.NET 期末项目文档

项目选题：虚拟仿真实验平台教师端系统

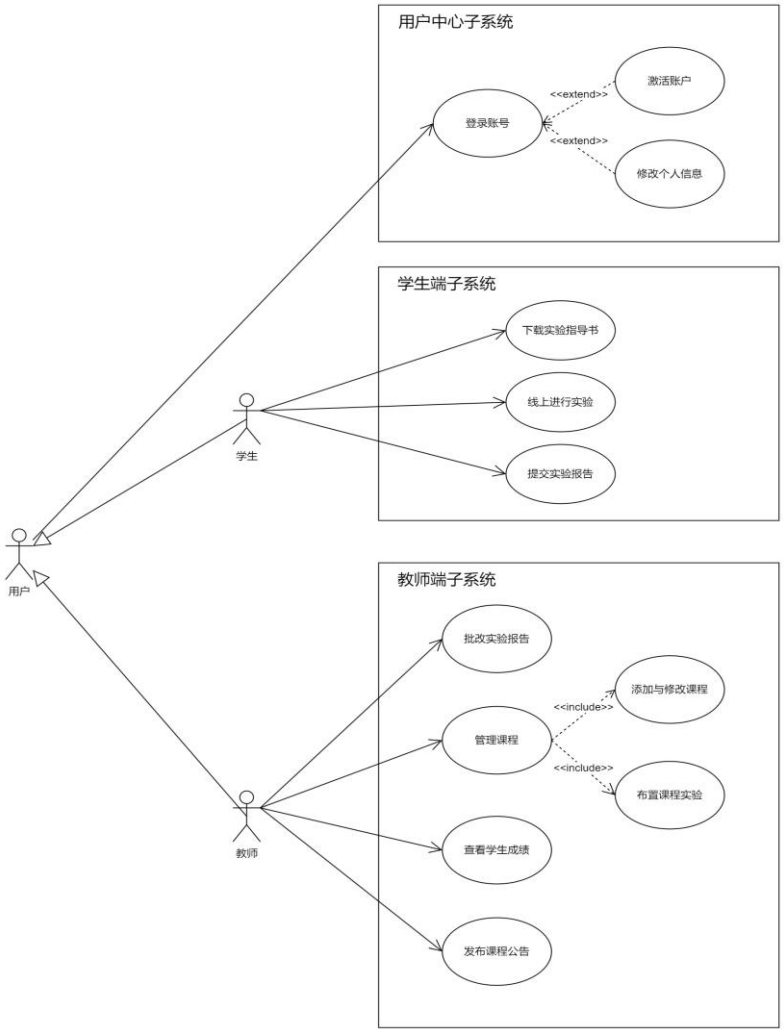
小组成员：2050865 黄彦铭 2051196 刘一飞

1. 项目简介

在我院“软件工程管理与经济”一课中，有许多实验供学生学习。但是，目前该课还处于学生手写实验报告，教师纸质批改实验报告的阶段。为了能够让该课的实验流程更加便捷，黄杰老师作为甲方，要求我们小组开发一个线上虚拟仿真实验平台。

