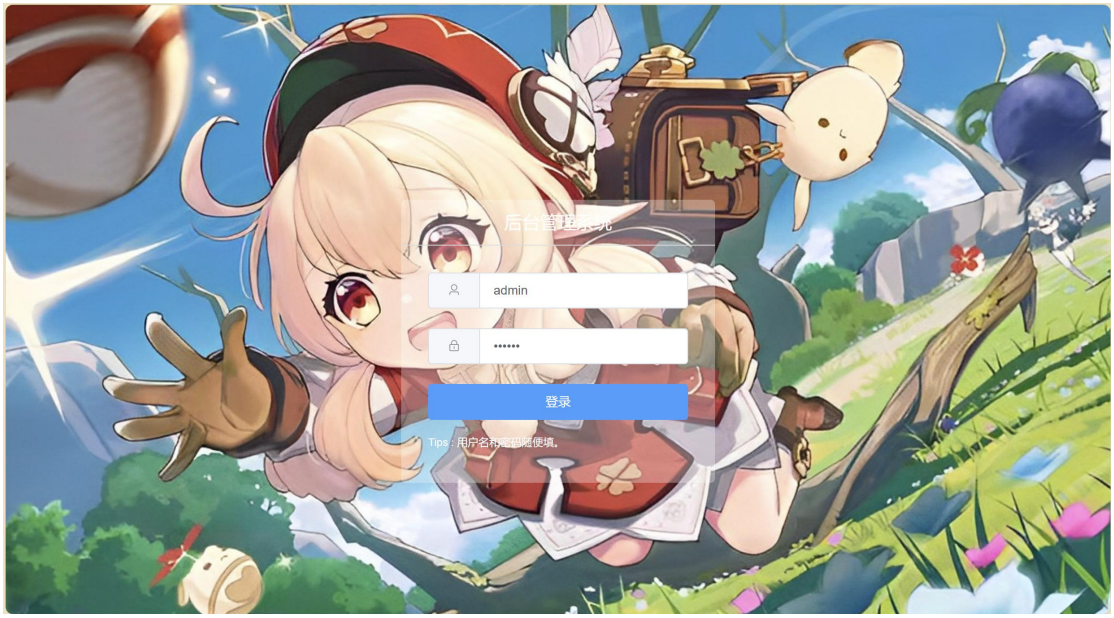

图 2

电影整体管理界面的外观如图 3 所示。

该界面主要部分展示了数据库中所有电影构成的列表，使用户能够了解包括电影名、评分、电影类型、所属国家、语言、上映时间、时长、别名、导演、演员、编剧等所有感兴趣的信息。用户可以通过表格上方的“添加电影”按钮向数据库中添加电影，或者点击“删除”按钮删除某一条电影信息。

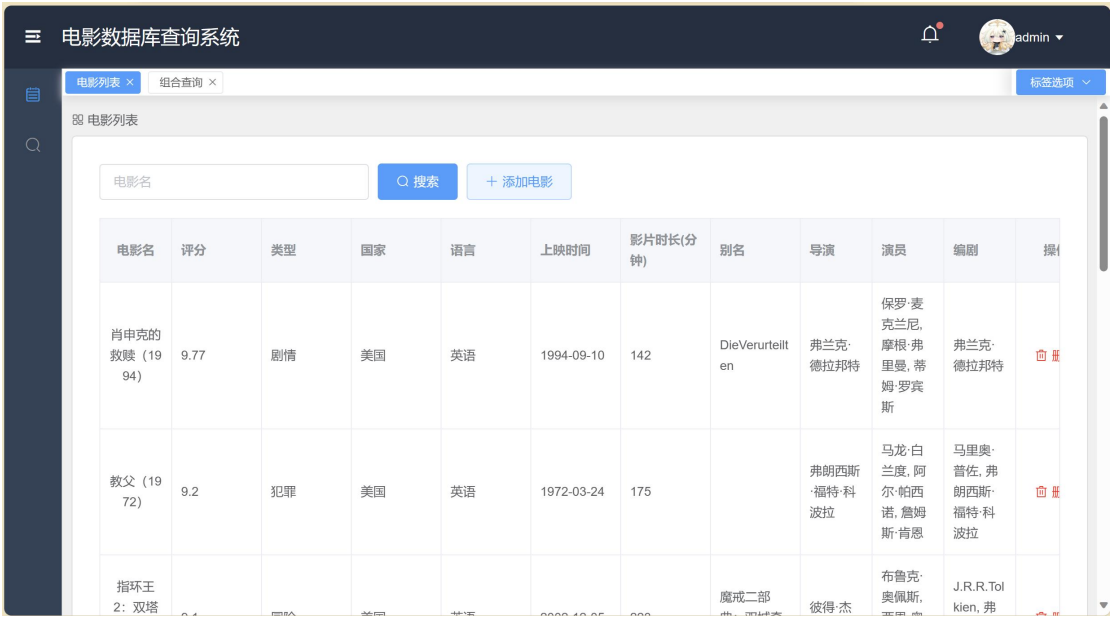

图 3

由图 4 所示，当用户点击“添加电影”按钮后，即出现填写新增电影信息的表单.

