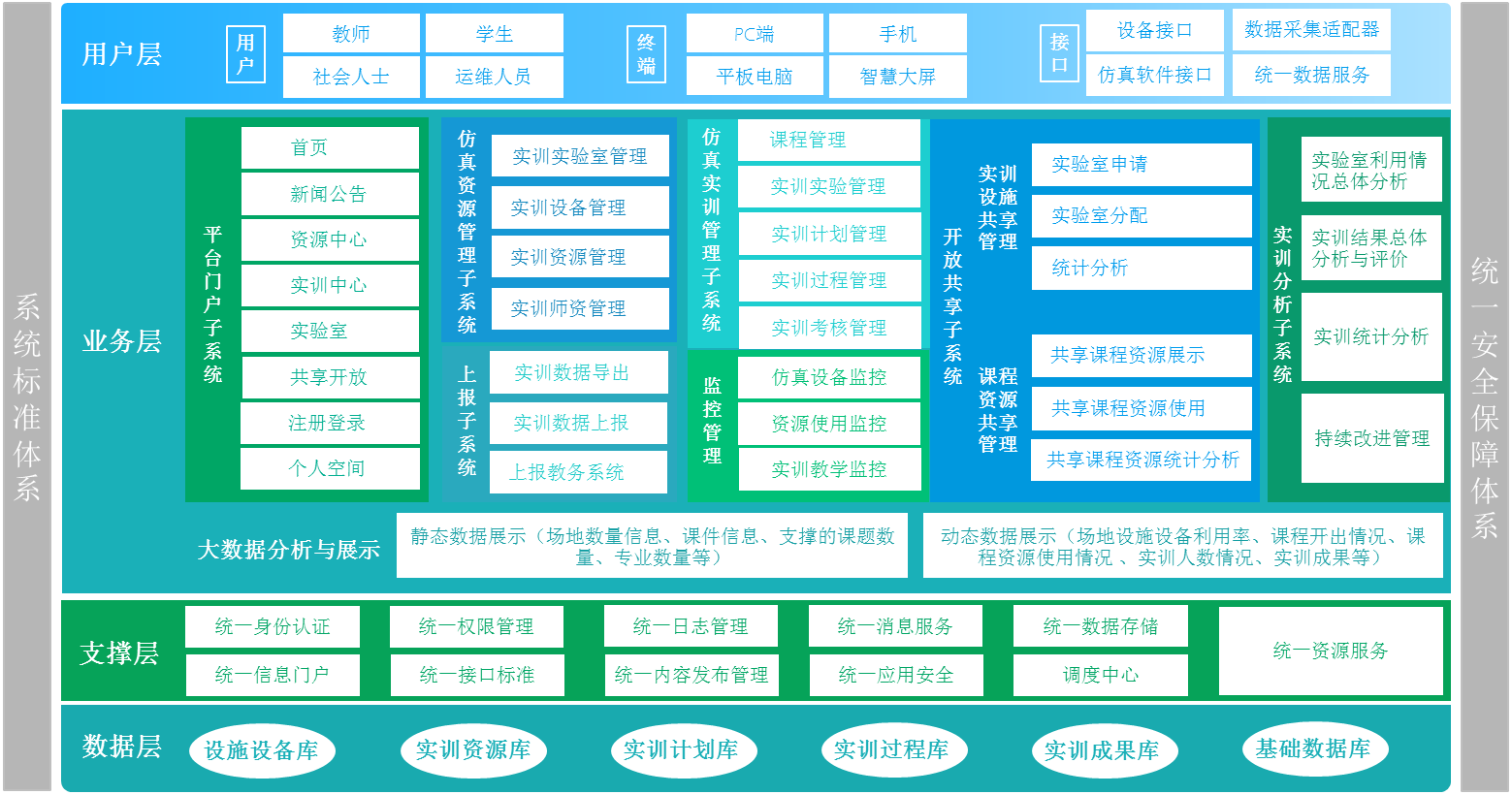
# 项目概述

虚拟仿真实训教学管理及资源共享云平台采用云计算、大数据、互联网、人工智能等新兴技术的融合及创新应用，对学校虚拟仿真实训教学场所、虚拟仿真实训设施设备和虚拟仿真实训资源进行跨专业、跨院校、跨地域的统筹管理，实现学校全方位虚拟仿真实训过程跟踪、实训资源互联互通、实训场景全覆盖、实训基地智慧管理所需要的多维度、系统化、智能化、数据化的一站式服务。

虚拟仿真实训教学管理及资源共享云平台项目业务需求包括：新闻公告、实验室、仿真设备、实训资源、教师管理、实验室申请、实验老师排课、实训课程、实训实验、学生管理、校本资源管理、教师发展管理、建设成效管理、课程共享管理、人才培养管理、成绩管理、角色管理、菜单管理、权限管理、设备监控管理、资源监控管理、教学监控管理、数据字典管理、消息管理、用户认证等功能。

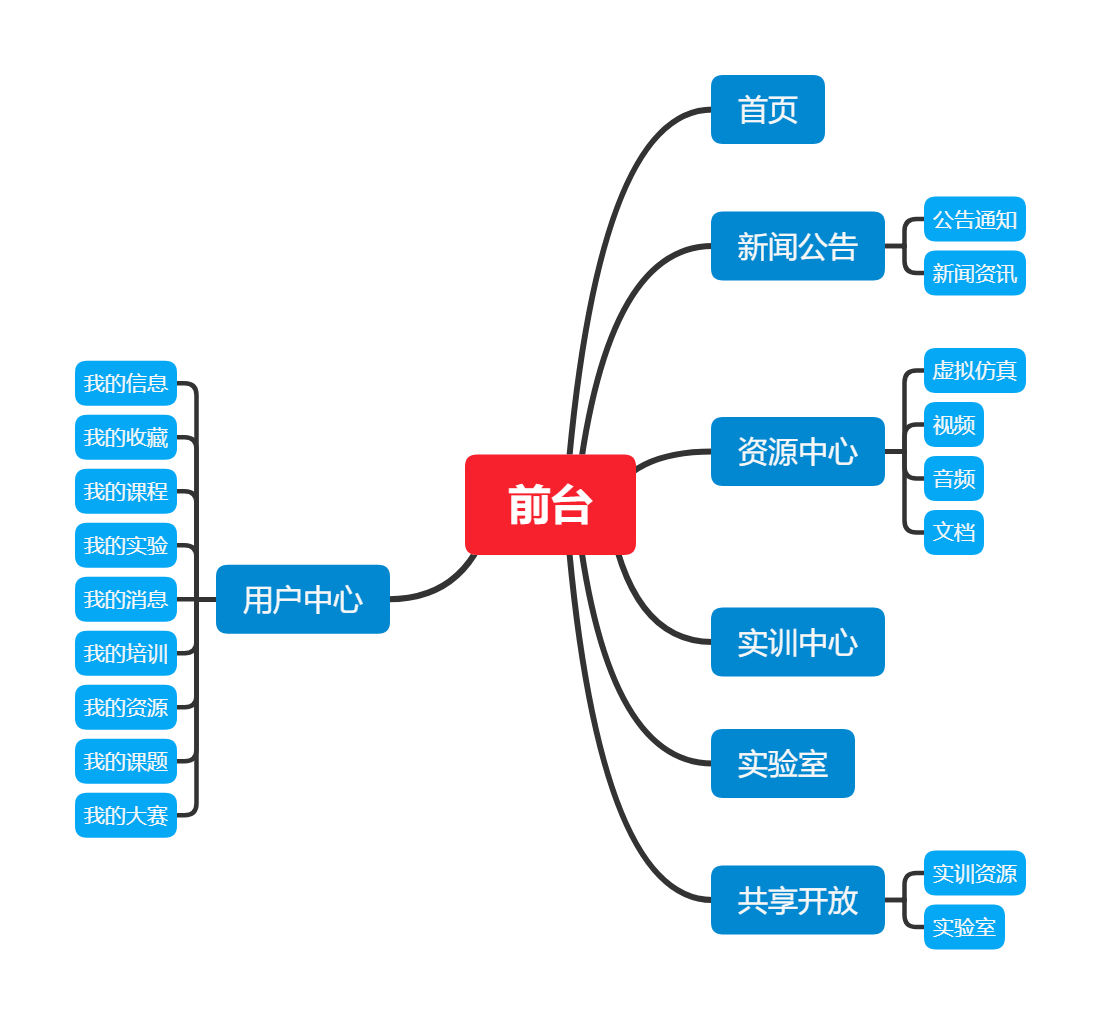
## 功能框架



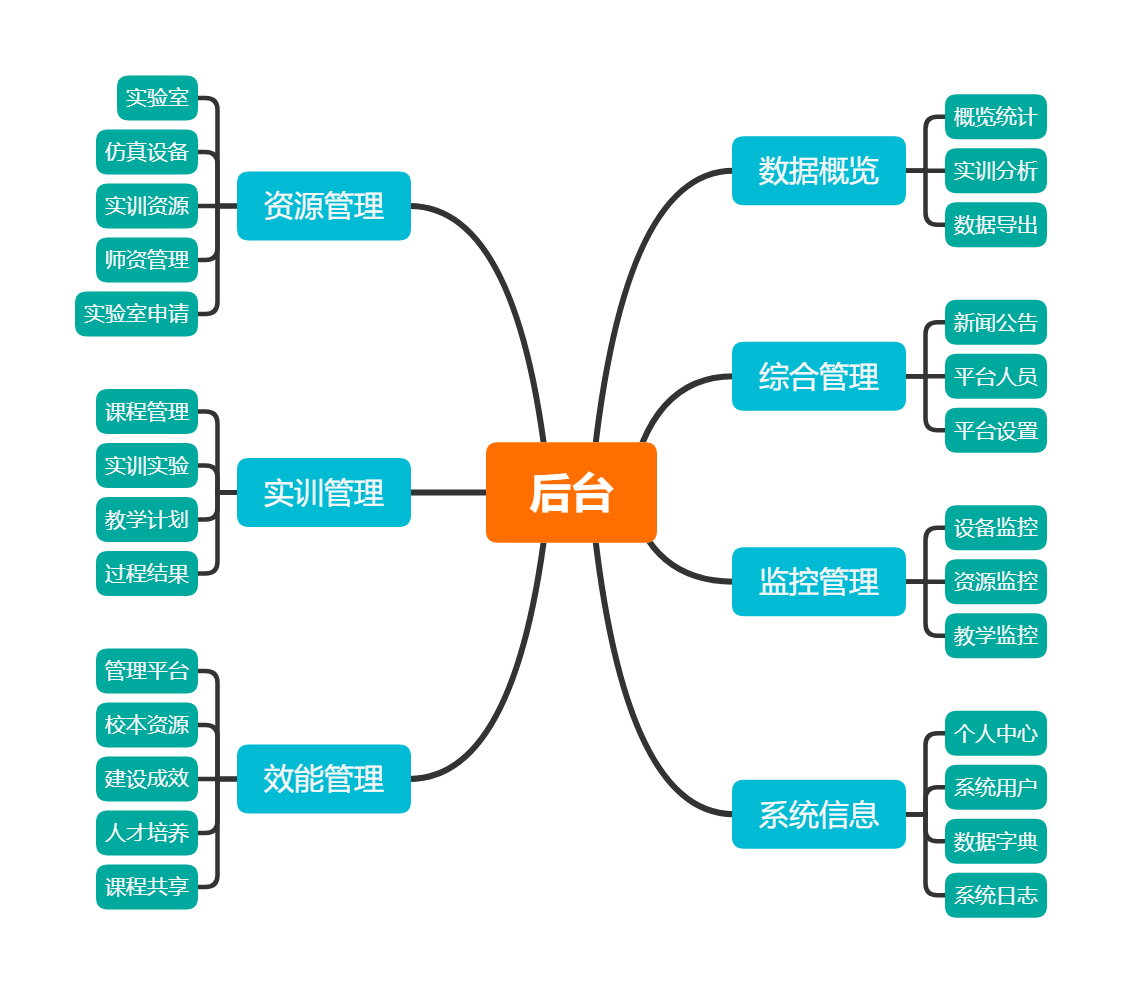
## 平台子系统



## 系统前台



## 系统后台



## 核心业务



## 实验室管理



## 实验管理



## 实验评分



## 数据管理



## 数据大屏



## 开放共享



## 平台特点



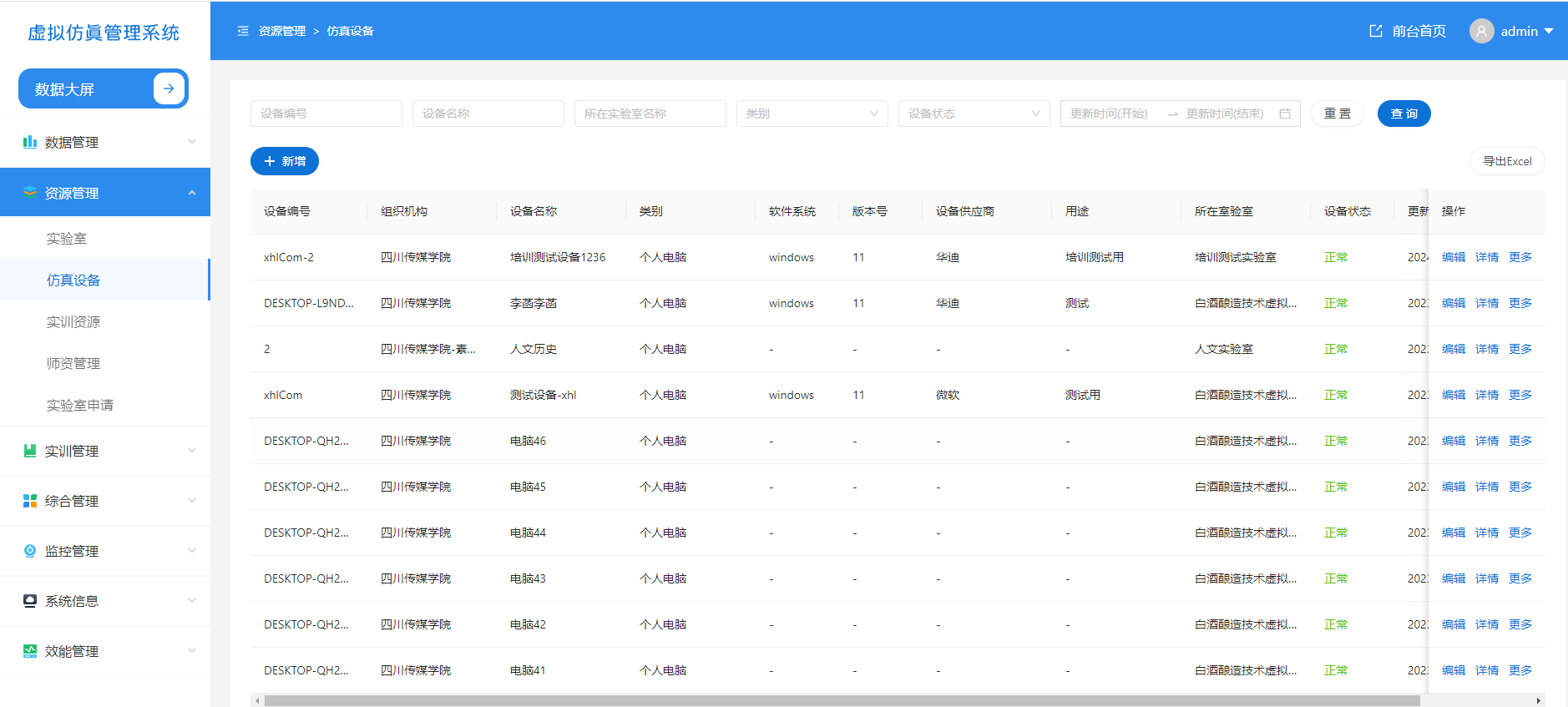
# 案例介绍

本次以虚拟仿真实训教学管理及资源共享云平台仿真设备为案例，给同学们讲解仿真设备业务的统一过程(UP),分别经历业务模型、分析模型、设计模型、实现模型、测试模型环节，最后完成仿真设备业务案例。