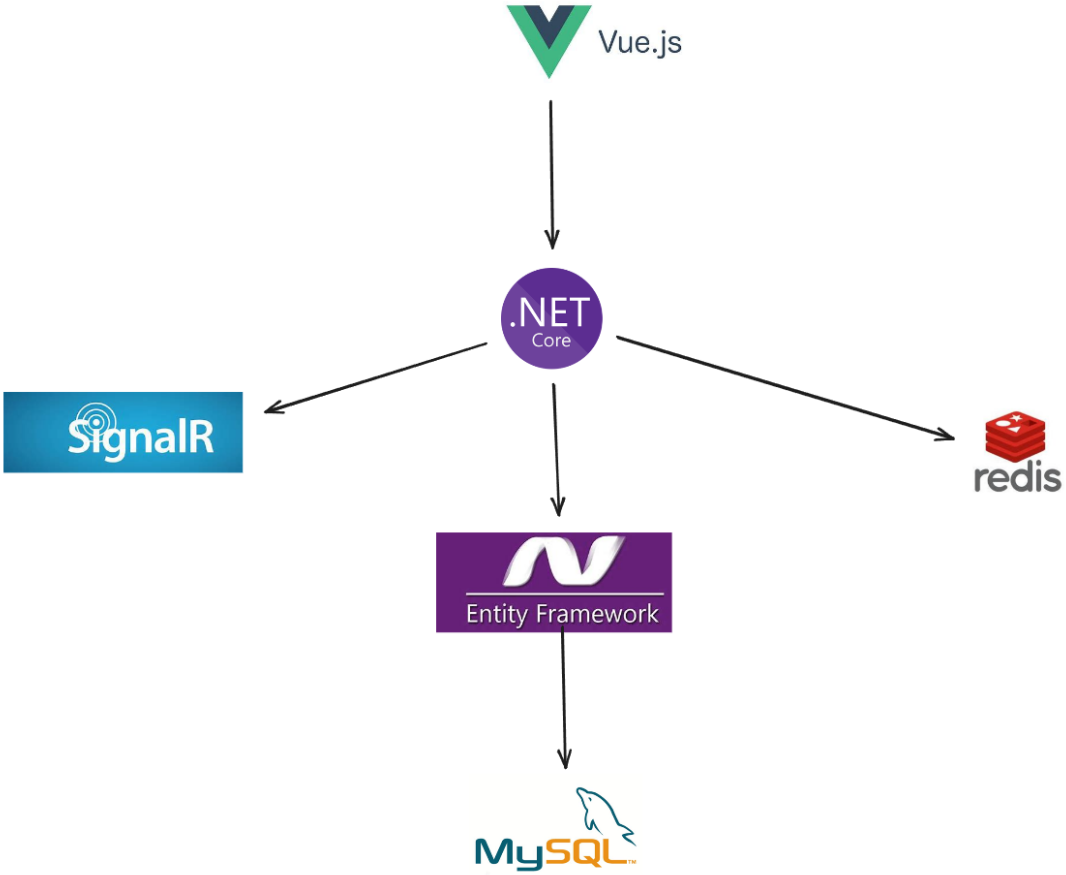
虚拟仿真实验平台分为学生端与教师端。学生可以在学生端系统中提交实验报告，教师可以在教师端系统中批改实验报告，以及能够更方便地管理课程信息。

系统的用例图如下：



在本次的课程项目中，我们的.NET 部分主要涉及到了用户中心子系统与教师端子系统。

2. 技术架构



3. 项目基本要求

3.1 C++/CLI

使用 C++/CLI 实现了邮箱合法性校验。

```
#pragma once
using namespace System;
namespace CLI {
    public ref class Class1
    {
    public:
        static bool ValidateEmail(String^ email)
        {
            // 判断邮箱是否可用
            if (String::IsNullOrEmpty(email))
            {
                return false;
            }

            // 判断邮箱合法性
            String^ pattern = "^[a-zA-Z0-9_.+-]+@[a-zA-Z0-9-]+\\.[a-zA-Z0-9-.]+$";
            System::Text::RegularExpressions::Regex^ regex = gcnew System::Text::RegularExpressions::Regex(pattern);
            return regex->IsMatch(email);
        }
    };
}
```

3.2 Win32 DLL

使用 Win32 DLL 实现了上传文件时，将用户提供的路径转化为服务器上文件系统的路径。

```
# define _CRT_SECURE_NO_WARNINGS
#include <ctime>
#include <string>

extern "C" __declspec(dllexport) void FileNameGenerate(const char* path, const char* fileName, char* result, int maxLength)
{
    std::string str_path(path);
    std::string str_fileName(fileName);

    std::string suffix;
    std::size_t dotPos = str_fileName.rfind('.');
    if (dotPos != std::string::npos && dotPos < str_fileName.length() - 1)
    {
        suffix = str_fileName.substr(dotPos);
    }

    std::string concatenated = "vse" + str_path + std::to_string(time(0)) + suffix;

    if (concatenated.length() < maxLength)
    {
        strcpy(result, concatenated.c_str());
    }
}
```