图 4

由图 5 所示，用户点击记录右侧的“删除”按钮，在弹出框点击确认可以删除电影。

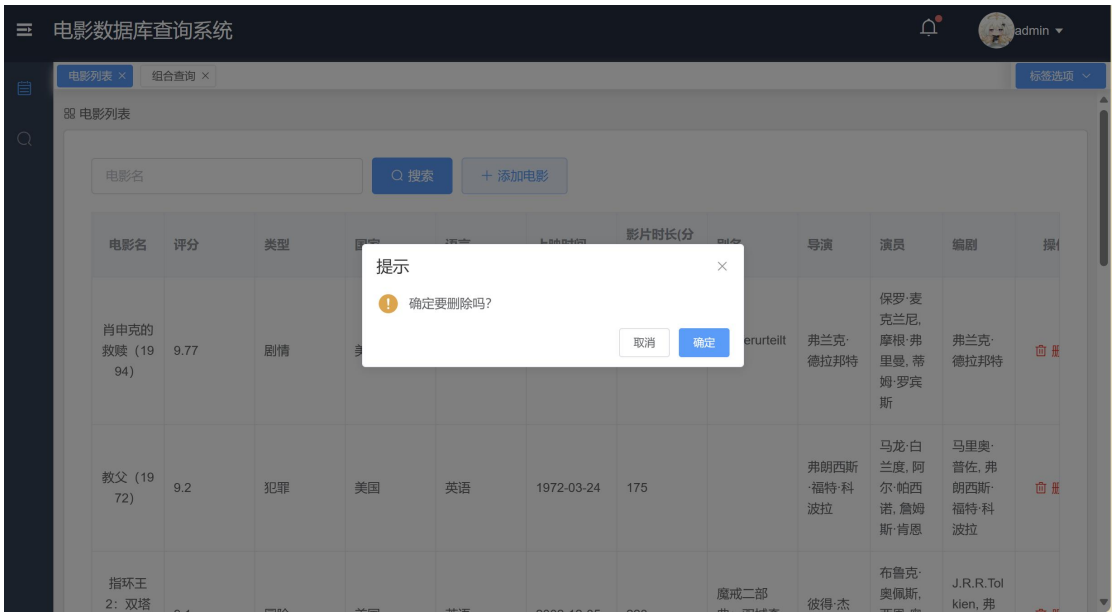

图 5

2.5.2 精细化查询评分界面

精细化查询评分界面的界面设计如图 6 图 7 所示。

图 6 是按照电影名查询，输入特定电影名，点击搜索进行查询。

图 7 是按照类型，语言等搜索关键词进行搜索的界面。

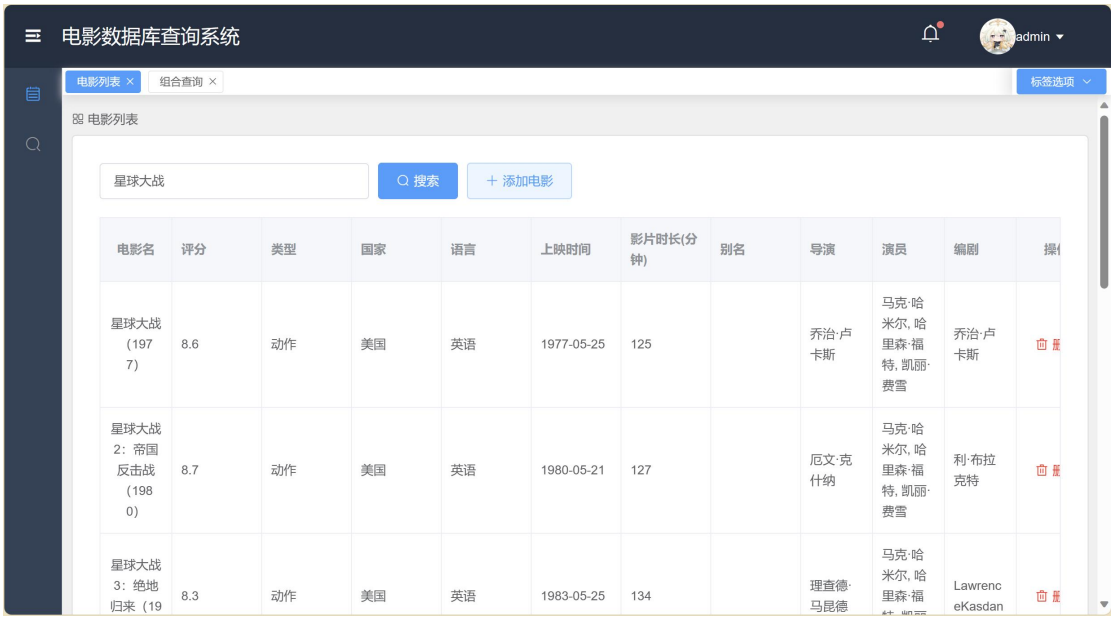

图 6

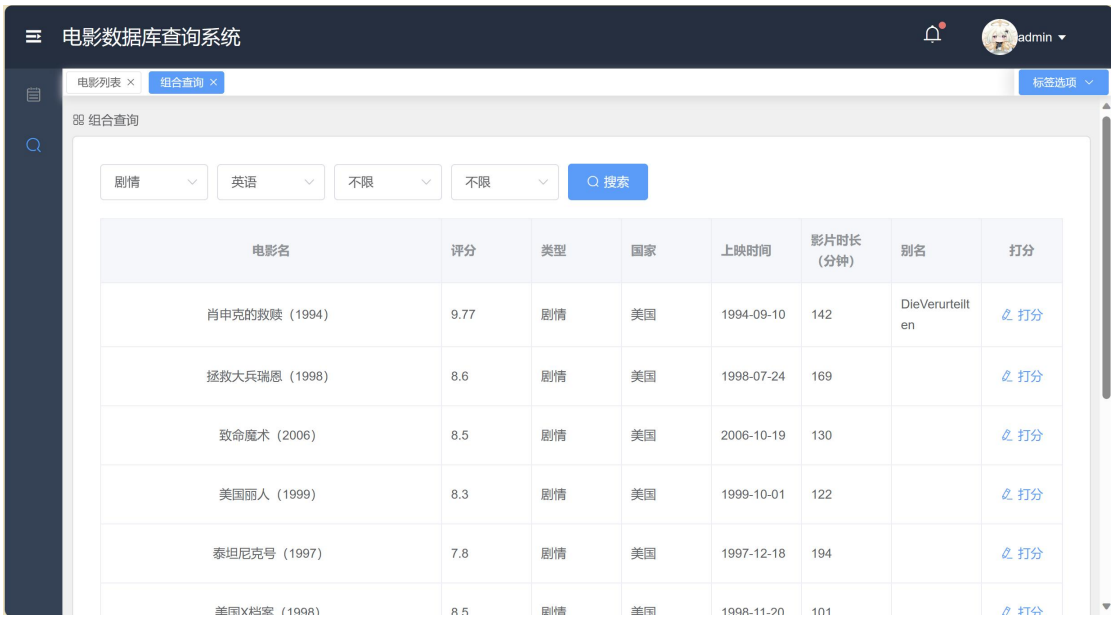

图 7

此后，用户可以点击某一电影右侧的“评分”按钮，在弹出框内写下对该电影的评分，如图 8 所示。

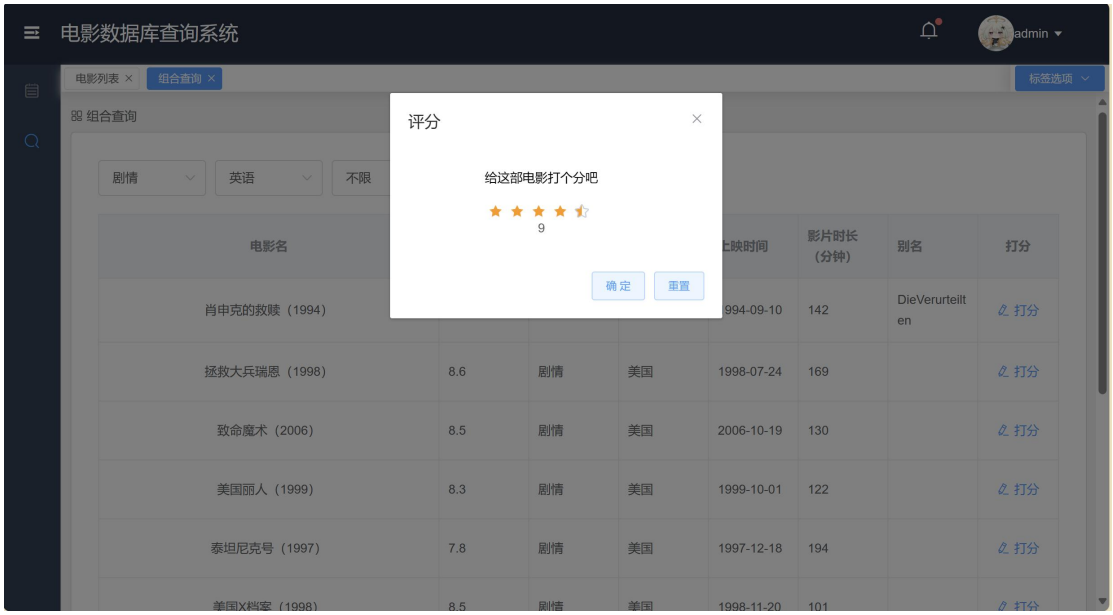

图 8

2.6.数据库设计

本系统中对象永久存储（Object Persistence）策略采用数据库进行存储。本小节对系统的数据库设计进行详细阐述。

2.6.1. 数据库表设计(schema or table design)

表名		movie				说明	电影目录表			
序号	字段名称	字段类型	长度	允许空	缺省值	主外键	索引	字段描述	创建日期	备注
1	movie_id	INT	10	N		PK		电影 ID	2023/12/10	自增长
2	movie	VARCHAR	20	N			Y	电影名称	2023/12/10	
3	grade	DECIMAL	2，1	N				评分	2023/12/10	
4	type	VARCHAR	20	N				电影类型	2023/12/10	
5	area	VARCHAR	10	N				地区	2023/12/10	
6	language	VARCHAR	10	N				语种	2023/12/10	
7	date	DATATIME		N				发布日期	2023/12/10	
8	num	DECIMAL	10	Y				评分人数	2023/12/10	
9	duration	INT		N				时长	2023/12/10	
10	alias	VARCHAR	255	Y				别名	2023/12/10	

①

字段名称：movie_id

字段描述：给每个电影进行编号，方便在其他实体中作为外码，因为使用 DECIMALmovie_id 数据占用空间大小相比直接存储地区名来得小，避免存储空间的浪费。

字段格式：该字段用来给各个地区进行编号，其格式为 1，2，3。

取值范围：[0,99]。