## 案例引入

我们可以从实际商业项目中进行了解和分析，实际项目的访问地址：<http://whcm.hwadee.cn/xf/#/portal/home/index。>

在虚拟仿真实训教学管理及资源共享云平台门户首页中，使用用户账号和密码，进行身份鉴权后，登录进行后台管理，选择进入仿真设备业务，如下图：



同学们，我们可以从实际的效果图中进行分析、整理，来开展仿真设备业务分析与实现。

## 技术栈

|  |
| --- |
| 1.工具：**后端(IntelliJ IDEA Community Edition)、前端(HBuilder)**  2.后端技术框架：**Springboot、SpringMVC、Mybatis/MybatisPlus**  3.前端框架：**VUE、Element-UI**  4.数据库：**Mysql**  5.中间件：**Tomcat**(独立或者Springboot内部集成)  6.系统：Windows、Linux |

## 业务模型

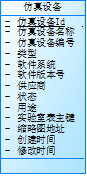
概念：领域模型是对领域内的概念类或现实世界中对象的可视化表示。领域模型也称为概念模型，领域对象模型和分析对象模型。领域模型建模的关键就是找出业务领域中关键的事物（实体）以及他们之间的关系。

首先我们需要明确该仿真设备业务是提供给谁使用？仿真设备主要是提供给实验室管理员使用，找到业务领域中的使用者(即：用户类型)。

其次我们需要分析仿真设备业务需要完成那些业务操作。通过对业务的理解，我们可以整理归纳出是对仿真设备的基础信息进行维护，包括：新增仿真设备、编辑仿真设备、删除仿真设备、查看仿真设备详情、查询仿真设备、导出仿真设备。

最后我们可以对仿真设备维护涉及到仿真设备属性进行整理，得出大概需要考虑到的数据项有（设备编码、缩略图、设备名称、类别、软件系统、版本号、设备供应商、用途、所在实验室、设备状态），以为后期分析设计作准备。

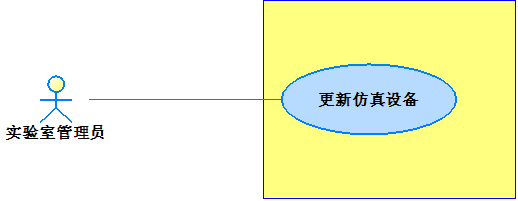
通过上面业务领域的分析，我们可以分析出仿真设备的领域模型如下：



## 用例模型

### 用例图

基于上节中的业务分析，我们可以使用用例模型来进行图形化描述，就是表述使用者和业务的关系，确定业务功能边界。使用者：实验室管理员，业务：通过对仿真设备的业务领域分析，我们是对仿真设备进行维护（增、删、改、查、导出），即我们可以总结就是对仿真设备数据的更新，从而得出用例名称为更新仿真设备。用例图如下：



### 用例描述（用例文本）

仿真设备Story：该功能提供给实验室管理员用户使用。用于仿真设备的基础信息进行维护。包括：新增仿真设备、编辑仿真设备、删除仿真设备、查看仿真设备详情、查询仿真设备、导出仿真设备。

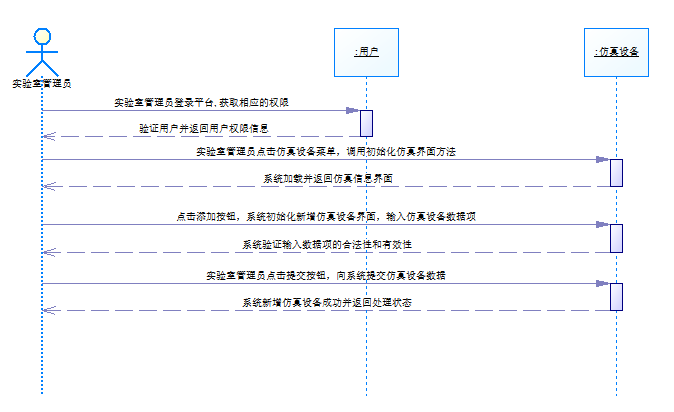
|  |  |
| --- | --- |
| **用例名** | 更新仿真设备 |
| **简要描述** | 该功能提供给实验室管理员用户使用。用于仿真设备的基础信息进行维护。包括：新增仿真设备、编辑仿真设备、删除仿真设备、查看仿真设备详情、查询仿真设备、导出仿真设备 |
| **参与者** | 实验室管理员 |
| **涉众** | 实验室管理员：用于实现对仿真设备的管理 |
| **相关用例** | 设备类型 |
| **前置条件** | 实验室管理员用户已经通过登陆权限认证，进入仿真设备管理界面 |
| **后置条件** | 系统存储的仿真设备信息 |
| **基本事件流**   1. 当实验室管理员仿真设备成功，并选择仿真设备菜单下的仿真设备管理时，启动本用例 2. 根据用户的操作请求，分别执行下列子事件流：   -当实验室管理员进行新增仿真设备操作时，执行“新增仿真设备”子流（A-1）；  -当实验室管理员进行编辑仿真设备信息操作时，执行“编辑仿真设备信息”子流（A-2）；  -当实验室管理员进行删除仿真设备操作时，执行“删除仿真设备”子流（A-3）；  -当实验室管理员进行查看仿真设备信息操作时，执行“查看仿真设备信息”子流（A-4）；  -当实验室管理员进行查询仿真设备信息操作时，执行“查询仿真设备信息”子流（A-5）；  -当实验室管理员进行导出仿真设备信息操作时，执行“导出仿真设备信息”子流（A-6）。   1. 系统保存或者更新信息。用例结束 | |
| **备选事件流**   1. \* 实验室管理员在选择仿真设备菜单前，随时可以终止该用例   A-1 新增仿真设备   1. 实验室管理员点击新增，填写新增仿真设备信息，如设备编码(B-1)(C-1)、缩略图、设备名称、类别等（D-1）； 2. 点击提交；系统验证实验室管理员输入数据的有效性和合法性（C-1）； 3. 实验室管理员点击保存操作，系统新增新一条仿真设备信息(C-2)。   A-2 编辑仿真设备信息  1) 实验室管理员选择某一仿真设备信息列表后的编辑按钮；  2) 系统跳转到仿真设备信息编辑界面，加载并显示被选仿真设备信息详情（D-1）；  3) 实验室管理员在仿真设备编辑界面中，对仿真设备信息的进行编辑；   1. 点击提交；系统验证实验室管理员输入数据的有效性和合法性（C-1）； 2. 实验室管理员点击提交操作，系统更新仿真设备信息(C-2)。   A-3 删除仿真设备  1) 实验室管理员在仿真设备信息列表中，选择某一仿真设备后的删除按钮；系统弹出“是否确定删除该条仿真设备信息”提示框；  2) 实验室管理员点击“确认”操作，系统从数据库删除指定仿真设备的数据。如果选择“取消”，则返回仿真设备列表页面。  A-4 查看仿真设备信息  1) 实验室管理员选择某一仿真设备信息列表后的查看按钮；  2) 系统跳转到仿真设备详细界面，加载该条仿真设备数据，在仿真设备详细界面以表单的形式显示信息；  A-5 查询仿真设备信息  1) 实验室管理员输入设备名称、设备编号查询条件，点击查询按钮；  2) 系统以列表显示符合查询条件的仿真设备信息(C-3)。  A-6 导出仿真设备信息  1) 实验室管理员输入查询条件(D-2)，点击查询按钮；  2) 系统以列表显示符合查询条件仿真设备信息(C-3)。  3) 实验室管理员点击导出按钮，系统生成对应的excel格式仿真设备数据。  C-\* 如果实验室管理员输入的数据格式不符合规范  C-1 设备编码数据格式不符合规范   1. 系统在校验设备编码数据合法性时，如果实验室管理员未输入设备编码，则提示“请输入设备编码”，也可以选择结束该用例 2. 如果实验室管理员所填项设备编码格式不符合规范，则提示“请按xxx格式输入” 3. 实验室管理员可以重新输入设备编码，也可以选择结束该用例   C-2 系统保存或者更新失败   1. 系统显示保存或者更新失败信息，并提醒实验室管理员重新输入仿真设备信息 2. 实验室管理员可以重新提交本次仿真设备信息，也可以选择结束该用例   C-3 没有找到满足查询条件的仿真设备   1. 系统显示没有满足条件的仿真设备 2. 实验室管理员可以重新设定查询条件，也可以选择结束该用例 | |
| **补充约束-数据需求**  D-1 仿真设备信息包括设备编码、缩略图、设备名称、类别、软件系统、版本号、设备供应商、用途、所在实验室、设备状态  D-2 查询条件包括：设备名称、设备编号  **补充约束-业务规则**  B-1 设备编码必须以字母开头，长度范围为6-18个字符，设备编码不能为空，不能重复  **补充约束-非功能需求**  界面布局合理、美观、易用。 | |
| **待解决问题**  （暂无） | |

### 系统顺序图

依据前面分析得到的更新仿真设备用例story分析推导出更新仿真设备用例的系统顺序图，主要描述了用户与系统之间的交互逻辑。

1. 实验室管理员登录平台,获取相应的权限;系统验证用户并返回用户权限信息。
2. 实验室管理员点击仿真设备菜单，调用初始化仿真界面方法;系统加载并返回仿真信息界面。
3. 点击添加按钮，系统初始化新增仿真设备界面，输入仿真设备数据项(设备编码、缩略图、设备名称、类别、软件系统、版本号、设备供应商、用途、所在实验室、设备状态等);系统验证输入数据项的合法性和有效性。
4. 实验室管理员点击提交按钮，向系统提交仿真设备数据;系统新增仿真设备成功并返回处理状态。

如下：



## 分析模型

依据前章节中的更新仿真设备需求分析内容，我们本节开展更新仿真设备功能的概要设计，概要设计我们只关注于业务，而与更新仿真设备业务实际的实现技术框架毫无关系。概要设计主要使用鲁邦分析法，从两个维度入手，一是：鲁邦分析【也就是通俗的MVC】、二是：时序图。

### 鲁棒分析

鲁棒分析BCE：边界类(Boundary)、控制类(Controller)、实体类(Entity)。边界——起与外界交互的作用，它只能与控制对象和执行者有关系。

控制——对业务控制、流程控制的作用，它能与边界对象和实体对象有关系。

实体——业务元素的存储对象，与领域模型中的对象有良好的关系。它只能与控制对象有关系。

在对虚拟仿真实训教学管理及资源共享云平台项目中的更新仿真设备业务按照鲁棒分析方法，我们找出仿真设备操作三种类：边界类(Boundary)、控制类(Controller)、实体类(Entity)。其中：边界类包括仿真设备界面、仿真设备提示界面；控制类包括仿真设备界面控制类和仿真设备业务逻辑控制类。

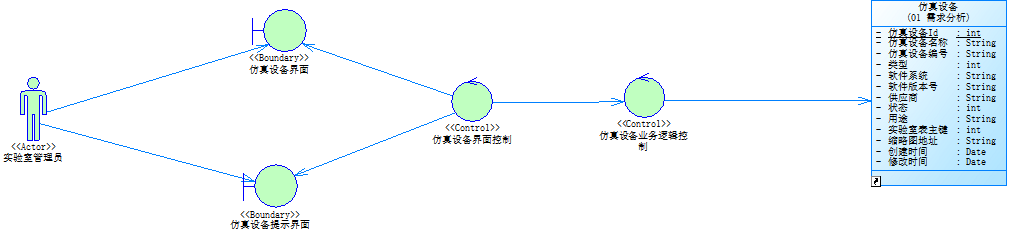
仿真设备界面：主要负责提供更新仿真设备维护时用户输入数据项和操作按钮。

仿真设备提示界面：主要负责提供对更新仿真设备表单项的提示。比如：“设备名称不能为空，请重新输入!”。

仿真设备控制类：主要负责接收更新仿真设备界面提交的仿真设备表单数据和向仿真设备提示界面返回提示数据。并承担调用仿真设备业务逻辑控制类中的业务方法。

仿真设备业务逻辑控制类：主要负责实现向仿真设备实体请求，进行仿真设备数据校验，并返回校验结果给仿真设备控制类。

更新仿真设备的鲁棒分析如下：

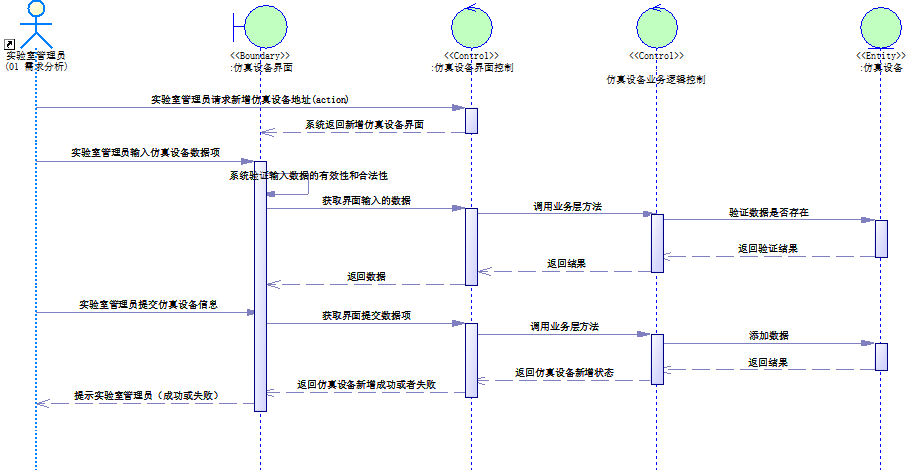


### 时序图

在仿真设备分析模型鲁棒分析流程中，我们可以将仿真设备与平台技术无关的请求校验流程以时序图的方式进行描述，这样更清晰的描述出更新仿真设备业务流程。在更新仿真设备时序图中，我们从使用者实验室管理员角度出发，进行仿真设备时序图描述：