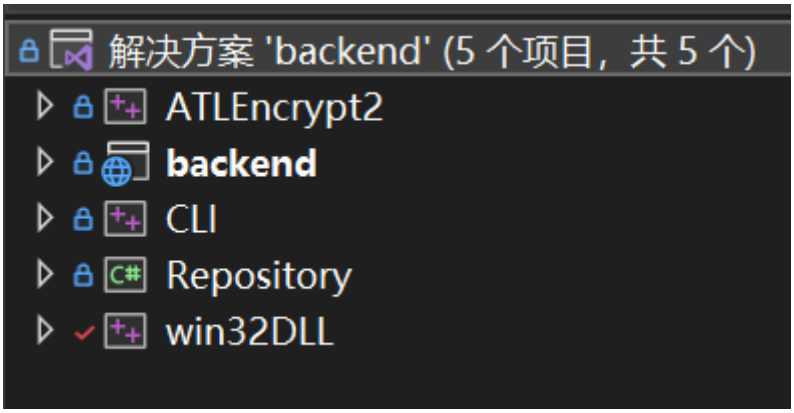
3.3 COM 组件

使用 COM 组件实现了 MD5 加密算法，输入输出均为一个字符串。

```
STDMETHODIMP CMD5Encryption::Encryption(BSTR strBefore, BSTR* strAfter)
{
    unsigned char* lpszText = (unsigned char*)_com_util::ConvertBSTRToString(strBefore);
    MD5 iMD5;
    const char* cLpszText = _com_util::ConvertBSTRToString(strBefore);
    iMD5.GenerateMD5(lpszText, strlen(cLpszText));
    string result = iMD5.ToString();
    _bstr_t bstr_t(result.c_str());
    *strAfter = bstr_t.GetBSTR();
    return S_OK;
}
```

3.4 程序集

我们的项目总共分为 5 个程序集，如下图所示：



- backend：负责提供后端提供 WebAPI 的所有逻辑，包括所有的后端接口以及拦截器。
- Repository：负责服务器端程序与数据库交互，主要是利用 Entity Framework 将实体类与数据库表做关系对象映射。
- ATLEncrypt2：该 COM 组件负责加密算法。该程序集实现了 MD5 加密算法，在后端存入数据库的密码都要经过 MD5 加密。
- CLI：负责验证邮箱合法性。用户在登录界面中，如果登录的邮箱格式错误，后端会返回错误信息并显示在前端。
- win32DLL：负责将用户提供的文件路径转换为服务器上的文件路径。

4. 技术亮点

4.1 JWT

JWT（JSON Web Token）是一种用于在网络应用中进行身份验证和信息传递的开放标准。它是一种轻量级的安全令牌，以 JSON 格式表示，并使用数字签名或加密进行验证和保护信息的完整性。在本项目中，使用 JWT 作为身份验证和授权的机制，提供安全的身份验证方式。

在项目中的 JwtUtil 类中，定义了生成 JWT 的算法：

```
// 生成jwt字符串，三天后过期  JWT(json web token)
1 个引用
public static string Sign(string userId)
{
    //Header,选择签名算法
    var signingAlogorithm = SecurityAlgorithms.HmacSha256;
    //Payload,存放用户信息，下面我们放了一个用户id
    var claims = new[]
    {
        new Claim("user_id",userId)
    };
    //Signature
    //取出私钥并以utf8编码字节输出
    var secretByte = Encoding.UTF8.GetBytes(_configuration["Authentication:SecretKey"]);
    //使用非对称算法对私钥进行加密
    var signingKey = new SymmetricSecurityKey(secretByte);
    //使用HmacSha256来验证加密后的私钥生成数字签名
    var signingCredentials = new SigningCredentials(signingKey, signingAlogorithm);
    //生成Token
    var Token = new JwtSecurityToken(
            issuer: _configuration["Authentication:Issuer"],        //发布者
            audience: _configuration["Authentication:Audience"],    //接收者
            claims: claims,                                         //存放的用户信息
            notBefore: DateTime.UtcNow,                             //发布时间
            expires: DateTime.UtcNow.AddDays(3),                      //有效期设置为3天
            signingCredentials                                      //数字签名
        );
    //生成字符串token
    var TokenStr = new JwtSecurityTokenHandler().WriteToken(Token);
    return TokenStr;
}
```