约束条件：primary key，即该字段为主码，同时也表明该字段是 NOT NULL 的。

②

字段名称：movie

字段描述：该字段存储的电影名称

字段类型：VARCHAR

取值范围：无。

约束条件：无。

缺省值：NULL

③

字段名称：grade

字段描述：该字段存储的电影评分

字段类型：VARCHAR

取值范围：无。

约束条件：无。

缺省值：无

④

字段名称：type

字段描述：该字段存储的电影的类型

字段类型：VARCHAR

取值范围：无。

约束条件：无。

缺省值：无

⑤

字段名称：area

字段描述：该字段存储的电影出品所在的地区

字段类型：VARCHAR

取值范围：无。

约束条件：无。

缺省值：无

⑥

字段名称：language

字段描述：该字段存储的电影的语言

字段类型：VARCHAR

取值范围：无。

约束条件：无。

缺省值：无

⑦

字段名称：date

字段描述：该字段存储的电影的出品日期

字段类型：DATATIME

取值范围：无。

约束条件：无。

缺省值：无

⑧

字段名称：num

字段描述：该字段存储的评分人数

字段类型：VARCHAR

取值范围：无。

约束条件：无。

缺省值：无

⑨

字段名称：alias

字段描述：该字段存储的电影的别名

字段类型：VARCHAR

取值范围：无。

约束条件：无。

缺省值：无

表名		director				说明	导演表			
序号	字段名称	字段类型	长度	允许空	缺省值	主外键	索引	字段描述	创建日期	备注
1	director_id	INT	10	N		PK		导演id	2023/12/10	从1开始，自增长
2	director	VARCHA	10	N				导演名	2023/12/	

		R						字	10	

①

字段名称：director_id

字段描述：该字段存储的导演序号

字段类型：INT

取值范围：无。

约束条件：主键

②

字段名称：director

字段描述：该字段存储的电影名称

字段类型：VARCHAR

取值范围：无。

约束条件：无。

缺省值：NULL

表名		performer				说明	演员表			
序号	字段名称	字段类型	长度	允许空	缺省值	主外键	索引	字段描述	创建日期	备注
1	performer_id	INT	10	N		PK		演员 id	2023/12/10	自增长从 1 开始
2	performer	VARCHAR	10	N				演员名字	2023/12/10	