1. 实验室管理员请求新增仿真设备地址(action)。
2. 仿真设备界面控制类拦截到实验室管理员的请求新增仿真设备地址，仿真设备界面控制类返回实验室管理员请求的仿真设备界面实体边界类，系统返回新增仿真设备界面。
3. 实验室管理员在仿真设备界面中输入仿真设备各数据项，系统验证输入数据的有效性和合法性。
4. 仿真设备界面控制类获取实验室管理员的输入数据，并调用仿真设备业务逻辑控制类的方法。
5. 仿真设备业务逻辑控制类调用仿真设备实体进行数据是否存在的验证。
6. 仿真设备实体进行设备名称或设备编号等验证，并返回给仿真设备业务逻辑控制类调用的相应方法。
7. 仿真设备业务逻辑控制类方法返回验证结果。
8. 仿真设备界面控制类获取验证（设备名称或设备编号等）结果（通过或者不通过），并将结果返回给仿真设备界面。
9. 仿真设备界面将返回的验证结果显示给用户。
10. 实验室管理员提交仿真设备所有必填信息项，仿真设备界面将实验室管理填写的仿真设备数据项进行提交，调用仿真设备界面控制类。
11. 仿真设备界面控制获取实验室管理员提交的数据，调用业务层添加仿真设备数据方法。
12. 仿真设备业务逻辑控制类调用仿真设备信息。
13. 仿真设备信息进行添加数据，并返回结果给仿真设备业务逻辑控制。
14. 仿真设备业务逻辑控制将返回的结果返回给仿真设备界面控制。
15. 仿真设备界面控制将结果返回给仿真设备界面向实验室管理员进行展示。

如下图：

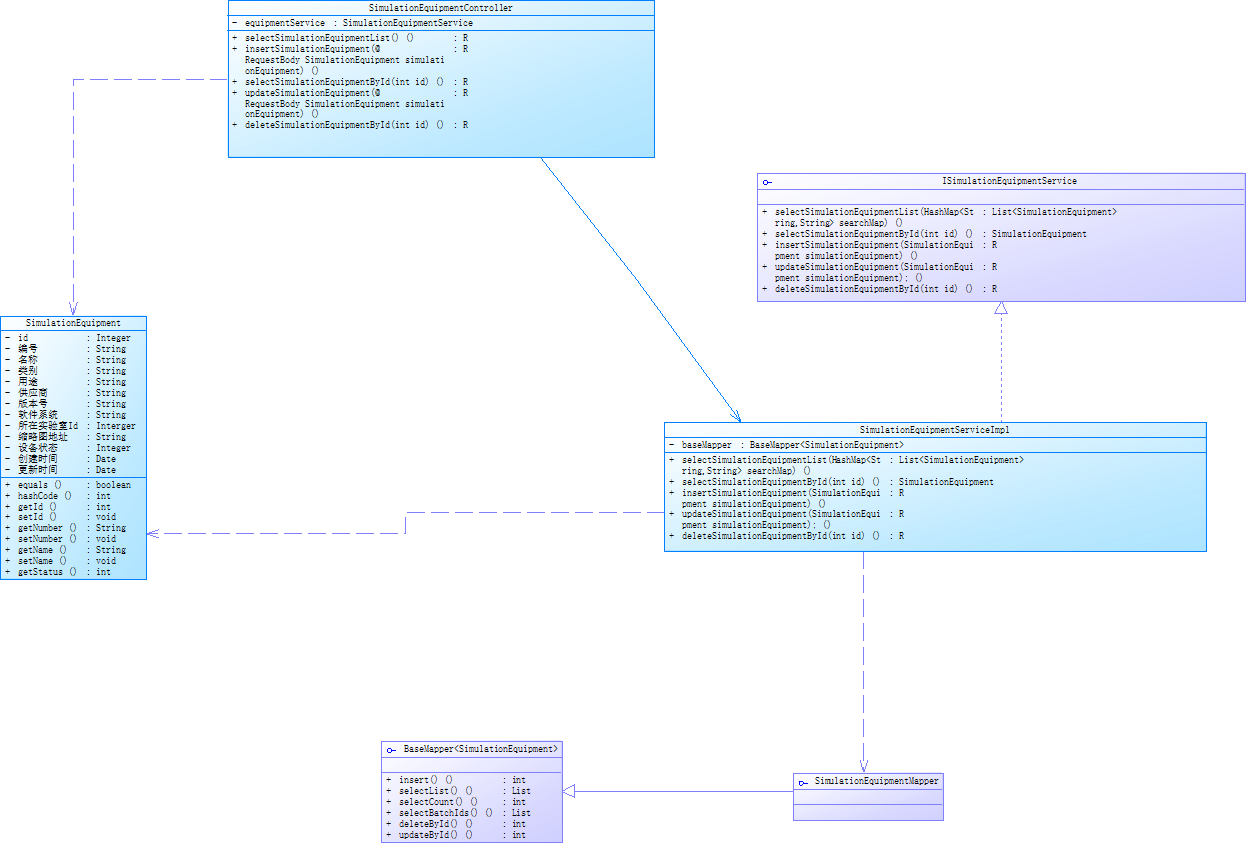


## 设计模型

在详细设计中，我们就需要把仿真设备流程与具体的开发技术和框架平台进行强关联，所以我需要对仿真设备业务设计出对应的实现类以及类中的属性和方法。使用结合用前后端分离的模式，仿真设备对应操作实现后端业务接口的调用。在设计模型中我们采用类图和时序图进行设计。

### 仿真设备类图

在仿真设备的类图中，我们本次涉及到控制层(SimulationEquipmentController)、实体层(SimulationEquipment)、业务逻辑层(SimulationEquipmentServiceImpl)、数据访问层(SimulationEquipmentMapper)等，具体类图如下：



1. 控制层(SimulationEquipmentController)：在SimulationEquipmentController中，我们设计出了调用仿真设备业务的服务层实例对象（equipmentService），并设计出新增仿真设备的方法（insertSimulationEquipment）。
2. 实体层(SimulationEquipment)：在对仿真设备数据数据进行封装时，我们设计出了仿真设备实体类SimulationEquipment，用于后期对相关数据的封装。
3. 业务逻辑层(SimulationEquipmentServiceImpl)：在业务逻辑层设计时，考虑到本业务需要处理更新仿真设备业务，所以设计了业务逻辑处理层仿真设备业务逻辑层类SimulationEquipmentServiceImpl，用于处理仿真设备新增、修改、查询、删除、导出等业务。
4. 数据访问层(SimulationEquipmentMapper)：在数据访问层设计时，我们设计出基于Myabatis框架的仿真设备映射Mapper。对相应数据进行操作，实现更新仿真设备业务的数据处理。

通过上面类图中的设计，我们整理出类中的方法和属性，如下表：

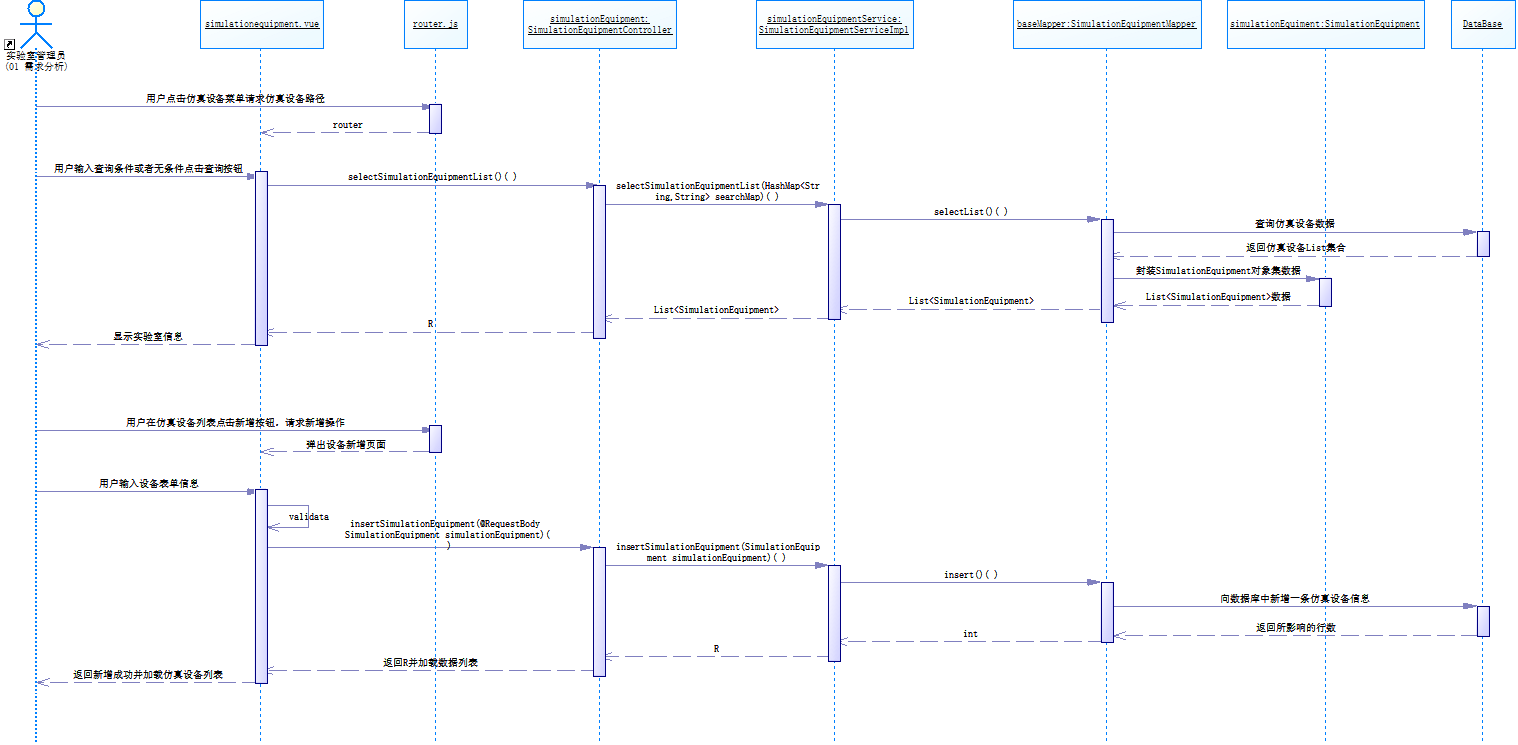
|  |  |  |
| --- | --- | --- |
| **类名** | **方法** | **属性** |
| 实体类(SimulationEquipment) | get\*\*();set\*\*(参数) | id、number、name、type、purpose、supplier、version、softwareSystem、laboratoryId、thumbnail、status、createTime、updateTime |
| 控制类(SimulationEquipmentController) | selectSimulationEquipmentList()  insertSimulationEquipment(@RequestBody SimulationEquipment simulationEquipment)  selectSimulationEquipmentById(int id)  deleteSimulationEquipmentById(int id)  updateSimulationEquipment(@RequestBody SimulationEquipment simulationEquipment) | equipmentService对象属性 |
| 业务逻辑类(SimulationEquipmentServiceImpl) | selectSimulationEquipmentList(HashMap<String,String> searchMap)  selectSimulationEquipmentById(int id)  insertSimulationEquipment(SimulationEquipment simulationEquipment)  updateSimulationEquipment(SimulationEquipment simulationEquipment);  deleteSimulationEquipmentById(int id) | baseMapper对象属性 |
| 数据访问类BaseMapper<SimulationEquipment> | insert()  deleteById()  updateById()  selectList() | Mybatisplus内部提供的增、删、改、查的方法 |

### 仿真设备时序图

仿真设备时序图，本次结合前后端框架,前端Vue,后端Springboot整合(SpringMvc、Mybatis/Mybatis-plus),数据库使用Mysql。然后按照分层的结构，面向接口(抽象)编程模式，与框架技术相关仿真设备时序图流程如下：

1. 实验室管理员点击仿真设备菜单请求仿真设备路径，主页面进行路由选择，并返回到仿真设备simulationEquipment.vue。
2. 实验室管理员在仿真设备界面中输入查询条件或者无条件点击查询按钮，仿真设备界面查询方法请求后端对应的仿真设备控制类查询方法(selectSimulationEquipmentList)。
3. 仿真设备控制类查询方法获取实验室管理员的输入数据项，并调用仿真设备业务处理类中的查询仿真设备方法。
4. 仿真设备业务逻辑类调用仿真设备数据层SimulationEquipmentMapper中的查询方法，SimulationEquipmentMapper中方法根据查询条件返回结果数据集。
5. 仿真设备数据层对返回的结果集进行数据处理和封装，准备成List<SimulationEquipment>集合对象数据并返回给仿真设备业务逻辑处理类。
6. 仿真设备业务逻辑类中的查询方法返回查询数据集给仿真设备控制类方法。
7. 仿真设备控制类将数据集进行重新组装，并带有数据状态一起返回给前端仿真设备界面。
8. 仿真设备simulationEquipment.vue进行仿真设备数据列表的渲染，并呈现给实验室管理员。
9. 实验室管理员在仿真设备界面表单中点击“新增”按钮，用户在仿真设备列表点击新增按钮，请求新增操作。系统弹出设备新增页面。
10. 在仿真设备新增页面中用户输入设备表单信息，页面进行验证(validata),这步主要实现前端数据项校验。如果前端验证通过，由通过Ajax调用后端的请示路径。
11. 后端对应的仿真设备控制层方法(insertSimulationEquipment)获取对应实验室管理员提交的仿真设备数据，并进行数据的封装，再调用仿真设备业务逻辑处理层新增方法（insertSimulationEquipment(SimulationEquipment simulationEquipment)）。
12. 仿真设备业务逻辑层调用数据处理层SimulationEquipmentMapper保存方法。
13. 数据处理层SimulationEquipmentMapper方法操作数据库Database，向数据库中添加一条仿真设备数据，并返回当前数据库成功的行数，并返回给仿真设备业务逻辑处理层的方法。
14. 仿真设备业务逻辑处理层方法返回行数给仿真设备控制层方法。
15. 仿真设备控制层方法封装返回前端仿真设备界面提示数据，并返回给前端页面。
16. 前端页面进行仿真设备新增成功或者失败提示。