在用户登录接口中，如果用户账号已经激活并且成功登录，则调用 Sign 方法生成 JWT 令牌，并将 JWT 令牌传递给客户端，存储在本地储存中。

```
[Route("login")]
[ProducesResponseType(StatusCodes.Status200OK)]
0 个引用
public ActionResult<Result> Login(UserLoginDto userLoginDto)
{
    string username = userLoginDto.username;
    string password = userLoginDto.password;
    string school = userLoginDto.school;

    // 使用CLI程序集进行邮箱合法性判断
    bool isValid = Class1.ValidateEmail(username);
    if (!isValid)
    {
        return Result.Fail(10010, "邮箱格式错误");
    }

    User user = ctx.Users.SingleOrDefault(u => u.email == username && u.school == school);
    if (user == null )
    {
        return Result.Fail(10001, "账号不存在");
    }
    //密码要MD5加密
    if (user.password != new MD5Encryption().Encryption(password)) {
        return Result.Fail(10002, "密码错误");
    }

    if (user.status == 1)
    {
        // 用户账号已经激活，返回token
        long index = user.index;
        string token = JwtUtil.Sign(index.ToString());

        Dictionary<string, string> res = new Dictionary<string, string>();
        res.Add("token", token);
        return Result.Success(res);
    }
```

在服务器端的过滤器中，同样定义了如下代码：

```
public void OnAuthorization(AuthorizationFilterContext context)
{
    //请求的地址
    var url = context.HttpContext.Request.Path.Value;
    Console.WriteLine(url);
    if (url == "/api/login" || url == "/api/review/school-name" || url == "getMD5" || url == "setAllMd5") return;// 以上接口不需要拦截

    // 取出 Authorization 头部
    string authHeader = context.HttpContext.Request.Headers["Authorization"];

    // 如果 Authorization 头部为空，或者不是以 Bearer 开头，返回 401 未授权响应
    if (string.IsNullOrEmpty(authHeader) || !authHeader.StartsWith("Bearer ", StringComparison.OrdinalIgnoreCase))
    {
        context.Result = new UnauthorizedResult();
        return;
    }

    // 取出 bearer token
    string bearerToken = authHeader.Substring("Bearer ".Length).Trim();

    if (!JwtUtil.CheckSign(bearerToken))
    {
        context.Result = new UnauthorizedResult();
        return;
    }
    string userId = JwtUtil.GetUserId(bearerToken);
    Console.WriteLine("用户ID:", userId);
    UserIndex.Value = long.Parse(userId);
}
```

即除了用户在登录阶段需要使用的接口不需要拦截之外，其他接口会被拦截，并且判断请求头中是否携带了合法的 JWT 令牌，如果没有携带或 JWT 不合法，则接口请求失败。

与此同时，在 JWT 的载荷中还携带了用户的 ID 信息，因此在拦截器验证 JWT 合法性从而验证用户身份的同时，还能够从 Payload 中提取用户的 ID 信息，用于后续逻辑的实现。

```
// 定义一个线程域，存放登录的用户index
private static ThreadLocal<long> UserIndex = new ThreadLocal<long>();
```

为了能够使拦截器中获取到的 userId 能够在 Controller 中可用，拦截器中定义了一个线程域单独用于存放登录用户的 Id，并且对外提供静态方法，来获取用户信息。

```
// 对外提供了静态的方法：getLoginUser()来获取User信息
9 个引用
public static long GetLoginUser()
{
    return UserIndex.Value;
}
```