①

字段名称：performer_id

字段描述：该字段存储的演员序号

字段类型：INT

取值范围：无。

约束条件：主键

②

字段名称：performer

字段描述：该字段存储的演员名称

字段类型：VARCHAR

取值范围：无。

约束条件：无。

缺省值：NULL

表名		scriptwriter				说明	编剧表		
序号	字段名称	字段类型	长度	允许空	缺省值	主外键	索引	字段描述	创建日期	备注
1	scriptwriter_id	INT	10	N		PK		编剧 id	2023/12/10	自增长从 1 开始
2	scriptwriter	VARCHAR	10	N				编剧名字	2023/12/10	

①

字段名称：scriptwriter_id

字段描述：该字段存储的编剧序号

字段类型：INT

取值范围：无。

约束条件：主键

②

字段名称：scriptwriter

字段描述：该字段存储的编剧名称

字段类型：VARCHAR

取值范围：无。

约束条件：无。

缺省值：NULL

2.6.2.存储过程(Stored Procedure)

数据库建立过程中，通过爬虫脚本直接将源数据存储到数据库；在后续的使用过程中，则通过管理员账户对数据库中的数据进行增删操作。

2.6.3.触发器(trigger)设计

本系统没有对数据库添加相关触发器。

2.7.系统使用方法设计

2.7.1.环境配置

本系统所需安装的环境依赖列举如下：

npm 8.19.2

nodejs v16.18.0

Google Chrome 108.0.5359.94

2.7.2.系统运行

①数据库建立

1）以根用户登录 MySQL，执行 create database moviedb;语句创建 moviedb 数据库，再执行 use moviedb 切换到该数据库

2）执行 sql 脚本 moviedb.sql，完成表的建立以及数据的插入

②后端运行

1）进入./BackEnd/resources/application.properties，将 user 和 password 值分别改为用户的用户名和密码

2）在./BackEnd/目录下执行 MovieApplication.java，开启后端程序

③前端运行

1）进入./FrontEnd，执行 npm run dev，开启前端进程

2) 在浏览器中输入 http://localhost:3000，即可进入用户界面，并使用本系统。

3.详细设计·

本阶段的主要目标是将概要设计阶段的构件设计进行详细分解，确定每个函数的算法

和流程，设计相关的辅助函数。

3.1.开发环境概述

1. 版本控制工具

git v2.35.1

2. 编译环境

- 硬件平台：Intel(R) Core(TM) i7-10510U CPU @ 1.80GHz　2.30 GHz
- 操作系统：Windows 10 家庭中文版
- 开发工具与环境：

前端 IDE：Webstorm v2022.2.3

后端 IDE：IntelliJ IDEA 2023.2.5

数据库可视化工具：Navicat Premium 16

3. 源程序目录

本系统的前端程序位于./FrontEnd 目录下，其目录结构为：

```
├─public
├─index.html
├─package.json
└─src
    ├─App.vue
    ├─main.js
    ├─api
    ├─assets
    │  ├─css
    │  └─img
    ├─components
    │  ├─Header.vue
    │  ├─Sidebar.vue
    │  └─Tags.vue
    ├─plugins
    ├─router
    ├─store
    ├─utils
    └─views
        ├─Home.vue
        ├─Login.vue
```

```
        ├─MovieList.vue
        ├─SeniorSearch.vue
        └─User.vue
```

后端程序位于./BackEnd 目录下，其目录结构为：

```
└─BackEnd
    ├─config
    ├─controller
    │      └─MovieController.java
    ├─demos
    │      ├─BasicController.java
    │      ├─PathVariableController.java
    │      └─User.java
    ├─entity
    │      ├─Director.java
    │      ├─Movie.java
    │      ├─Performer.java
    │      └─Scriptwriter.java
    ├─mapper
    │      ├─DirectorMapper.java
    │      ├─MovieMapper.java
    │      ├─PerformerMapper.java
    │      └─ScriptwriterMapper.java
    ├─service
    │      └─MovieService.java
    └─resources
```

3.2.构件(Component)详细设计

3.2.1.构件 1 电影搜索-前端构件

此构件可以获取数据库中所有电影信息，并且可以通过搜索特定电影名的方式进行全局搜索，在用户界面以分页表格形式展示电影的电影名、评分、类型、国家、语言、上映时间、时长、别名、导演、演员、编剧等信息。

3.2.1.1 源程序列表

./FrontEnd/src/views/Movielist.vue

3.2.1.2 设计思想

将前端的表格组件与列表类型的变量 TableData 绑定，让前端的电影信息能够随着信息的增删而实时改变；页面首次加载时，此构件使用 axios 工具向后端发起 GET 请求，获取电影数据后，将后端的相应结果写入 TableData 变量。

3.2.1.3 实现描述

对主要的函数，进行如下的描述：

3.2.1.3.1 函数 1

函数原型：onMounted(()=>{})

功能描述：在页面首次加载时，向后端发起针对电影信息列表的 GET 请求，并将结果写入 TableData 变量。

参数：输入参数为后端电影信息处理 URL（/selectAll），输出参数为前端的电影信息列表变量 TableData

返回值：电影信息列表（JSON 对象数组）

算法：

```
onMounted(() => {
        axios.get(localStorage.getItem("ip") + "selectAll").then(
            (response) =>{
              let list = response.data;
              TableData.value = list;
            }
        )
    })
```

3.2.1.3.2 函数 2

函数原型：handleSearchByName = () =>{}

功能描述：在页面首次加载时，向后端发起针对电影信息列表的 GET 请求，并将结果写入 TableData 变量。

参数：输入参数为后端电影信息处理 URL（/selectByName），输出参数为前端的电影信息列表变量 TableData

返回值：电影信息列表（JSON 对象数组）

算法：

```
handleSearchByName = () =>{
        axios.post(localStorage.getItem("ip") + 'selectByName', sendpara)
            .then((response) => {
              let list = response.data;
              TableData.value = list;
            })
      )
    })
```