完成仿真设备时序图如下：



### 数据库设计

从前面**仿真设备的领域模型**得知【得出大概需要考虑到的数据项有（设备编码、缩略图、设备名称、类别、软件系统、版本号、设备供应商、用途、所在实验室、设备状态）】和**仿真设备用例文本story**分析得到数据项【设备编码、缩略图、设备名称、类别、软件系统、版本号、设备供应商、用途、所在实验室、设备状态】，从而我们可以分析出仿真设备的E-R图。

在分析仿真设备用例的E-R图中，我们首页先找到仿真设备业务的实体(Entity),然后再去找与仿真设备业务相关的联系（Relationship），在当前仿真设备的业务中，显然我们找到实体，而无任务的联系。所以在仿真设备用例的E-R设计中，我们可以把仿真设备更新抽象成一个实体“仿真设备”，把从业务建模和用例建模中分析得出的数据项作为仿真设备实体的属性，从而仿真设备的属性有：仿真设备唯一标识符（主键）、设备名称、设备编码、类型、软件系统、软件版本号、设备供应商、用途、所在实验室、状态、缩略图、创建时间、更新时间。从而得到如下的E-R图：



通过上面E-R图我们可以整理出仿真设备的E-R表结构：

|  |  |  |  |
| --- | --- | --- | --- |
| 序号 | 字段描述 | 字段名称 | 字段类型 |
| 2 | 实验室表主键 | laboratory\_id | int |
| 3 | 仿真设备名称 | name | varchar(64) |
| 4 | 仿真设备编号 | number | varchar(64) |
| 5 | 用途 | purpose | varchar(64) |
| 6 | 软件系统 | software\_system | varchar(64) |
| 7 | 状态 | status | int |
| 8 | 供应商 | supplier | varchar(50) |
| 9 | 缩略图地址 | thumbnail | varchar(255) |
| 10 | 类型 | type | varchar(64) |
| 11 | 创建时间 | create\_time | datetime |
| 12 | 最后修改时间 | update\_time | datetime |

## 实现模型

在更新仿真设备实现模型中，我们采用前后端分离模式，前端框架使用Vue、Element-ui;后端框架使用Springboot整合（SpringMvc、Mybatis/Mybatis-plus）。按照分层思想，我们把后端分层Controller、Service、Mapper、Model(Entity)、Common。

前后端交互我们可以通过nginx或者直接访问tomcat两种方式。我们将技术框架与案例解析进行分层对应，对应如下：

1、技术框架Vue对应案例中的仿真设备界面表单。

2、技术框架SpringBoot（Controller、Service、Mapper），其中Controller层包含案例中SimulationEquipmentController类，Service层包含案例中SimulationEquipmentServiceImpl类，Mapper层包括SimulationEquipmentMapper类。

### 技术框架与案例解析

我们将技术框架与案例解析进行分层对应，来给同学们梳理技术架构与实际案例项目的逻辑对应关系，方便同学们后期在技术选型和业务功能开发中思路清晰、有条有理。能更好的为同学们独立开发打下坚实的基础。对应如下：



我们通过技术架构得出，前端我们选择使用Vue框架来实现，后端我们选择使用Springboot+SSM框架来实现，前端与后端的交互，我们可以使用直接调用tomcat或者使用第三方中间件(nginx)来作代理。

**Tomcat和Nginx是两种不同的服务器软件，各自具有不同的功能和特点。**

* **Tomcat:**

Tomcat是一个开源的、基于Java的Web应用服务器，主要用于托管Java Web应用程序。

它能够解释和执行Java Servlet、JavaServer Pages（JSP）和相关的Java技术。

Tomcat通常被用作应用服务器，处理动态内容和Java应用程序的部署。

它提供了处理Java代码、数据库连接池、会话管理等功能。

* **Nginx:**

Nginx是一个轻量级的、高性能的开源Web服务器和反向代理服务器。

它通常用于静态文件服务、负载均衡、反向代理和缓存等方面。

Nginx非常擅长处理并发连接，适合作为高性能的前端代理服务器。

总的来说，Tomcat更专注于Java应用程序的托管和执行，而Nginx更适合作为高性能的Web服务器和反向代理服务器，可以用来处理静态内容、负载均衡和缓存等功能。在一些场景中，两者也可以配合使用，比如Nginx作为静态内容的前端服务器，而Tomcat处理动态内容和Java应用程序。

**Tomcat和Nginx在实际应用中有不同的使用场景，主要取决于它们的特点和功能：**

* **Tomcat的应用场景：**

托管Java Web应用程序：Tomcat是一个专门用于托管Java Web应用程序的服务器，可以执行Java Servlet、JavaServer Pages（JSP）等技术。

动态内容处理：适合处理动态生成的内容，如数据库查询、业务逻辑处理等。

Java应用部署：用于部署和运行基于Java的Web应用，包括企业级应用、网站和Web服务等。

* **Nginx的应用场景：**

静态文件服务：Nginx非常擅长提供静态文件的服务，如图片、CSS、JavaScript等，可以快速地响应这些静态内容的请求。

反向代理和负载均衡：Nginx可以作为反向代理服务器，接收客户端请求并将其转发给后端的多个服务器，同时可以实现负载均衡，分发请求到多个后端服务器上。

缓存代理：Nginx可以缓存经常访问的内容，加速响应速度并减轻后端服务器的压力。

在实际应用中，通常会根据具体需求将Tomcat和Nginx结合使用。例如，Nginx可以作为前端服务器接收并处理静态内容，并将动态内容请求转发给Tomcat进行处理，从而充分发挥两者的优势，提高整体性能和稳定性。

前端框架：

|  |  |
| --- | --- |
| **序号** | **名称** |
| 1 | Html+css+javascript+ajax |
| 2 | bootstrap |
| 3 | layui |
| **4** | **Vue** |
| 5 | React |
| 6 | Angular |
| 7 | ..... |

后端技术框架：

|  |  |
| --- | --- |
| **序号** | **名称** |
| 1 | Servlet |
| **2** | **SpringMVC、Spring、Mybatis** |
| 3 | SpringMVC、Spring、Hibernate |
| **4** | **Springboot** |
| 5 | Springcloud |
| 6 | ...... |

前后端交互我们可以通过第三方中间件nginx或者直接访问tomcat两种方式。

**方式一（直接调用tomcat服务接口）：**

前端调用后端接口：

|  |
| --- |
| url = “http://localhost:8088/simulationEquipment/selectSimulationEquipmentList”  Param={}  this.request.post(url,param).then(res=>{  if (res.code == 200) {  //获取后端接口返回数据  res.data.data;  }  }) |

后端提供服务接口：

http://localhost:8088//simulationEquipment/selectSimulationEquipmentList

**方式二（使用第三方中间件nginx）：**

前端调用后端接口：

|  |
| --- |
| url = “www.cqupt.hwadee.com//simulationEquipment/selectSimulationEquipmentList”  Param={}  this.request.post(url,param).then(res=>{  if (res.code == 200) {  //获取后端接口返回数据  res.data.data;  }  }) |

nginx代理服务：

|  |
| --- |
| user www-data;  worker\_processes auto;  pid /run/nginx.pid;  include /etc/nginx/modules-enabled/\*.conf;  events {  worker\_connections 768;  }  http {  sendfile on;  tcp\_nopush on;  tcp\_nodelay on;  keepalive\_timeout 65;  types\_hash\_max\_size 2048;  include /etc/nginx/mime.types;  default\_type application/octet-stream;  error\_log /var/log/nginx/error.log;  gzip on;  upstream mycqlcbserver {  server 172.0.0.1:8080 weight=20;  ip\_hash;  }  fastcgi\_intercept\_errors on;  server {  listen 80;  server\_name www.cqupt.hwadee.com;  access\_log /etc/nginx/logs/access.log;  large\_client\_header\_buffers 4 128k;  #zx add  error\_page 403 404 500 502 503 504 /404.html;  location = /404.html {  root /etc/nginx/html/etc/nginx/html;  }  location / {  proxy\_pass http://mycqlcbserver;  proxy\_redirect off;  proxy\_set\_header Host $host;  proxy\_set\_header X-Real-IP $remote\_addr;  proxy\_set\_header X-Forwarded-For $proxy\_add\_x\_forwarded\_for;  client\_max\_body\_size 300m;  client\_body\_buffer\_size 128k;  proxy\_connect\_timeout 9000;  proxy\_send\_timeout 9000;  proxy\_read\_timeout 9000;  proxy\_buffer\_size 4k;  proxy\_buffers 4 32k;  proxy\_busy\_buffers\_size 64k;  proxy\_temp\_file\_write\_size 64k;  proxy\_intercept\_errors on;  #zx add  error\_page 403 404 500 502 503 504 /404.html;  location = /404.html {  root /etc/nginx/html;  }  }  }  } |

后端提供服务接口：

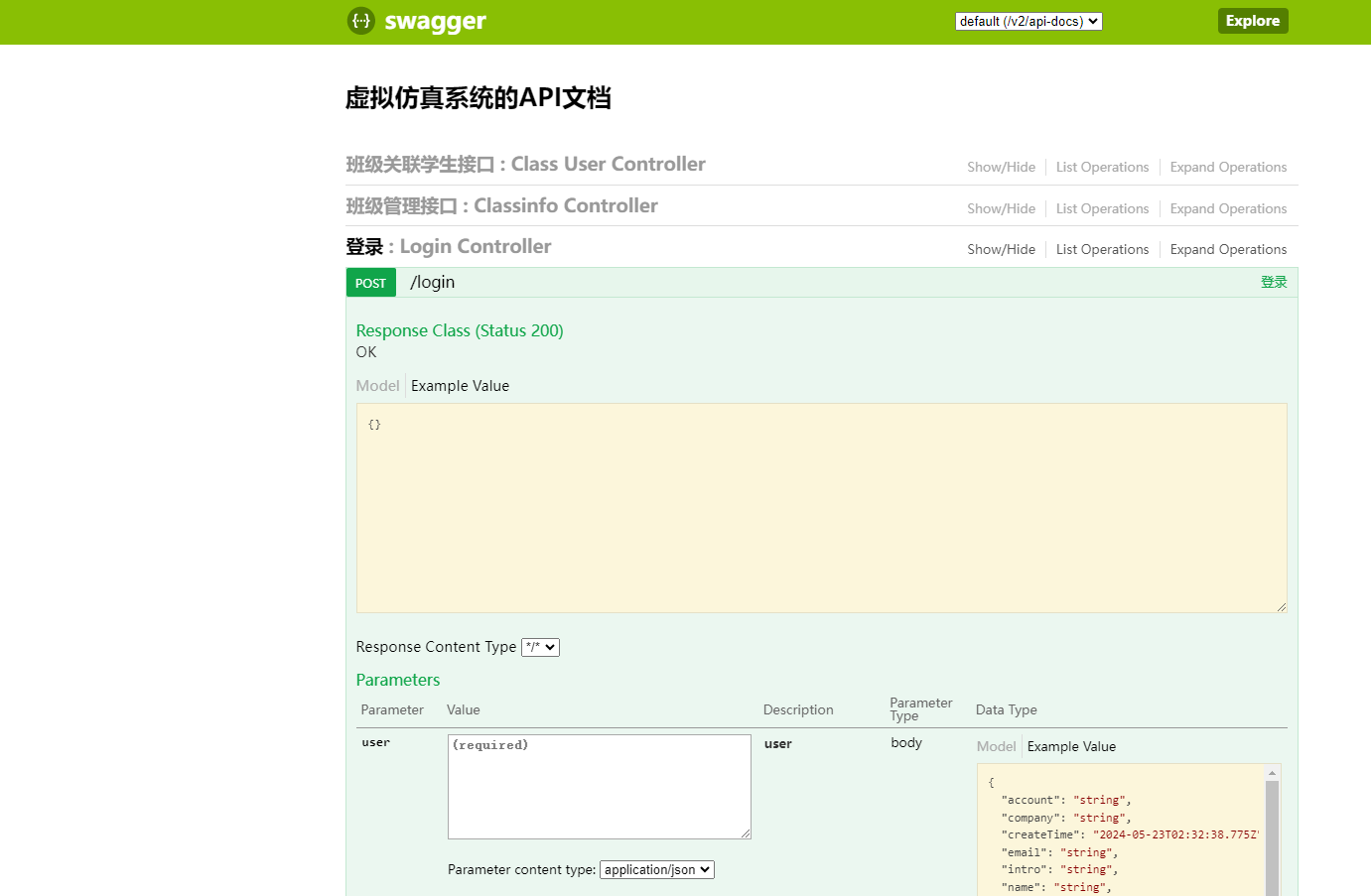
http://localhost:8088//simulationEquipment/selectSimulationEquipmentList

### 前后端独立开发接口调用或者调试方式

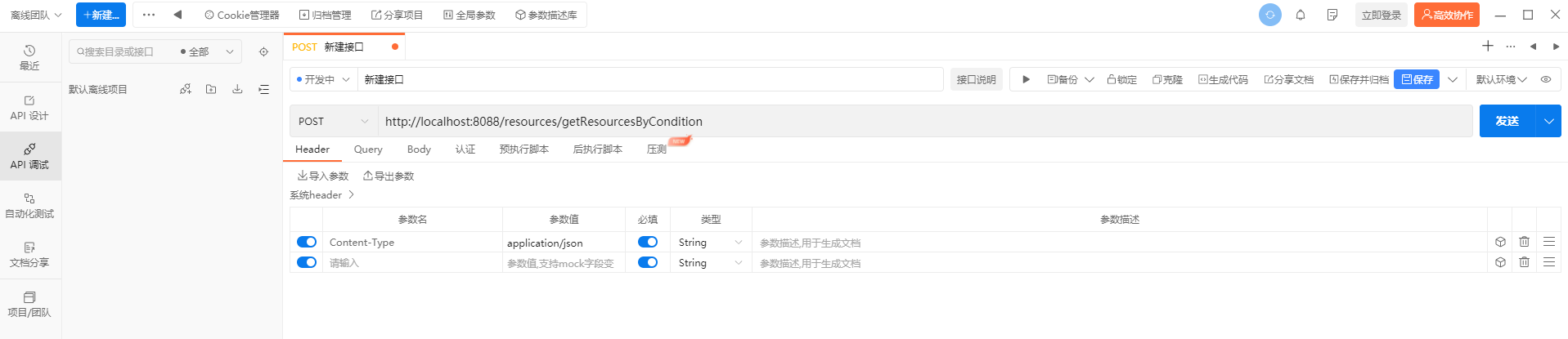
在实际的项目开发中，我们的开发人员可能只专注于前端或者后端开发，那这种情况下，我们前端或者后端开发人员可以使用接口工具进行对需要调用的业务接口进行调试，接口工具有：ApiPost7、Swagger等。