4.2 邮件发送

项目中使用了 .NET 框架中的 SMTP（Simple Mail Transfer Protocol）邮件发送技术，来实现用户在激活账户时发送验证码邮件的功能。

```
public static string sendingMail(string receiver, string code)
{
    string smtpService = "smtp.qq.com";
    string sendEmail = "3155002905@qq.com";
    string sendpwd = "blmkcdggktphdgdb";


    //确定smtp服务器地址 实例化一个Smtp客户端
    SmtpClient smtpclient = new SmtpClient();
    smtpclient.Host = smtpService;
    //smtpClient.Port = "";//qq邮箱可以不用端口

    //确定发件地址与收件地址
    MailAddress sendAddress = new MailAddress(sendEmail);
    MailAddress receiveAddress = new MailAddress(receiver);

    //构造一个Email的Message对象 内容信息
    MailMessage mailMessage = new MailMessage(sendAddress, receiveAddress);
    mailMessage.Subject = "虚拟实验仿真系统VSB:请验证您的邮箱";
    mailMessage.SubjectEncoding = System.Text.Encoding.UTF8;
    mailMessage.Body = "您的验证码为" + code;
    mailMessage.BodyEncoding = System.Text.Encoding.UTF8;

    //邮件发送方式  通过网络发送到smtp服务器
    smtpclient.DeliveryMethod = SmtpDeliveryMethod.Network;

    //如果服务器支持安全连接，则将安全连接设为true
    smtpclient.EnableSsl = true;
    try
    {
        //是否使用默认凭据，若为false，则使用自定义的证书，就是下面的networkCredential实例对象
        smtpclient.UseDefaultCredentials = false;

        //指定邮箱账号和密码,需要注意的是，这个密码是你在QQ邮箱设置里开启服务的时候给你的那个授权码
        NetworkCredential networkCredential = new NetworkCredential(sendEmail, sendpwd);
        smtpclient.Credentials = networkCredential;

        //发送邮件
        smtpclient.Send(mailMessage);
        Console.WriteLine("发送邮件成功");

    }
    catch (System.Net.Mail.SmtpException ex) { Console.WriteLine(ex.Message, "发送邮件出错"); }
    return "邮件发送成功!";
}
```

主要使用了 System.Net.Mail 命名空间中的相关类，包括：

1. SmtpClient 类：用于与 SMTP 服务器进行通信，并发送电子邮件。
2. MailMessage 类：用于构建电子邮件的内容，包括主题、正文和发件人/收件人等信息。
3. MailAddress 类：用于表示邮件地址，可设置发件人和收件人的电子邮箱地址。
4. NetworkCredential 类：用于指定发件人邮箱的账号和密码（或授权码），以进行身份验证。

4.3 Redis 实现验证码功能

项目中，使用 StackExchange.Redis 这个 .NET Redis 客户端库来连接和操作 Redis 数据库，以实现的验证码的存储功能。

```
using StackExchange.Redis;
namespace backend.Tools
{
    3 个引用
    public static class RedisTools
    {
        public static StackExchange.Redis.IDatabase db;
        0 个引用
        static RedisTools(){
            // 创建连接到 Redis 的 ConnectionMultiplexer 实例，参数可以根据实际情况进行设置
            ConnectionMultiplexer redis = ConnectionMultiplexer.Connect("localhost:6379");

            // 获取 Redis 数据库
            db = redis.GetDatabase();
        }
        1 个引用
        public static void saveVerifyCode(string username, string code) {
            TimeSpan expiresIn = TimeSpan.FromMinutes(3); // 设置验证码有效期为 3 分钟
            db.StringSet(username, code, expiresIn); // 将验证码存储到 Redis 中
            return;
        }
        1 个引用
        public static string queryVerifyCode(string username)
        {
            RedisValue code = db.StringGet(username); // 从 Redis 中获取验证码
            if (!code.HasValue)
            {
                // 数据不存在
                return null;
            }
            else
            {
                // 数据存在
                return code.ToString();
            }
        }
    }
}
```

上面的 RedisTools 类中，实现了 saveVerifyCode 和 queryVerifyCode 两个方法，用于实现验证码在 redis 中的存储和查询。

```
if (user.status == 1)
{
    // 用户账号已经激活，返回token
    long index = user.index;
    string token = JwtUtil.Sign(index.ToString());

    Dictionary<string, string> res = new Dictionary<string, string>();
    res.Add("token", token);
    return Result.Success(res);
}
else
{
    // 生成六位随机验证码
    string code = CodeTools.GenerateCode();
    // 发送验证邮件
    EmailSender.sendingMail(user.email, code);
    // 将验证码存入redis
    RedisTools.saveVerifyCode(username, code);
    return Result.Fail(1, "账户需要激活，验证码已发送");
}
```