3.2.2 构件 2 电影数据删除-前端构件

此构件根据用户删除的电影的属性值信息，向后端发起删除请求，删除这条电影记录。

3.2.2.1 源程序列表

./FrontEnd/src/views/Movielist.vue

3.2.2.2 设计思想

根据用户删除的电影信息找到对应的 JSON 对象，并通过 axios 工具生成包含该对象的删除电影 POST 请求并向后端相应的 url 发送。展示界面的数据更新由构件 1 完成。

3.2.2.3 实现描述

对主要的函数，进行如下的描述：

3.2.2.3.1 函数 1

函数原型：handleDelete = (index, row) => {}

功能描述：根据待删除电影的下标和电影信息对应的 JSON 对象构造 POST 请求，并向后端处理电影删除的 URL 发起该请求。

参数：输入参数为待删除电影的下标 index、电影信息对应的 JSON 对象 row，后端电影删除处理 URL（/delete），无输出

返回值：删除电影操作运行结果（成功/失败）

异常处理：若删除失败，则弹出“删除失败”窗口

算法：

```
const handleDelete = (index, row) => {
        ElMessageBox.confirm("确定要删除吗？", "提示", {
            type: "warning",
        })
            .then(() => {
                let sendpara = qs.stringify(row);
                axios.post(localStorage.getItem("ip") + "delete", sendpara).then(
                    (response) => {
                        console.log(row);
                        let dt = response.data;
                        if (dt) {
                            ElMessage.success("删除成功");
                            this.$router.go(0);
                        } else {
                            ElMessage.error("删除失败");
                            return false;
                        }
                    }
                )
            })
            .catch(() => {});
        getData();
    };
```

3.2.3 构件 3 电影数据添加-前端构件

该构件根据用户输入的添加电影的信息，向后端发起添加该条电影记录的请求。

3.2.3.1 源程序列表

./FrontEnd/src/views/Movielist.vue

3.2.3.2 设计思想

将用户界面的添加电影表单内容与本构件描述电影信息的结构体变量 Movie 绑定，用户确认添加后，通过 axios 工具，由 Movie 变量的当前值生成 POST 请求，通知后端进行电影数据新增操作。此后展示界面的数据更新由构件 1 完成。

3.2.3.3 实现描述

对主要的函数，进行如下的描述：

3.2.3.3.1 函数 1

函数原型：saveEdit = (Movie) => {}

功能描述：根据新增电影信息结构体 Movie 构造 POST 请求，并向后端处理电影新增的 URL 发起该请求。

参数：输入参数为新增电影信息结构体 Movie，后端电影添加处理 URL（/insert），无输出

返回值：电影添加操作运行结果（成功/失败）

异常处理：若后端操作失败，则弹出“添加失败”窗口;若填写信息不足，则弹出“请填写完整信息”窗口。

算法：

```
const saveEdit = (form) => {
                    editVisible.value = false;
                   formRef.value.validate((valid) =>{
                     if (valid){
                          Movie.date = `${Movie.date.substr(0, 4)}-${Movie.date.substr(4, 2)}-
${Movie.date.substr(6, 2)} 00:00:00`;
                          let sendpara = qs.stringify(Movie);
                          axios.post(localStorage.getItem("ip") + "insert", sendpara).then(
                               (response) =>{
                                    console.log(Movie);
                                    let dt = response.data;
```

```
                    if (dt){
                        ElMessage.success("添加成功");
                    } else {
                        ElMessage.error("添加失败");
                        return false;
                    }
                },
                function (err){
                    ElMessage.error("添加失败");
                    console.log(err);
                });
        } else {
            console.log(valid);
            ElMessage.error("请填写完整信息");
            return false;
        }
    });
    });
```

3.2.4 构件 4 个性化查询-前端构件

该构件根据用户选取的类型、国家、语言和评分属性值，发起个性化查询请求并将符合条件的电影展示在用户界面。

3.2.4.1 源程序列表

./FrontEnd/src/views/SeniorSearch.vue

3.2.4.2 设计思想

在本构件中使用包含 type、country、language、grade 四个属性的 query 结构体，实时接收用户界面的类型、语言、国家、评分下拉栏的选择内容，用户选择个性化信息进行查询，根据 query 结构体的当前值，调用 axios 工具发送包含 query 参数的 POST 请求，并将后端返回的查询结果写入本构件的电影信息列表变量 TableData，进而实时展示在用户界面。

3.2.4.3 实现描述

对主要的函数，进行如下的描述：

3.2.4.3.1 函数 1

函数原型：handleSearch = (query) => {}

功能描述：根据 query 结构体中包含的电影类型、国家、语言、评分信息构造 POST 请求并向后端处理个性化查询请求的 URL 发送该请求，将后端返回结果写入本构件的电影信息列表 TableData。

参数：输入参数为待查询内容 query、后端个性化查询处理 URL（/personizedSelect），输出参数为电影信息列表 TableDadata。

返回值：电影信息列表（JSON 对象数组）

算法：

```
const handleSearch = (query) => {
        query.pageIndex = 1;
        let sendpara = qs.stringify(query);
        axios.post(localStorage.getItem("ip") + "personalizedSelect", sendpara).then(
            function (response){
                let list = response.data
                TableData.value = list;
                tableData.value = list.slice(
                   query.pageSize*(query.pageIndex-1),
                   query.pageSize*(query.pageIndex));
                pageTotal.value = list.length;
            }
            );
        getData();
    };
```

3.2.5 构件 5 电影评分-前端构件

该构件用于支持用户对电影的评分需求，将用户对某一电影的评分发送给后端，并在用户界面展示更新后的平均评分。

3.2.5.1 源程序列表

./FrontEnd/src/views/SeniorSearch.vue

3.2.5.2 设计思想

在本构件中，先接收来自用户的评分输入，由包含电影名称与评分的结构体 send_grade 的当前值构造 POST 请求，通知后端更新该条电影的评分人数和平均评分。前端显示的该条电影的评分可由构件 1 完成更新。