1. Swagger：

访问地址：http://localhost:8088/swagger-ui.html



2、ApiPost7：



### 本次教学案例技术选型

**从上面的技术中，我们本次的案例选型如下**：

编码使用到的技术栈：

后端：Springboot、SpringMVC、Mybatis/MybatisPlus。

前端：VUE、Element-UI

数据库：Mysql

### 界面设计大体思路

界面大体的构思：仿真设备包含：设备编码、缩略图、设备名称、类别、软件系统、版本号、设备供应商、用途、所在实验室、设备状态等信息，提交、返回按钮构成（这数据项信息来源于用例描述中，请追溯到2.2）。界面设计：同学们可以使用原型工具：axure或者HBuilder。仿真设备原型设计如下：



### 前端

按照MVC的思想，V就是视图的意思，本次视图我们使用前端框架Vue来实现。

MVC（Model-View-Controller）是一种软件架构模式，用于组织和管理应用程序中的代码，以实现模块化、可维护和可扩展的设计。MVC模式将应用程序分为三个核心部分：模型（Model）、视图（View）和控制器（Controller）。每个部分都有不同的责任和功能，有助于降低代码耦合度，并且使得代码更易于维护和扩展。

模型（Model）：模型代表应用程序中处理数据和业务逻辑的部分。它负责管理应用程序的状态、数据和行为。模型通常包括数据访问对象、实体对象、业务逻辑和数据持久化逻辑。模型与视图和控制器无关，它们独立于用户界面或输入机制，可以被多个视图和控制器共享。

视图（View）：视图负责应用程序的用户界面部分，向用户展示数据并接受用户输入。视图通常是用户能够看到和与之交互的部分，包括UI元素、页面布局等。视图通常会从模型中获取数据，在用户界面上呈现这些数据，并将用户的操作反馈给控制器。

控制器（Controller）：控制器充当模型和视图之间的中介，负责处理用户的输入、调用模型来处理数据和业务逻辑，并更新视图来显示最新的数据。它接收来自用户的输入，然后根据输入选择合适的模型进行处理，并最终更新视图。控制器将用户的行为转换为对模型的操作，反之亦然。它负责应用程序的流程控制和业务逻辑的协调。

MVC模式的优点包括提高了代码的可重用性、可维护性和可测试性，同时也降低了各部分之间的耦合度，使得应用程序的各个部分更易于独立开发和维护。该模式已成为许多现代软件开发框架和平台的核心设计思想。

框架：在计算机科学和软件工程中，框架（Framework）是一种提供了特定功能的、可重用的软件环境或基础结构。框架通常包括一系列预先设计好的类、接口、函数和其他组件，以及定义了它们之间关系和交互方式的规范。开发人员可以在框架的基础上构建应用程序，并使用框架提供的工具和功能，从而加快开发速度、降低开发成本，并且提高软件质量。

使用框架通常涉及以下几个步骤：

安装和设置：首先，你需要安装框架及其相关的工具和依赖。这通常涉及使用包管理工具（如npm等）来下载框架的代码库，并且可能需要进行一些配置和设置。

学习框架的基本概念：在开始实际开发之前，建议你先学习框架的基本概念和核心特性。这可能包括了解框架的组件化方式、数据管理机制、路由系统等。

创建项目：一旦你熟悉了框架的基本概念，接下来可以创建一个新的项目或者应用程序。框架通常提供了一些命令行工具或脚手架来辅助创建项目的初始结构。

编写代码：使用框架提供的API和工具，你可以开始编写应用程序的代码。这可能涉及创建组件、定义路由、管理状态等。

调试和测试：在编写代码的过程中，你可能需要进行调试和测试。框架通常会提供一些工具和技术来帮助你进行单元测试、集成测试以及调试。

部署：最后，当应用程序开发完成后，你需要考虑如何将其部署到生产环境中。框架通常也提供了一些推荐的部署方式和最佳实践。

总的来说，使用框架需要先了解其基本概念，然后按照框架的规范和约定进行项目创建和代码编写。随着你对框架的熟悉度增加，你将能够更高效地利用框架提供的功能来构建优秀的Web应用程序。

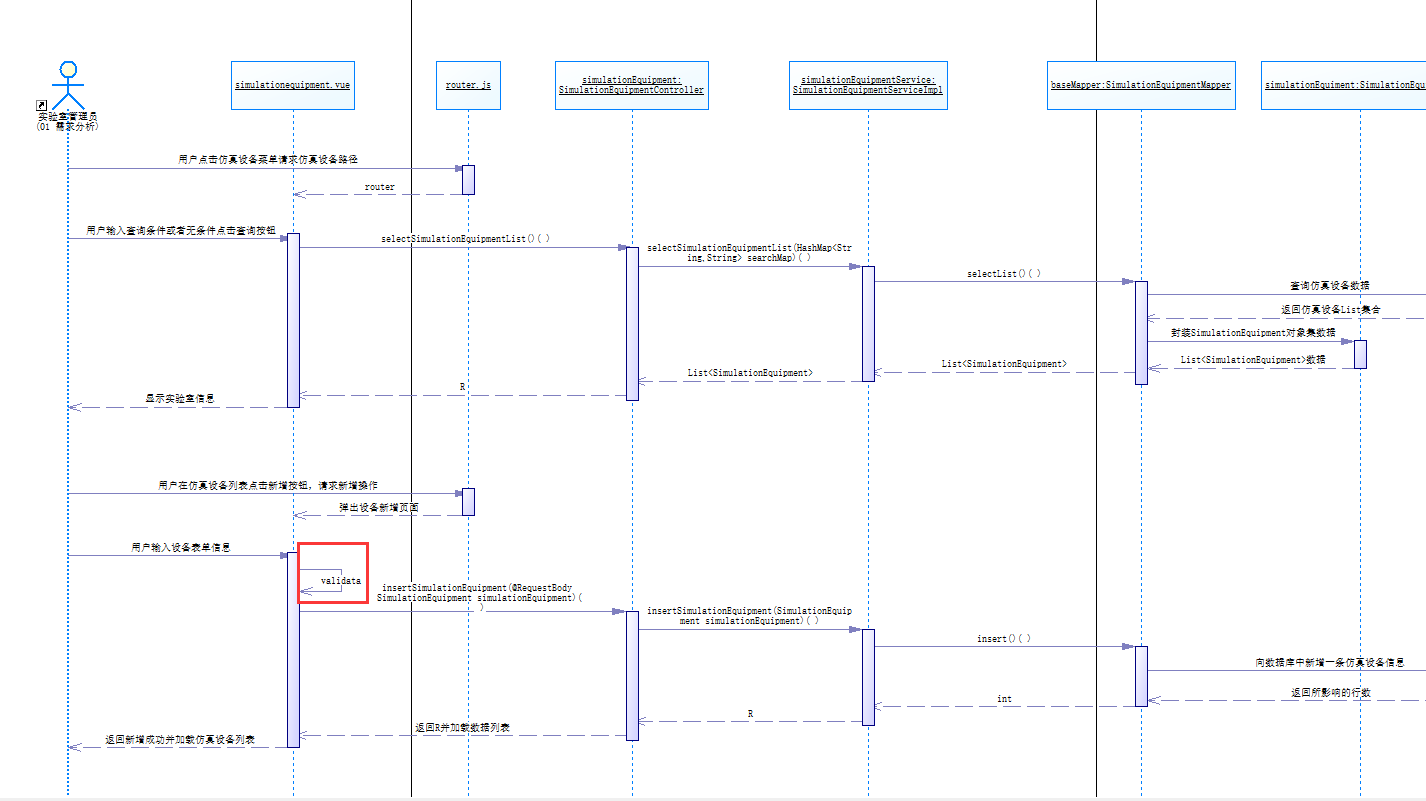
在仿真设备用例中我们的MVC实现是这样来规划的：

M：就是仿真设备信息实体。

V：就是仿真设备前端的页面.

C：就是接收用户提交仿真设备数据和反回仿真设备提示的后端类。

在仿真设备用例的前端实现视图页面我们设计并命名simulationequipment.vue。我们通过设计模型中的时序图来进行前端实现编码，如下：



在仿真设备时序图中，前端页面simulationequipment.vue中有验证用户输入仿真设备数据(validate/rules)和仿真设备提交(submitEditForm)操作。对应如下前端代码中红色部分：

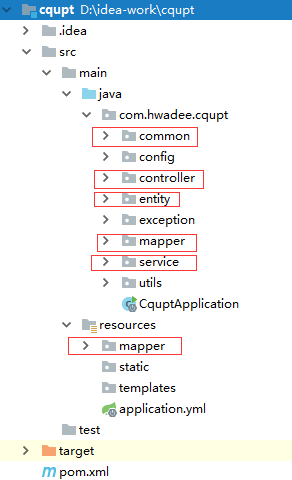
|  |
| --- |
| <template>  <div>  <!-- 查询条件 -->  <el-form :inline="true" :model="searchForm">  <el-form-item label="名称">  <el-input v-model="searchForm.name" placeholder="名称"></el-input>  </el-form-item>  <el-form-item label="编号">  <el-input v-model="searchForm.number" placeholder="编号"></el-input>  </el-form-item>  <el-form-item>  <el-button type="primary" @click="search" icon="el-icon-search">查询</el-button>  </el-form-item>  <el-form-item>  <el-button type="success" @click="add()" icon="el-icon-edit">添加</el-button>  </el-form-item>  </el-form>  <!-- 表格 -->  <el-table :data="tableData" style="width: 100%">  <el-table-column  v-for="(item,index) in headArr"  :label="item.lable"  :key="index"  :prop="item.prop">  <template slot-scope="scope">  <span v-if="item.prop != 'operate'">{{scope.row[item.prop]}}</span>  <div v-if="item.prop == 'operate'" width="100px">  <el-tooltip class="item" effect="dark" content="编辑" placement="top">  <el-button type="primary" icon="el-icon-edit" circle  @click="updateSimulationEquipment(scope.row)">  </el-button>  </el-tooltip>  <el-tooltip class="item" effect="dark" content="删除" placement="top">  <el-button type="danger" icon="el-icon-delete" circle  @click="deleteSimulationEquipmentById(scope.row)">  </el-button>  </el-tooltip>  </div>  </template>  </el-table-column>  </el-table>  <!-- 弹出框 -->  <el-dialog :title="textMap[dialogStatus]" :visible.sync="dialogFormVisible">  <el-form ref="editform" :rules="rules" :model="editForm">  <el-form-item label="id" prop="id" v-if="false">  <el-input v-model="editForm.id"></el-input>  </el-form-item>  <el-form-item label="名称" prop="name" :label-width="formLabelWidth">  <el-input placeholder="名称" v-model="editForm.name"></el-input>  </el-form-item>  <el-form-item label="编号" prop="number" :label-width="formLabelWidth">  <el-input placeholder="编号" v-model="editForm.number"></el-input>  </el-form-item>  <el-form-item label="类别" prop="type" :label-width="formLabelWidth">  <el-input placeholder="类别" v-model="editForm.type"></el-input>  </el-form-item>  <el-form-item label="用途" prop="purpose" :label-width="formLabelWidth">  <el-input placeholder="用途" v-model="editForm.purpose"></el-input>  </el-form-item>  <el-form-item label="供应商" prop="supplier" :label-width="formLabelWidth">  <el-input placeholder="供应商" v-model="editForm.supplier"></el-input>  <el-select v-model="editForm.supplier" placeholder="请选择">  <el-option  v-for="item in supplierOptions"  :key="item.value"  :label="item.label"  :value="item.value">  </el-option>  </el-select>  </el-form-item>  </el-form>  <div slot="footer" class="dialog-footer">  <el-button @click="dialogFormVisible = false">取 消</el-button>  <el-button type="primary" v-if="dialogStatus=='add'" @click="submitEditForm('editform')">提交</el-button>  <el-button type="warning" v-else @click="updateEditForm('editform')">修 改</el-button>  </div>  </el-dialog>  </div>    </template>  <script>  import {  simulationEquipmentUrl  } from "../../api/api.js";  export default {  data() {  return {  headArr:[  {lable:'ID',prop:'id'},  {lable:'名称',prop:'name'},  {lable:'编号',prop:'number'},  {lable:'类别',prop:'type'},  {lable:'用途',prop:'purpose'},  {lable:'供应商',prop:'supplier'},  {lable:'软件系统',prop:'softwareSystem'},  // {lable:'缩略图地址',prop:'thumbnail'},  {lable:'设备状态',prop:'status'},  // {lable:'更新时间',prop:'updateTime'},  // {lable:'创建时间',prop:'createTime'},  {lable:'操作',prop:'operate'},  ],  tableData: [],  //supplierOptions:{},  searchForm: {  name: '',  number: '',  },  dialogFormVisible: false,  dialogStatus: 'add',  textMap: {  add: '新增',  update: '编辑'  },  editForm: {  id: '',  name: '',  number: '',  type: '',  purpose: '',  supplier: '',  softwareSystem: '',  thumbnail: '',  status: ''  },  rules: {  name: [{  required: true,  message: '请输入名称',  trigger: 'blur'  }],  number: [{  required: true,  message: '请输入编号',  trigger: 'blur'  }]  }  }  },  created() {  this.getData();  },  methods: {  getData(){  this.request.get(simulationEquipmentUrl.qureySimulationEquipmentList).then(res => {  if (res.code == 200) {  this.tableData = res.data.list;  } else {  alert(res.msg);  }  })  },  search(){  let params = {  name: this.form.name,  number: this.form.number,  }  this.getData(params);  },  // 新增  add(){  this.dialogFormVisible = true;  this.dialogStatus = 'add';  },  submitEditForm(formname){  this.$refs[formname].validate((valid) => {  if (valid) {  let params = this.editForm;  this.request.post(simulationEquipmentUrl.insertSimulationEquipment,params).then((res) => {  if (res.code == 200) {  console.log('成功');  this.dialogFormVisible = false;  this.getData();  } else {  this.$message.error('添加数据失败！');  }  })  } else {  console.log('error submit!!');  return false;  }  });  },    // 编辑  updateSimulationEquipment(row){  console.log('update id:'+row.id);  this.dialogFormVisible = true;  this.dialogStatus = 'update';  //查询数据  let param = {id:row.id};  this.request.get(simulationEquipmentUrl.selectSimulationEquipmentById,{params:param}).then(res => {  if (res.code == 200) {  this.editForm = res.data.data;  } else {  alert(res.msg);  }  })  },  updateEditForm(formname){  this.$refs[formname].validate((valid) => {  if (valid) {  let params = this.editForm;  console.log(params);  this.request.post(simulationEquipmentUrl.updateSimulationEquipment,params).then((res) => {  if (res.code == 200) {  this.dialogFormVisible = false;  this.getData();  this.$message.success('编辑数据成功！');  } else {  this.$message.error('编辑数据失败！');  }  })  } else {  console.log('error submit!!');  return false;  }  });  },  // 删除  deleteSimulationEquipmentById(row){  this.$confirm('确定要删除【' + row.name + '】吗？', '提示', {  confirmButtonText: '确定',  cancelButtonText: '取消',  type: 'warning'  }).then(() => {  console.log('delete id:'+row.id);  let param = {id:row.id};  this.request.get(simulationEquipmentUrl.deleteSimulationEquipmentById,{params:param}).then(res =>{  if (res.code == 200) {  this.getData();  } else {  alert(res.msg);  }  })  });      }    }  }  </script>  <style>  </style> |