在用户登录接口中，如果用户的账户没有被激活，则会生成六位随机验证码存入 Redis 中，用于此后用户激活账号时验证身份使用。

4.4 SignalR

SignalR 是一个开源的.NET 库，用于在 Web 应用程序中实现实时通信功能，包括服务器到客户端的推送通知和客户端到服务器的双向通信。在项目中，使用 SignalR 实现了系统消息的时时通知。

```
6 个引用
public class SignalRHub: Hub
{
    private string Lock = "";
    private readonly IHubContext<SignalRHub> _hubContext;
    private static object lockObject = new object();
    0 个引用
    public SignalRHub(IHubContext<SignalRHub> hubContext)
    {
        _hubContext = hubContext;
    }
    /// <summary>
    /// 客户连接成功时触发
    /// </summary>
    /// <returns></returns>
    0 个引用
    public override async Task OnConnectedAsync()
    {
        var connectionId = Context.ConnectionId;
        Console.WriteLine("Websocket Connected:", connectionId);

        // 在此处启动定时任务，定时向客户端推送消息
        StartTimer(connectionId);

        await base.OnConnectedAsync();
    }
```

继承自 Hub 类的 SignalRHub 类定义了服务器端与客户端之间的通信方法。在 SignalRHub 类中，每当有客户端连接，都会调用 OnConnectedAsync 启动定时任务，定时向客户端推送消息。

在客户端中，创建了 SignalR 的连接，并且使用 connection.on 注册事件处理程序，当服务器端发送命名为 "ReceiveMessage" 的消息时，该事件处理程序将会被触发，重新去服务器端获取当前用户的所有消息，更新消息列表。

```
//.net core 版本中默认不会自动重连，需手动调用 withAutomaticReconnect
const connection = new signalR.HubConnectionBuilder()
  .withAutomaticReconnect() //断线自动重连
  .withUrl(hubUrl) //传递参数Query["access_token"]
  .build();
// 心跳包设置
connection.serverTimeoutInMilliseconds = 24e4;
connection.keepAliveIntervalInMilliseconds = 12e4;

connection.on("ReceiveMessage", (message) => {
  console.log("Received message:", message);
  getAllMessage()
  .then((res) => {
    messageList.value = res.data;
    console.log("获取到的消息信息为", messageList.value);
  })
  .catch((err) => {
    console.log(err);
  });
});
```

4.5 动态路由

教师可以增减课程，课程的前端路由必须随之改变，这就要求前端显示的侧边栏路由，需要后端动态返回：

后端需要动态地查询该名教师目前教了哪些课，生成的路由以 json 形式返回给前端。前端每次加载页面的时候都需要调用该接口，以生成侧边栏的菜单。

```
actions: {
  // getAuthButtonList
  async getAuthButtonList() {
    const { data } = await getAuthButtonListApi();
    this.authButtonList = data;
  },
  // getAuthMenuList
  async getAuthMenuList() {
    const { data } = await getAuthMenuListApi();
    this.authMenuList = data;
  },
  // setRouteName
  async setRouteName(name: string) {
    this.routeName = name;
  }
}
```

4.6 线程池

在系统中，学生提交实验报告这一行为是流量不平稳的。在大部分时间，学生提交实验报告的接口访问并不频繁。但如果到了大家共同的 ddl 时间前，例如临近晚上 12 点时，提交实验报告的接口访问量会骤增。