3.2.5.3 实现描述

对主要的函数，进行如下的描述：

3.2.5.3.1 函数 1

函数原型：const saveEdit = (send_grade) => {}

功能描述：根据 send_grade 中包含的电影名称和电影数量信息，调用 axios 工具构造 POST 请求，并向后端处理电影评分请求的 URL 发送该请求，

参数：输入参数为电影名和用户评分构成的结构体 send_grade、后端电影评分处理 URL（/update），无输出参数

返回值：电影评分的执行状态（成功/失败）

异常处理：若评分操作失败，则弹出“评分失败”窗口。

算法：

```
const saveEdit = (send_grade) => {
        editVisible.value = false;
        send_grade.grade = my_grade.value * 2;
        let sendpara = qs.stringify(send_grade);
        axios.post(localStorage.getItem("ip") + "update", sendpara).then(
            (response) => {
                let dt = response.data;
                if (dt){
                    ElMessage.success("评分成功");
                } else {
                    ElMessage.error("评分失败");
                    return false;
                }
            },
            function (err){
                ElMessage.error("评分失败");
                console.log(err);
```

```
        });
    };
```

3.2.6 构件 6 电影搜索-后端处理构件

该构件在监听到来自前端的电影全局搜索 HTTP 请求后，即查询数据库并将数据库中所有电影信息返回至前端。

3.2.6.1 源程序列表

./BackEnd/controller/MovieController.java

3.2.6.2 设计思想

在本构件中，使用 request 变量存储接收到的 GET 请求，并构造对数据库相应表的查询 SQL 语句，将查询结果存入电影信息结构体列表，并转化为 JSON 格式返回给前端。

3.2.6.3 实现描述

对主要的函数，进行如下的描述：

3.2.6.3.1 函数 1

函数原型：selectAll()

功能描述：在监听到前端对/selectAll 这个 url 的 GET 请求后，构造相应的针对数据库 Movie、Director、Performer、Scriptwriter 表的自然连接查询 SQL 语句，并将结果以 JSON 格式变量 JsonResponse 返回给前端。

参数：输入参数为/selectAll 这一 url 监听到的 GET 请求，无输出参数。

返回值：包含所有电影信息的 JSON 格式变量 JsonResponse。

算法：

```
@Mapper
public interface MovieMapper extends BaseMapper<Movie> {
    @Select("select * from movie")
    @Results(
            {
                    @Result(column = "movie_id", property = "movieId"),
```

```
            @Result(column = "movie", property = "movie"),
            @Result(column = "grade", property = "grade"),
            @Result(column = "type", property = "type"),
            @Result(column = "country", property = "country"),
            @Result(column = "language", property = "language"),
            @Result(column = "date", property = "date"),
            @Result(column = "duration", property = "duration"),
            @Result(column = "alias", property = "alias"),
            @Result(column = "grade_num", property = "gradeNum"),
            @Result(column = "movie_id", property = "directorList", javaType = List.class,
                    many = @Many(select = "com.example.movie.mapper.DirectorMapper.selectDirector")),
            @Result(column = "movie_id", property = "performerList", javaType = List.class,
                    many = @Many(select = "com.example.movie.mapper.PerformerMapper.selectPerformer")),
            @Result(column = "movie_id", property = "scriptwriterList", javaType = List.class,
                    many = @Many(select = "com.example.movie.mapper.ScriptwriterMapper.selectScriptwriter"))
        }
    )
    List<Movie> selectAll();
```

3.2.6.3.2 函数 2

函数原型：selectByName()

功能描述：在监听到前端对/selectByName 这个 url 的 GET 请求后，构造相应的针对数据库 Movie、Director、Performer、Scriptwriter 表的自然连接查询 SQL 语句，并将结果以 JSON 格式变量 JsonResponse 返回给前端。

参数：输入参数为/selectByName 这一 url 监听到的 GET 请求，无输出参数。

返回值：包含所有电影信息的 JSON 格式变量 JsonResponse。

算法：

```
    @Select("select * from movie where movie like CONCAT('%', #{movie}, '%')")
        @Results(
                {
                        @Result(column = "movie_id", property = "movieId"),
                        @Result(column = "movie", property = "movie"),
                        @Result(column = "grade", property = "grade"),
                        @Result(column = "type", property = "type"),
                        @Result(column = "country", property = "country"),
```

```
                @Result(column = "language", property = "language"),
                @Result(column = "date", property = "date"),
                @Result(column = "duration", property = "duration"),
                @Result(column = "alias", property = "alias"),
                @Result(column = "grade_num", property = "gradeNum"),
                @Result(column = "movie_id", property = "directorList", javaType =
List.class,
                        many = @Many(select =
"com.example.movie.mapper.DirectorMapper.selectDirector")),
                @Result(column = "movie_id", property = "performerList", javaType =
List.class,
                        many = @Many(select =
"com.example.movie.mapper.PerformerMapper.selectPerformer")),
                @Result(column = "movie_id", property = "scriptwriterList", javaType =
List.class,
                        many = @Many(select =
"com.example.movie.mapper.ScriptwriterMapper.selectScriptwriter"))
            }
    )
    List<Movie> selectByName(@Param("movie") String movie);
```

3.2.7 构件 7 电影数据添加-后端处理构件

该构件负责监听来自前端的电影添加 HTTP 请求，根据请求中包含的电影信息，在数据库中添加电影记录。

3.2.7.1 源程序列表

./BackEnd/controller/MovieController.java

3.2.7.2 设计思想

在本构件中，根据 POST 请求中包含的电影各属性值构造相应的数据插入 SQL 语句，并将执行的成功/失败信息通知前端。

3.2.7.3 实现描述

对主要的函数，进行如下的描述：

3.2.7.3.1 函数 1

函数原型：insert()

功能描述：在监听到前端对/insert 这个 url 的 POST 请求后，解析请求中包含的电影信息各属性值，并构造在 Movie、Director、Performer、Scriptwriter 表以及相应的三个关联实体表中新增记录的 SQL 语句，执行结束后通过 JsonResponse 变量向前端返回执行的成功/失败信息。