### 后端

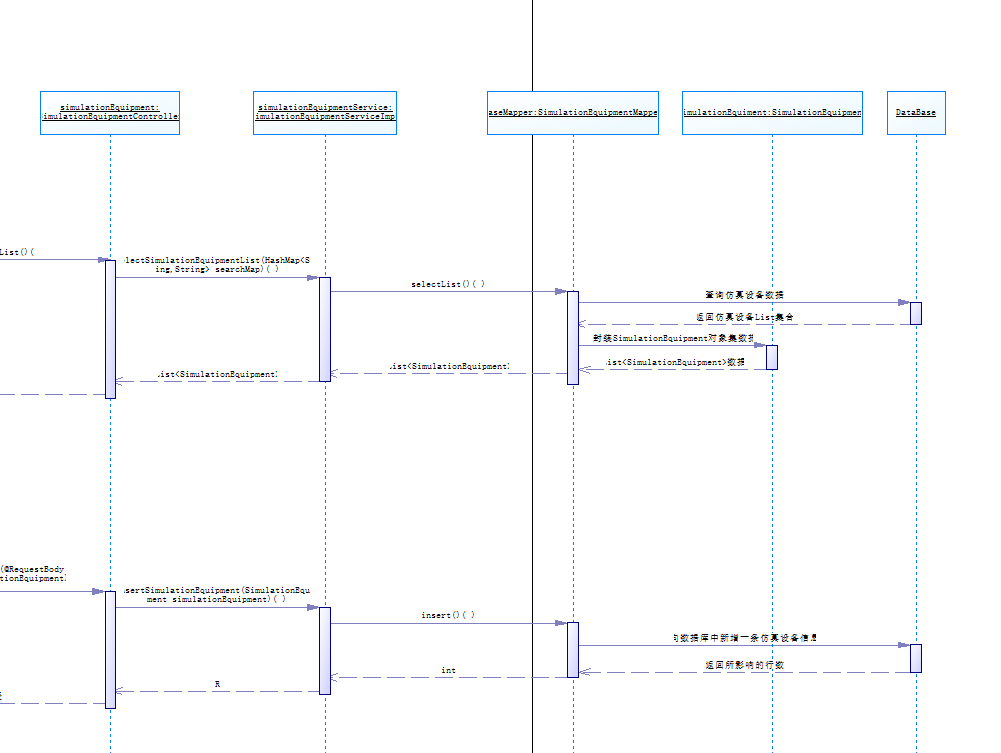
关于仿真设备用例的后端实现逻辑，我们按照设计模型中的时序图来进行开展编码实现。其中我们按照软件开发分层思想，我们把后端分成控制层、业务逻辑处理层、持久层（数据层）、实体层、公共通用层。

|  |  |  |  |
| --- | --- | --- | --- |
| **序号** | **中文名** | **英文名** | **作用** |
| **1** | 控制层 | **Controller** | 负责请求转发，接受页面过来的参数，传给Service处理，接到返回值，再传给页面。负责具体的业务模块流程的控制，controller层主要调用Service层里面的接口控制具体的业务流程，控制的配置也需要在配置文件中进行。 |
| **2** | 业务逻辑层（服务层） | **Service** | Service层叫服务层，被称为服务，粗略的理解就是对一个或多个DAO进行的再次封装，封装成一个服务，所以这里就需要事物控制，service层主要负责业务模块的应用逻辑应用设计，先设计接口，再设计其实现的类，接着再Spring的配置文件中配置其实现的关联。这样我们就可以在应用中调用Service接口来进行业务处理。Service层的业务实现，具体要调用到已定义的DAO层的接口，封装Service层的业务逻辑有利于通用的业务逻辑的独立性和重复利用性，程序显得非常简洁。 |
| **3** | 持久层（数据层） | **Mapper或者Dao** | dao层属于一种比较底层，比较基础的操作，具体到对于某个表的增删改查，也就是说某个DAO一定是和数据库的某一张表一一对应的，其中封装了增删改查基本操作。  Dao层用于定义方法接口，然后在service层和serviceimpl层实现接口的方法，从而达到操作数据库的目的。对数据库进行数据持久化操作，方法语句主要是针对数据库的，主要实现一些增删改查操作，在Mybatis中的方法主要与xx.xml内相互一一映射。 |
| **4** | 实体层 | **Model(Entity)** | 是一个简单的Java对象，没有任何限制和框架，仅包含一些属性和访问这些属性的方法。 |
| **5** | 公共通用层 | **Common** | 公共方法封装、通用类定义等 |

按照上面的分层思想，本次教学案例后端项目结构如下：



注：详细设计类图与源代码的对应，请参考如下图片和源代码：



根据设计模型中的类图，我们可以整理出本次仿真设备用例实现的类和方法，如下表：

|  |  |  |  |
| --- | --- | --- | --- |
| **序号** | **类名** | **类描述** | **方法** |
| 1 | CquptApplication | 启动类：是整个Spirngboot项目的运行入口。 | main方法 |
| 2 | SimulationEquipmentController | 控制类：负责仿真设备业务控制，获取前端用户的仿真设备相关数据，并调用业务层的方法。 | 1、查询仿真设备方法  @GetMapping(value = "/selectSimulationEquipmentList")  R selectSimulationEquipmentList()  2、新增仿真设备方法  @PostMapping(value = "/insertSimulationEquipment")  R insertSimulationEquipment(@RequestBody SimulationEquipment simulationEquipment)  3、更新仿真设备方法  @PostMapping(value = "/updateSimulationEquipment")  R updateSimulationEquipment(@RequestBody SimulationEquipment simulationEquipment)  4、删除仿真设备方法  @GetMapping(value = "/deleteSimulationEquipmentById")  R deleteSimulationEquipmentById(int id)  5、定义根据Id获取详细信息方法  @GetMapping(value = "/selectSimulationEquipmentById")  public R selectSimulationEquipmentById(int id) |
| 3 | ISimulationEquipmentService | 业务逻辑接口 | 1、声明查询仿真设备方法  List<SimulationEquipment> selectSimulationEquipmentList(HashMap<String,String> searchMap); 2、声明新增仿真设备方法  R insertSimulationEquipment(SimulationEquipment simulationEquipment); 3、声明更新仿真设备方法  R updateSimulationEquipment(SimulationEquipment simulationEquipment); 4、声明删除仿真设备方法  R deleteSimulationEquipmentById(int id);  5、声明根据Id获取详细信息方法  SimulationEquipment selectSimulationEquipmentById(int id); |
| 4 | SimulationEquipmentServiceImpl | 业务逻辑实现类 | 1、查询仿真设备方法，并返回仿真设备数据集  List<SimulationEquipment> selectSimulationEquipmentList(HashMap<String,String> searchMap)  2、新增仿真设备方法，并返回成功的状态R对象  R insertSimulationEquipment(@RequestBody SimulationEquipment simulationEquipment)  3、更新仿真设备方法，并返回成功的状态R对象  R updateSimulationEquipment(@RequestBody SimulationEquipment simulationEquipment)  4、删除仿真设备方法，并返回影响行数的状态R对象  R deleteSimulationEquipmentById(int id)  5、定义根据Id获取详细信息方法，返回数据对象R  public R selectSimulationEquipmentById(int id) |
| 5 | SimulationEquipmentMapper | 仿真设备Mapper接口SimulationEquipmentMapper 继承是Mybatisplus基础BaseMapper，并将仿真设备对象作为参数传递对基础Mapper | //删除  int deleteById(Serializable id)  //更新  int updateById(@Param("et") T entity)  //查询  List<T> selectList(@Param("ew") Wrapper<T> queryWrapper)  //新增  int insert(T entity);  //根据Id获取详细信息  T selectById(Serializable id) |
| 6 | BaseMapper<T> | 这个类我们本次主要是使用是MybatisPlus自己封装的方法，因为MybatisPlus内部实现了对单个数据（实体）的增、删、改、查操作。 | MybatisPlus内部实现方法 |

#### 启动类

我们创建了一个名为CquptApplication的类，这是SpringBoot应用程序的主启动类。主要涉及以下几个重要点：

@SpringBootApplication 注解：这个注解标记了这是一个SpringBoot应用程序的入口类。它包含了多个其他注解的组合，用于自动配置Spring应用程序上下文、组件扫描等。

@MapperScan是MyBatis框架中的一个注解，它的作用是指定MyBatis Mapper接口的扫描路径，用于自动扫描并注册这些Mapper接口。在使用MyBatis进行数据库操作时，通常会编写Mapper接口来定义数据访问的方法。而使用@MapperScan注解可以告诉 Spring 容器去扫描特定的包路径，以找到并注册这些Mapper接口，从而使其能够被正确地注入到其他组件中并被调用。

main方法：这是Java 应用程序的入口方法。在这个方法中，我们通过 SpringApplication.run() 方法启动了SpringBoot应用程序。run()方法接受两个参数，第一个参数是主启动类的类对象，第二个参数是命令行参数。

|  |
| --- |
| package com.hwadee.cqupt; import org.mybatis.spring.annotation.MapperScan; import org.springframework.boot.SpringApplication; import org.springframework.boot.autoconfigure.SpringBootApplication; @SpringBootApplication @MapperScan("com.hwadee.cqupt.mapper") public class CquptApplication {  public static void main(String[] args) {  SpringApplication.run(CquptApplication.class, args);  } } |

#### 控制类

Controller控制类是Spring框架中用于处理用户请求、调用业务逻辑并返回响应的组件。在一个典型的MVC（Model-View-Controller）架构中，Controller扮演着连接视图和数据模型之间的角色。具体来说，Controller控制类的作用包括：

接收请求：Controller接收来自客户端（如浏览器、移动应用等）的请求，这些请求可以是HTTP请求中的GET、POST等方法或者其他协议的请求。

调用业务逻辑：一旦接收到请求，Controller会调用相应的业务逻辑组件（Service、Manager等）来执行实际的业务处理，例如查询数据库、处理数据、调用其他服务等。

处理请求参数：Controller可以从请求中获取参数，并将这些参数传递给业务逻辑组件以便进行处理。

生成响应：在完成业务逻辑处理后，Controller会根据处理结果生成合适的响应，通常是一个包含数据的视图或者一个JSON/XML格式的数据返回给客户端。

路由转发：根据请求的URL路径或其他条件，Controller可能会决定将请求转发给其他的控制器或处理器进行进一步的处理。

在Spring框架中，通常使用@Controller或@RestController注解来定义一个类为Controller控制类。这些注解告诉Spring框架将这些类识别为处理请求的组件，并进行相应的处理和管理。

在仿真设备业务中，我们创建了SimulationEquipmentController 控制类。

|  |
| --- |
| package com.hwadee.cqupt.controller; import com.baomidou.mybatisplus.core.toolkit.StringUtils; import com.hwadee.cqupt.common.R; import com.hwadee.cqupt.entity.SimulationEquipment; import com.hwadee.cqupt.service.impl.SimulationEquipmentServiceImpl; import io.swagger.annotations.Api; import io.swagger.annotations.ApiOperation; import org.springframework.web.bind.annotation.\*; import javax.annotation.Resource; import java.util.HashMap; import java.util.List; import java.util.Map;  /\*\*  \* 仿真设备接口  \* @author: bxl  \* @date: 2023/7/3  \*/ @RestController @Api(value = "仿真设备Controller", tags = { "仿真设备接口" }) @RequestMapping("simulationEquipment") public class SimulationEquipmentController {   @Resource  private SimulationEquipmentServiceImpl simulationEquipmentService;   @ApiOperation(value = "查询所有仿真设备")  @GetMapping(value = "/selectSimulationEquipmentList")  @ResponseBody  public R selectSimulationEquipmentList(){  HashMap<String,String> searchMap = new HashMap<>();  Map<String,Object> resultMap = new HashMap<>();  List<SimulationEquipment> list = simulationEquipmentService.selectSimulationEquipmentList(searchMap);  R r = R.ok();  resultMap.put("list",list);  r.setData(resultMap);  return r;  }   @ApiOperation(value = "新增仿真设备")  @PostMapping(value = "/insertSimulationEquipment")  public R insertSimulationEquipment(@RequestBody SimulationEquipment simulationEquipment){  //判断设备名称是否为空  if(null == simulationEquipment || StringUtils.isEmpty(simulationEquipment.getName())){  return R.error();  }  return simulationEquipmentService.insertSimulationEquipment(simulationEquipment);  }   @ApiOperation(value = "根据id查询仿真设备")  @GetMapping(value = "/selectSimulationEquipmentById")  public R selectSimulationEquipmentById(int id){  SimulationEquipment simulationEquipment = simulationEquipmentService.selectSimulationEquipmentById(id);  R r = R.ok();  HashMap<String,Object> map = new HashMap<>();  map.put("data",simulationEquipment);  r.setData(map);  return r;  }   @ApiOperation(value = "更新仿真设备")  @PostMapping(value = "/updateSimulationEquipment")  public R updateSimulationEquipment(@RequestBody SimulationEquipment simulationEquipment){  //判断id是否为空  if(null == simulationEquipment || null == simulationEquipment.getId()){  return R.error();  }  return simulationEquipmentService.updateSimulationEquipment(simulationEquipment);  }   @ApiOperation(value = "根据id删除仿真设备")  @GetMapping(value = "/deleteSimulationEquipmentById")  public R deleteSimulationEquipmentById(int id){  return simulationEquipmentService.deleteSimulationEquipmentById(id);  } } |