为了保证在高并发时能够有足够多的线程数处理接口调用，又要保证在接口访问低谷期不造成资源浪费，我们选择采用线程池来统一管理并发资源：

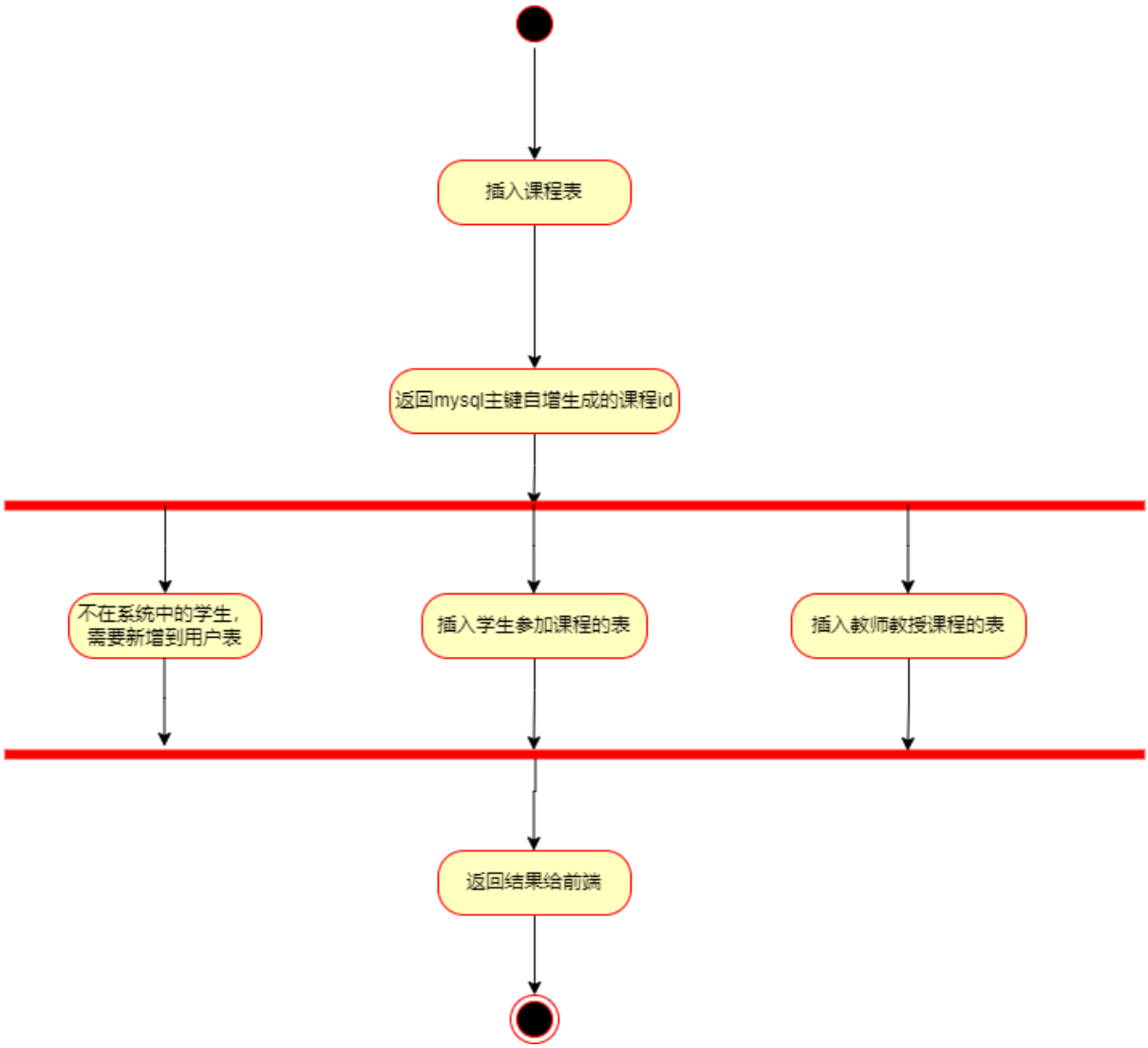
```
ThreadPool.SetMinThreads(5, 5);
ThreadPool.SetMaxThreads(100, 100);

ThreadPool.QueueUserWorkItem(_ =>
{
```

我们将最小空闲线程数设为 5，最大线程数设为 100。这样在并发访问频率小的时间段，只需要很少的几个线程就可以处理并发；当在提交作业的高峰期时，线程数能够增长到较大的数目，处理高并发的问题。

4.7 异步编程

后端中对于一些比较复杂的接口，使用了基于任务的异步编程。

例如，在添加课程的接口，需要涉及到 4 个表的插入，具体如下图所示：



其中，有三类插入操作可以并行，故采用异步编程的思想。让三类插入操作并行，等待这三类操作全部结束后再返回给前端。

```
// 并行执行所有数据库操作
await Task.WhenAll(studentTasks.Concat(teacherTasks).Concat(studentAttendTasks));
```

4.8 事务

如果在一个接口内涉及到多个对数据库进行更改的操作，则有可能出现某个操作失败而产生脏写。为了避免这种情况，我们需要将整个接口的操作当成一个事务，如果事务中有任何一个环节出错，都必须回滚这个事务。

在代码中，使用 TransactionScope 实现事务。只有当事务运行到了 ts.Complete()，才会提交事务，否则回滚事务。

```
using (var ts = new
TransactionScope(TransactionScopeAsyncFlowOption.Enabled))
{
    try
    {
        //......
        ts.Complete();
    }
    catch(Exception e)
    {
        //......
    }
}
```