参数：输入参数为/insert 这一 url 监听到的 POST 请求，无输出参数。

返回值：包含插入电影操作执行成功/失败消息的 JSON 格式变量 JsonResponse。

算法：

```
private ScriptwriterMapper scriptwriterMapper;

    public int save(Movie movie){

        int i = movieMapper.insert(movie);

        int movieId = movie.getMovieId();

        List<Director> directorList = movie.getDirectorList();

        List<Performer> performerList = movie.getPerformerList();
```

```
List<Scriptwriter> scriptwriterList = movie.getScriptwriterList();
for (Director director: directorList){
    List<Director> director1 = directorMapper.selectByName(director.getDirector());
    if (director1.size() != 0){
        directorMapper.insertdir(movieId, director1.get(0).getDirectorId());
    }else {
        directorMapper.insert(director);
        directorMapper.insertdir(movieId, director.getDirectorId());
    }
}
for (Performer performer: performerList){
    List<Performer> performer1 =
 performerMapper.selectByName(performer.getPerformer());
    if (performer1.size() != 0){
        performerMapper.insertper(movieId, performer1.get(0).getPerformerId());
    }else {
        performerMapper.insert(performer);
        System.out.println(movieId);
        performerMapper.insertper(movieId, performer.getPerformerId());
    }
}
for (Scriptwriter scriptwriter : scriptwriterList){
    List<Scriptwriter> scriptwriter1 =
 scriptwriterMapper.selectByName(scriptwriter.getScriptwriter());
    if(scriptwriter1.size() != 0){
```

```
                scriptwriterMapper.insertscr(movieId, scriptwriter1.get(0).getScriptwriterId());
            }else {
                scriptwriterMapper.insert(scriptwriter);
                scriptwriterMapper.insertscr(movieId, scriptwriter.getScriptwriterId());
            }
        }
        return i;
    }
```

3.2.8 构件 8 电影数据删除-后端处理构件

该构件负责监听来自前端的电影删除 HTTP 请求，根据待删除的电影名称，发起在数据库中删除相应电影记录的操作。

3.2.8.1 源程序列表

./BackEnd/controller/MovieController.java

3.2.8.2 设计思想

在本构件中，使用 request 变量存储接收到的 POST 请求，并根据 POST 请求中包含的电影名称属性值构造相应的数据删除 SQL 语句，并将执行的成功/失败信息通知前端。

3.2.8.3 实现描述

对主要的函数，进行如下的描述：

3.2.8.3.1 函数 1

函数原型：delete()

功能描述：在监听到前端对/delete 这个 url 的 POST 请求后，解析请求中包含的电影名称属性值，并构造针对 Movie 表相应电影记录的删除 SQL 语句交由数据

库执行。由于数据库设置了级联删除，该处无需针对涉及的关联实体构造删除SQL 语句。数据删除的执行结果（成功/失败）通过 JsonResponse 变量向前端返回。

参数：输入参数为/delete 这一 url 监听到的 POST 请求，无输出参数。

返回值：包含删除电影操作执行成功/失败消息的 JSON 格式变量 JsonResponse。

算法：

```
@Delete("delete from movie where movie_id = #{movieId}")
    int delete(@Param("movieId") int movieId);
@PostMapping("/delete")
    public String delete(int movieId){
        System.out.println(movieId);
        int i = movieMapper.delete(movieId);
        if (i > 0){
            return "删除成功";
        }else {
            return "删除失败";
        }
    }
```

3.2.9 构件 9 个性化查询-后端处理构件

该构件负责监听来自前端的个性化查询 HTTP 请求，根据请求中包含的电影类型、语言、国家和评分区间，在数据库中查询符合条件的电影记录并返回给前端。

3.2.9.1 源程序列表

./BackEnd/controller/MovieController.java

3.2.9.2 设计思想

在本构件中，使用 request 变量存储接收到的 POST 请求，并根据 POST 请求中包含的类型、语言、国家、评分属性值构造针对 Movie 表的相应 SQL 语句，查询完成后将结果返回给前端。

3.2.9.3 实现描述

对主要的函数，进行如下的描述：

3.2.9.3.1 函数 1

函数原型：personlizedSelect()

功能描述：在监听到前端对/personizedSelect 这个 url 的 POST 请求后，解析该请求中包含的 type、language、country、grade 四个属性值，并构造相应的针对 Movie 表的查询 SQL 语句，并将查询得到的符合条件的电影信息列表通过 JsonResponse 变量向前端返回。

参数：输入参数为/personizedSelect 这一 url 监听到的 POST 请求，无输出参数。

返回值：包含属性值匹配的电影信息列表的 JSON 格式变量 JsonResponse。

算法：

```
    @PostMapping("/personalizedSelect")
        public List<Movie> personalizedSelect(String type, String country, String language,
String grade){
            double[] gradeNum= new double[2];
            if (grade.equals("不限")){
                gradeNum[0]=0;
                gradeNum[1]=10;
            } else if (grade.equals("6.0 以下")) {
                gradeNum[0]=0;
                gradeNum[1]=6;
            }else {
                String[] s=grade.split("~");
                gradeNum[0]=Double.parseDouble(s[0]);
                gradeNum[1]=Double.parseDouble(s[1]);
            }
            List<Movie> movieList = movieService.personalizedSelect(type, country, language,
gradeNum);
            return movieList;
        }
    @Select("select * from movie where grade >= #{gradeLow} and grade <=
#{gradeHigh}${Sql}")
        @Results(
                {
                        @Result(column = "movie_id", property = "movieId"),
                        @Result(column = "movie", property = "movie"),
```

```
                @Result(column = "grade", property = "grade"),
                @Result(column = "type", property = "type"),
                @Result(column = "country", property = "country"),
                @Result(column = "language", property = "language"),
                @Result(column = "date", property = "date"),
                @Result(column = "duration", property = "duration"),
                @Result(column = "alias", property = "alias"),
                @Result(column = "grade_num", property = "gradeNum"),
                @Result(column = "movie_id", property = "directorList", javaType =
List.class,
                        many = @Many(select =
"com.example.movie.mapper.DirectorMapper.selectDirector")),
                @Result(column = "movie_id", property = "performerList", javaType =
List.class,
                        many = @Many(select =
"com.example.movie.mapper.PerformerMapper.selectPerformer")),
                @Result(column = "movie_id", property = "scriptwriterList", javaType =
List.class,
                        many = @Many(select =
"com.example.movie.mapper.ScriptwriterMapper.selectScriptwriter"))
            }
    )
    List<Movie> personalizedSelect(@Param("gradeLow") double gradeLow,
@Param("gradeHigh") double gradeHigh, @Param("Sql") String Sql);
```