#### 业务逻辑接口

业务逻辑接口在一个典型的三层架构（表示层、业务逻辑层、数据访问层）或者更现代化的架构中起着非常重要的作用。它通常位于业务逻辑层，负责定义应用程序的业务逻辑规则和行为。

业务逻辑接口的一些重要作用：

封装业务逻辑：业务逻辑接口定义了应用程序的业务逻辑规则，将具体的业务逻辑实现与表示层（如控制器）和数据访问层（如DAO）分离开来。这样做有利于降低代码的耦合性，提高代码的可维护性和可测试性。

定义业务方法：业务逻辑接口中包含了各种业务方法的声明，这些方法描述了系统中各种业务功能的行为和规则，如用户注册、仿真设备、登录等。

提供规范：通过业务逻辑接口，可以清晰地定义系统中各个业务功能的规范和约定，使得整个应用程序的开发过程更加规范和有序。

便于扩展和替换：业务逻辑接口定义了业务功能的标准接口，使得在需要扩展或替换具体业务逻辑实现时更加容易，只需修改相应的实现类而不需要改动接口定义。

支持多态：通过业务逻辑接口，可以实现多态的特性，使得不同的实现类可以根据相同的接口进行统一调用。

在仿真设备业务中，我们创建了ISimulationEquipmentService 接口。

|  |
| --- |
| package com.hwadee.cqupt.service; import com.baomidou.mybatisplus.extension.service.IService; import com.hwadee.cqupt.common.R; import com.hwadee.cqupt.entity.SimulationEquipment; import java.util.HashMap; import java.util.List; /\*\*  \* @author: bxl  \* @date: 2023/7/3  \*/ public interface ISimulationEquipmentService extends IService<SimulationEquipment> {  List<SimulationEquipment> selectSimulationEquipmentList(HashMap<String,String> searchMap);  SimulationEquipment selectSimulationEquipmentById(int id);  R insertSimulationEquipment(SimulationEquipment simulationEquipment);  R updateSimulationEquipment(SimulationEquipment simulationEquipment);  R deleteSimulationEquipmentById(int id); } |

#### 业务逻辑实现类

业务逻辑实现类是业务逻辑接口的具体实现，它承担着将业务逻辑接口定义的规则和行为转化为具体功能的责任。业务逻辑实现类通常位于应用程序的业务逻辑层，负责执行各种具体的业务逻辑操作。

业务逻辑实现类的一些重要作用：

实现业务规则：业务逻辑实现类负责具体实现业务逻辑接口中定义的各种业务方法，根据实际需求编写具体的业务逻辑代码，例如处理订单、计算账单、验证用户权限等。

封装具体业务逻辑：业务逻辑实现类将业务逻辑接口定义的抽象业务规则转化为具体的功能实现，从而实现了业务逻辑的封装和具体化。

与数据访问层交互：在实现具体业务逻辑时，业务逻辑实现类通常需要与数据访问层（如DAO）进行交互，从数据库或其他数据源中获取数据并对其进行处理。

提供业务功能：通过业务逻辑实现类，应用程序可以提供各种具体的业务功能，满足用户需求并支持应用程序的核心业务流程。

支持业务逻辑的扩展和定制：业务逻辑实现类可以根据具体的需求进行定制和扩展，以满足不同业务场景下的需求变化。

在仿真设备业务中，我们创建了SimulationEquipmentServiceImpl业务逻辑实现类。

|  |
| --- |
| package com.hwadee.cqupt.service.impl;  import com.baomidou.mybatisplus.core.conditions.query.QueryWrapper; import com.baomidou.mybatisplus.extension.service.impl.ServiceImpl; import com.hwadee.cqupt.common.R; import com.hwadee.cqupt.entity.SimulationEquipment; import com.hwadee.cqupt.mapper.SimulationEquipmentMapper; import com.hwadee.cqupt.service.ISimulationEquipmentService; import org.springframework.stereotype.Service;  import javax.annotation.Resource; import java.util.HashMap; import java.util.List;  /\*\*  \* @author: bxl  \* @date: 2023/7/3  \*/ @Service public class SimulationEquipmentServiceImpl extends ServiceImpl<SimulationEquipmentMapper, SimulationEquipment> implements ISimulationEquipmentService {   @Resource  private SimulationEquipmentMapper simulationEquipmentMapper;   @Override  public List<SimulationEquipment> selectSimulationEquipmentList(HashMap<String,String> searchMap) {  QueryWrapper queryWrapper = new QueryWrapper(); // if(!StringUtils.isEmpty(searchMap.get("name"))){ // queryWrapper.like("name",searchMap.get("name")); // } // if(!StringUtils.isEmpty(searchMap.get("number"))){ // queryWrapper.like("number",searchMap.get("number")); // }  return simulationEquipmentMapper.selectList(queryWrapper);  }   @Override  public SimulationEquipment selectSimulationEquipmentById(int id) {  return simulationEquipmentMapper.selectById(id);  }   @Override  public R insertSimulationEquipment(SimulationEquipment simulationEquipment) {  int num = simulationEquipmentMapper.insert(simulationEquipment);  return num > 0 ? R.ok() : R.error();  }   @Override  public R updateSimulationEquipment(SimulationEquipment simulationEquipment) {  int num = simulationEquipmentMapper.updateById(simulationEquipment);  return num > 0 ? R.ok() : R.error();  }   @Override  public R deleteSimulationEquipmentById(int id) {  simulationEquipmentMapper.deleteById(id);  return R.ok();  } } |

#### 数据访问层接口

数据访问层接口在软件开发中扮演着至关重要的角色，它定义了应用程序与数据存储之间的交互规则和方法。以下是数据访问层接口的一些重要作用：

封装数据访问逻辑：数据访问层接口定义了对数据存储（如数据库、文件系统、外部API等）进行读取、写入、更新和删除等操作的规范，将数据存储相关的细节封装起来，使得业务逻辑层无需关心具体的数据存储实现细节。

提供数据访问方法：数据访问层接口中包含了与数据存储相关的各种方法声明，如查询数据、保存数据、更新数据等，为业务逻辑层提供了统一的数据访问接口。

实现数据访问逻辑：数据访问层接口的具体实现类负责将接口定义的数据访问方法转化为具体的数据库查询、写入或更新等实际操作，与数据库或其他数据存储进行交互。

提供数据访问的规范：通过数据访问层接口，可以定义数据访问的规范和约定，使得整个应用程序的数据访问过程更加规范和有序。

支持多数据存储源：数据访问层接口可以根据具体的需求支持多种数据存储源，例如关系型数据库、NoSQL数据库、文件系统等，而业务逻辑层无需关心不同数据存储源的差异。

总之，数据访问层接口通过定义数据访问规范和方法，实现了数据访问逻辑与业务逻辑的分离，提高了系统的可扩展性、可维护性和可测试性，是现代化软件架构中不可或缺的一部分。

在仿真设备业务中，我们创建了SimulationEquipmentMapper 数据访问层接口。

|  |
| --- |
| package com.hwadee.cqupt.mapper;  import com.baomidou.mybatisplus.core.mapper.BaseMapper; import com.hwadee.cqupt.entity.SimulationEquipment;  /\*\*  \* @author: bxl  \* @date: 2023/7/3  \*/ public interface SimulationEquipmentMapper extends BaseMapper<SimulationEquipment> { } |

#### 数据访问层实现类

数据访问层实现类是数据访问层接口的具体实现，其作用主要包括以下几个方面：

与数据存储进行交互：数据访问层实现类负责与具体的数据存储进行交互，例如数据库、文件系统或外部API等。它通过执行数据访问接口中定义的方法，实际进行数据的读取、写入、更新和删除操作。

封装数据访问逻辑：数据访问层实现类将数据访问接口中定义的数据访问规范转化为具体的数据库查询、写入或更新等操作，从而将数据存储相关的细节封装起来，使得业务逻辑层无需直接操作数据存储。

提供数据访问功能：数据访问层实现类提供了具体的数据访问功能，使得业务逻辑层能够通过调用其方法来实现对数据的操作，例如获取用户信息、存储订单数据、更新商品库存等。

支持事务管理：在需要保证数据操作的原子性、一致性和隔离性时，数据访问层实现类可以支持事务管理，确保数据操作的安全性。

优化数据访问性能：数据访问层实现类在设计时可以考虑优化数据访问的性能，例如通过合理的数据库查询语句、索引设计等方式来提高数据访问的效率。

总之，数据访问层实现类在应用程序中扮演着将数据访问接口定义的抽象规则转化为具体功能的关键角色，它负责实际与数据存储进行交互，并为业务逻辑层提供了可靠的数据访问功能。通过数据访问层实现类的封装和实现，应用程序能够更好地管理和操作数据，从而支持应用程序的核心业务功能。

在仿真设备业务中，我们使用了Mybatisplus的基础接口BaseMapper,从而实现对仿真设备的数据操作实现。

|  |
| --- |
| package com.baomidou.mybatisplus.core.mapper;  import com.baomidou.mybatisplus.core.conditions.Wrapper; import com.baomidou.mybatisplus.core.metadata.IPage; import com.baomidou.mybatisplus.core.toolkit.CollectionUtils; import com.baomidou.mybatisplus.core.toolkit.ExceptionUtils; import java.io.Serializable; import java.util.Collection; import java.util.List; import java.util.Map; import org.apache.ibatis.annotations.Param;  public interface BaseMapper<T> extends Mapper<T> {  int insert(T entity);   int deleteById(Serializable id);   int deleteById(T entity);   int deleteByMap(@Param("cm") Map<String, Object> columnMap);   int delete(@Param("ew") Wrapper<T> queryWrapper);   int deleteBatchIds(@Param("coll") Collection<?> idList);   int updateById(@Param("et") T entity);   int update(@Param("et") T entity, @Param("ew") Wrapper<T> updateWrapper);   T selectById(Serializable id);   List<T> selectBatchIds(@Param("coll") Collection<? extends Serializable> idList);   List<T> selectByMap(@Param("cm") Map<String, Object> columnMap);   default T selectOne(@Param("ew") Wrapper<T> queryWrapper) {  List<T> ts = this.selectList(queryWrapper);  if (CollectionUtils.isNotEmpty(ts)) {  if (ts.size() != 1) {  throw ExceptionUtils.mpe("One record is expected, but the query result is multiple records", new Object[0]);  } else {  return ts.get(0);  }  } else {  return null;  }  }   default boolean exists(Wrapper<T> queryWrapper) {  Long count = this.selectCount(queryWrapper);  return null != count && count > 0L;  }   Long selectCount(@Param("ew") Wrapper<T> queryWrapper);   List<T> selectList(@Param("ew") Wrapper<T> queryWrapper);   List<Map<String, Object>> selectMaps(@Param("ew") Wrapper<T> queryWrapper);   List<Object> selectObjs(@Param("ew") Wrapper<T> queryWrapper);   <P extends IPage<T>> P selectPage(P page, @Param("ew") Wrapper<T> queryWrapper);   <P extends IPage<Map<String, Object>>> P selectMapsPage(P page, @Param("ew") Wrapper<T> queryWrapper); } |

## 测试模型

针对更新仿真设备业务用例，我们进行测试模型设计，包含识别测试需求、定义测试用例、设计测试用例。

### 识别测试需求

根据更新仿真设备用例，确定系统需要具备对仿真设备的基础信息进行维护功能。

### 定义测试用例

根据更新仿真设备用例，定义更新仿真设备测试用例，包括：新增、编辑、删除、查看、查询、导出仿真设备等功能。测试用例应该覆盖用例中的基本流和备选流。

|  |  |
| --- | --- |
| 用例 | 测试用例 |
| 更新仿真设备 | 新增仿真设备(1个) |
| 编辑仿真设备(1个) |
| 删除仿真设备(1个) |
| 查看仿真设备(1个) |
| 查询仿真设备(3个) |
| 导出仿真设备(1个) |

### 设计测试用例

新增仿真设备

|  |  |  |  |
| --- | --- | --- | --- |
| 测试用例序号 | TC\_FZSB\_01 | 版本号 | 1.0 |
| 测试环境 | Win10，Google浏览器 | | |
| 前提条件 | 管理员登录成功，并选择仿真设备菜单下的仿真设备管理 | | |
| 测试步骤 | 1. 点击新增，填写新增仿真设备信息（设备编码、缩略图、设备名称、类别、软件系统、版本号、设备供应商、用途、所在实验室、设备状态） | | |
| 预期输出 | 1. 系统自动对各信息进行校验，必填项不填写或输入不正确格式的进行提示“请按xxx格式输入”。   2、各信息校验无误后，点击保存，提示保存成功，系统新增该仿真设备信息 | | |

删除仿真设备

|  |  |  |  |
| --- | --- | --- | --- |
| 测试用例序号 | TC\_FZSB\_02 | 版本号 | 1.0 |
| 测试环境 | Win10，Google浏览器 | | |
| 前提条件 | 管理员登录成功，并选择仿真设备菜单下的仿真设备管理 | | |
| 测试步骤 | 1、选择某一仿真设备后，点击删除；  2、系统提示管理员 “您确定要删除该条仿真设备数据?”， | | |
| 预期输出 | 1. 点击是，删除成功，系统从数据库删除指定仿真设备的数据 2. 点击否，则返回仿真设备列表页面。 | | |

查看仿真设备

|  |  |  |  |
| --- | --- | --- | --- |
| 测试用例序号 | TC\_FZSB\_03 | 版本号 | 1.0 |
| 测试环境 | Win10，Google浏览器 | | |
| 前提条件 | 管理员登录成功，并选择仿真设备菜单下的仿真设备管理 | | |
| 测试步骤 | 1. 在列表中选择某一仿真设备信息， 2. 点击查看按钮； | | |
| 预期输出 | 1. 系统跳转到该仿真设备详细页面，以表单的形式显示指定仿真设备信息 2. 仿真设备的各项信息显示正确，与列表显示的记录一致，没有空项或缺失项、乱码等。 | | |