3.2.10 构件 10 电影评分-后端处理构件

该构件负责该构件负责监听来自前端的电影评分 HTTP 请求，根据所评分的电影名称和评分值，发起更新数据库中该条电影记录的平均评分值的操作。

3.2.10.1 源程序列表

./BackEnd/controller/MovieController.java

3.2.10.2 设计思想

在本构件中，使用 request 变量存储接收到的 POST 请求，并根据 POST 请求中包含的电影名称和评分值构造相应的数据更新 SQL 语句，并将执行的成功/失败信息通知前端。

3.2.10.3 实现描述

对主要的函数，进行如下的描述：

3.2.10.3.1 函数 1

函数原型：update()

功能描述：在监听到前端对/update 这个 url 的 POST 请求后，解析请求中包含的电影名称 name 和评分值 grade，首先构造查询 SQL 语句获取该电影当前的评分人数和平均评分，根据当前的用户评分值计算新的评分人数和平均评分，并通过数据更新 SQL 语句写入该条电影记录。此后通过 JsonResponse 变量向前端返回更新操作的成功/失败信息。

参数：输入参数为/update 这一 url 监听到的 POST 请求，无输出参数。

返回值：包含更新评分操作执行成功/失败消息的 JSON 格式变量 JsonResponse。

算法：

```
public int update(int movieId, int grade){
        Movie movie = movieMapper.selectById(movieId);
        double gradeOld = movie.getGrade();
        int gradeNumOld = movie.getGradeNum();
        int gradeNumNew = gradeNumOld + 1;
        double gradeNew = (gradeOld * gradeNumOld + grade) / gradeNumNew;
        return movieMapper.updateGrade(gradeNew,gradeNumNew,movieId);
    }

@PostMapping("/update")
    public String update(Integer movieId, Integer grade){
        int i = movieService.update(movieId, grade);
        if (i > 0){
            return "更新成功";
        }else {
            return "更新失败";
        }
    }
```

3.2.11 构件 11 数据库管理系统构件

该构件负责具体执行后端程序构造的 SELECT、INSERT、UPGRADE、DELETE 语句，查询语句所要求的内容返回给后端程序，或者根据语句要求对数据库的数据进行相应改变。

3.2.11.1 源程序列表

用户本地的 MySQL 应用程序

3.2.11.2 设计思想

首先根据物理设计，使用 MySQL 进行表建立、约束关系定义、电影数据导入等操作，此后运行 MySQL 服务，即可正常地响应后端程序构造的各类 SQL 语句，并对数据库中的电影信息数据进行相应的操作。

3.2.11.3 实现描述

对主要的函数，进行如下的描述：

3.2.11.3.1 函数 1 查询所有电影 SQL 语句

函数原型：SELECT 语句

功能描述：查询所有的电影记录

参数：无

返回值：电影信息元组构成的集合

算法：select * from movie;

3.2.11.3.2 函数 2 插入电影信息 SQL 语句

函数原型：INSERT 语句

功能描述：根据新增的电影信息，在表中新增记录，同时在三个关联实体中增加相应记录。

参数：输入为 movie_id,director_id, performer_id, scriptwriter_id;无输出参数。

返回值：执行状态（成功/失败）

算法：

insert into dir_movie(movie_id, director_id) values (#{movieId},#{directorId})

insert into per_movie(movie_id, performer_id) values (#{movieId},#{performerId})

insert into scr_movie(movie_id, scriptwriter_id) values (#{movieId},#{scriptwriterId})

insert into movie(movie_id,movie,grade,type,country,`language`, `date`,duration,

alias,grade_num) values ({},{},{},{},{},{},{},{},{},{});

insert into director(director_id, director)values ({},'{}');

insert into performer(performer_id, performer)values ({},'{}');

insert into scriptwriter(scriptwriter_id, scriptwriter)values ({},'{}');

3.2.11.3.3 函数 3 删除电影信息 SQL 语句

函数原型：DELETE 语句

功能描述：删除某一电影记录

参数：输入参数为待删除电影的 movie_id 属性，无输出参数

返回值：删除语句执行结果（成功/失败）

算法：delete from movie where movie_id = #{movieId};

3.2.11.3.4 函数 4 电影信息个性化查询 SQL 语句

函数原型：SELECT 语句

功能描述：根据国家、类型、语言的值以及分数范围查询符合要求的电影元组

参数：输入参数为 country、type、language、grade 属性值；无输出参数。

返回值：属性值匹配的电影信息元组构成的集合

算法：select * from movie where grade >= #{gradeLow} and grade <=

#{gradeHigh}${Sql};

3.2.11.3.5 函数 5 评分信息更新 SQL 语句

函数原型：UPDATE 语句

功能描述：向被评分电影元组分别写入用户评分后的新平均评分和新评分人数。

参数：输入参数为被评分电影的 movieID 属性以及更新后的 grade、grade_num 属性；无输出参数。

返回值：执行状态（成功/失败）

算法：

update movie set grade=#{gradeNew},grade_num=#{gradeNumNew} where movie_id = #{movieId};

3.2.11.3.6 函数 6 查询特定名称电影 SQL 语句

函数原型：SELECT 语句

功能描述：查询所有的电影记录

参数：无

返回值：电影信息元组构成的集合

算法：select * from movie where movie like CONCAT('%', #{movie}, '%');