查询仿真设备

|  |  |  |  |
| --- | --- | --- | --- |
| 测试用例序号 | TC\_FZSB\_04 | 版本号 | 1.0 |
| 测试环境 | Win10，Google浏览器 | | |
| 前提条件 | 管理员登录成功，并选择仿真设备菜单下的仿真设备管理 | | |
| 测试步骤 | 1、输入已有的正确设备名称、设备编号查询条件，点击查询按钮； | | |
| 预期输出 | 1. 查询成功，系统以列表显示仿真设备查询结果信息，包含 (设备编号、设备名称、类别、软件系统、版本号、设备供应商、用途、所在实验室、设备状态、更新时间)。 | | |

|  |  |  |  |
| --- | --- | --- | --- |
| 测试用例序号 | TC\_FZSB\_05 | 版本号 | 1.0 |
| 测试环境 | Win10，Google浏览器 | | |
| 前提条件 | 管理员登录成功，并选择仿真设备菜单下的仿真设备管理 | | |
| 测试步骤 | 输入查询条件是模糊、精确或组合条件 | | |
| 预期输出 | 按查询条件显示查询结果 | | |

|  |  |  |  |
| --- | --- | --- | --- |
| 测试用例序号 | TC\_FZSB\_06 | 版本号 | 1.0 |
| 测试环境 | Win10，Google浏览器 | | |
| 前提条件 | 管理员登录成功，并选择仿真设备菜单下的仿真设备管理 | | |
| 测试步骤 | 1、输入没有的、或者错误的设备名称、设备编号查询条件，点击查询按钮； | | |
| 预期输出 | 系统提示查询无结果 | | |

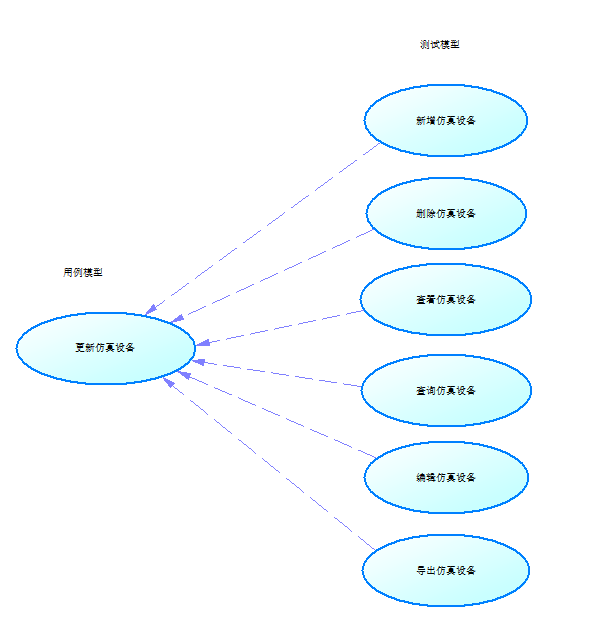
编辑仿真设备

|  |  |  |  |
| --- | --- | --- | --- |
| 测试用例序号 | TC\_FZSB\_07 | 版本号 | 1.0 |
| 测试环境 | Win10，Google浏览器 | | |
| 前提条件 | 管理员登录成功，并选择仿真设备菜单下的仿真设备管理 | | |
| 测试步骤 | 1、在列表中选择某一仿真设备信息列表后，  2、点击编辑按钮；  3、系统跳转到仿真设备信息编辑页面，系统加载并显示被选仿真设备信息详情（设备编码、缩略图、设备名称、类别、软件系统、版本号、设备供应商、用途、所在实验室、设备状态）；  4.管理员修改被选仿真设备信息的详细内容； | | |
| 预期输出 | 1. 系统自动对修改的信息进行校验，不正确的输入进行提示“请按xxx格式输入”。 2. 点击确认后，系统更新仿真设备信息 | | |

导出仿真设备

|  |  |  |  |
| --- | --- | --- | --- |
| 测试用例序号 | TC\_FZSB\_08 | 版本号 | 1.0 |
| 测试环境 | Win10，Google浏览器 | | |
| 前提条件 | 管理员登录成功，并选择仿真设备菜单下的仿真设备管理 | | |
| 测试步骤 | 1、输入设备名称、设备编号查询条件，点击查询按钮；  2、系统显示仿真设备信息。  3、点击导出按钮，。 | | |
| 预期输出 | 1. 按查询结果导出正确对应的excel格式数据 2. 导出的设备信息无错位、缺失、空白等现象 | | |

### 测试模型



## 部署与发布

### 部署环境搭建

Window环境

* 安装JDK。
* 配置环境变量。

Linux环境

* 安装JDK。
* 配置环境变量。

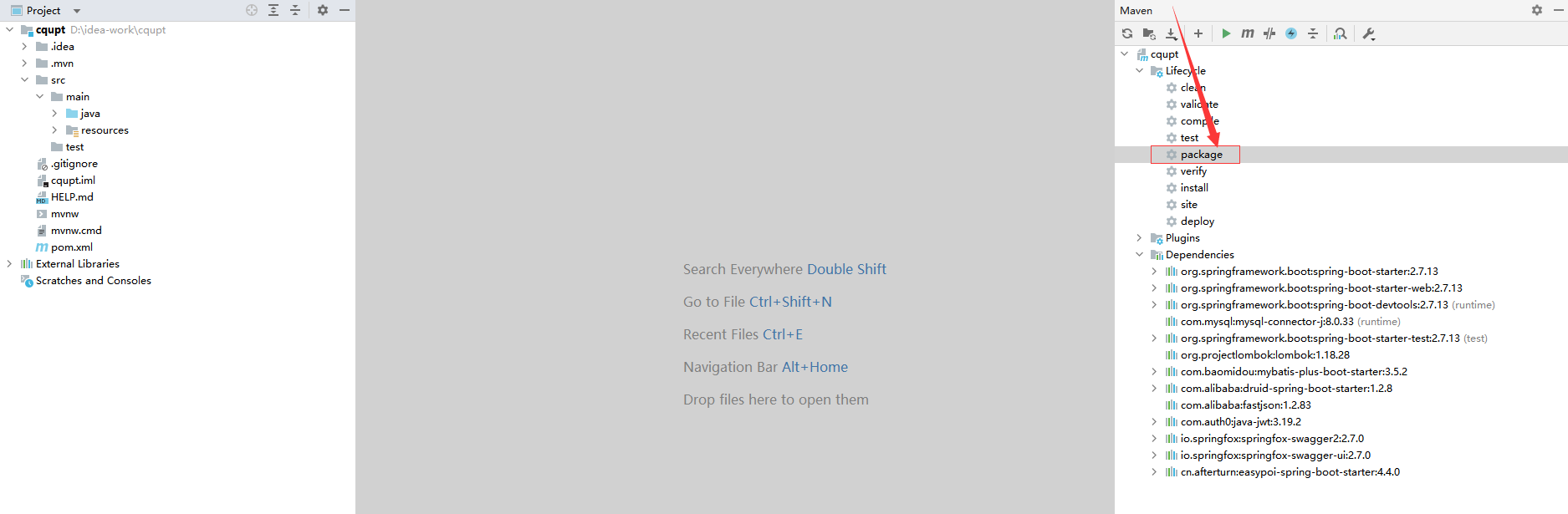
### 项目打包

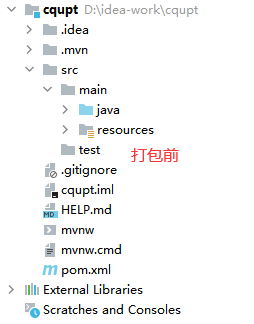
使用maven打包工具进行项目打包，打包后的结果为xxx.jar或者xxx.war，我们可以修改为cqupt.jar或者cqupt.war。

SpringBoot既可以打成war发布，也可以打成jar包发布。说一下区别：

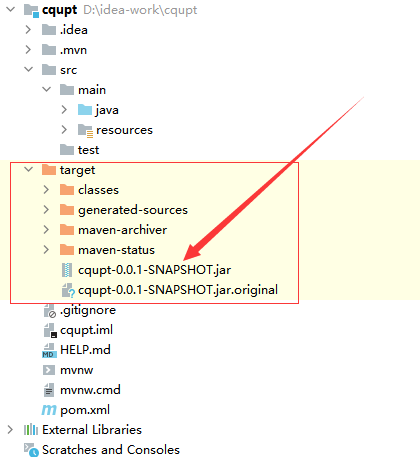
jar包：直接通过内置tomcat运行，不需要额外安装tomcat。如需修改内置tomcat的配置，只需要在SpringBoot的配置文件中配置。内置tomcat没有自己的日志输出，全靠jar包应用输出日志。但是比较方便，快速，比较简单。

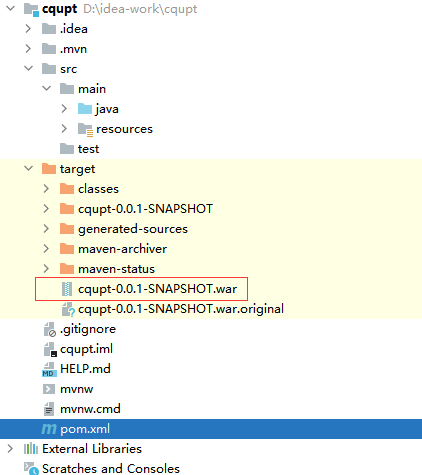
war包：传统的应用交付方式，需要安装tomcat，然后放到wabapps目录下运行war包，可以灵活选择tomcat版本，可以直接修改tomcat的配置，有自己的tomcat日志输出，可以灵活配置安全策略。相对打成jar包来说没那么快速方便。



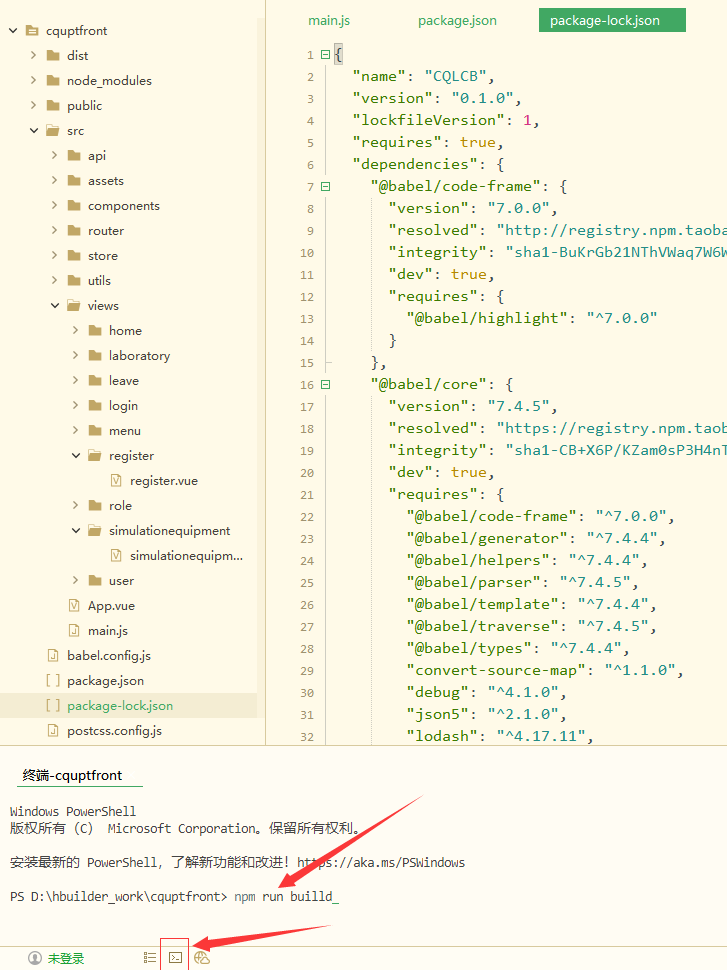


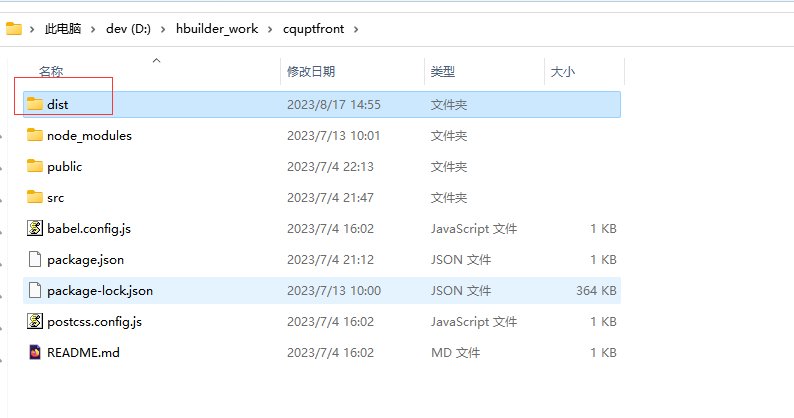
打包后：





前端Vue项目



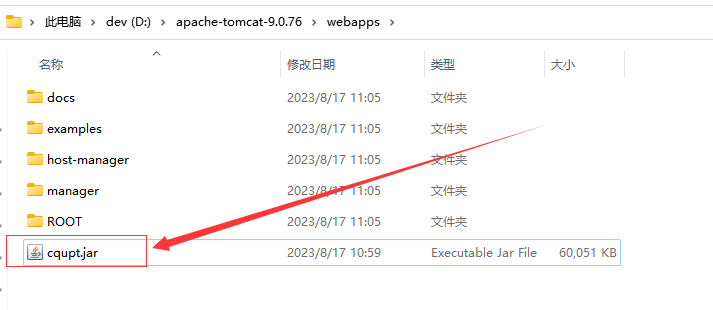


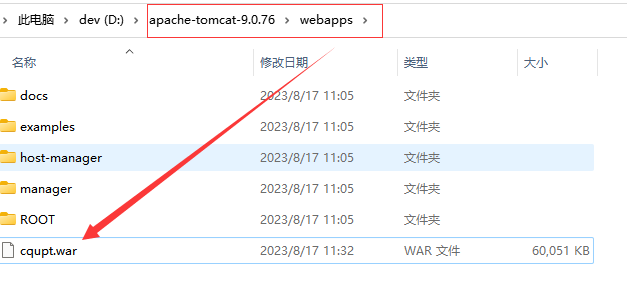
打开tomcat 服务器本地路径下的webapps文件夹，新建cqupt项目文件夹存放【来自vue项目dist 里面的index.html 和static 文件夹】,例如新建vue项目文件夹命名为cqupt。

### 部署

1、Windows部署

将打包好的项目部署到Tomcat服务器的webapps目录下即可。





2、linux部署

将war包放在webapps目录下

重启Tomcat

（1）进入Tomcat的bin目录

cd /usr/local/tomcat/bin

（2）启动Tomcat

./startup.sh