# 1.Dapper框架介绍

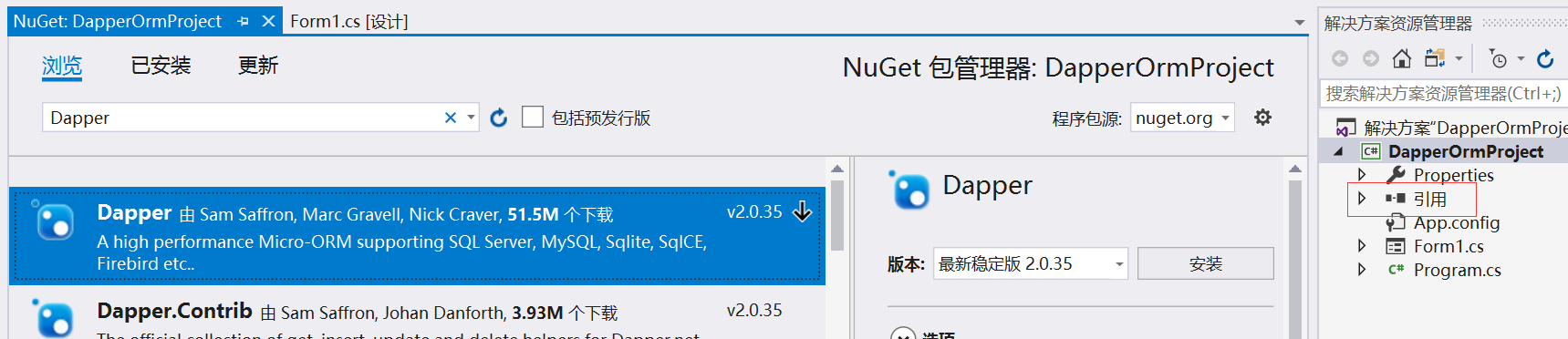
Dapper是.NET开发中一款轻量级的ORM映射框架，它轻量级、速度快，与同类型ORM框架如EF框架相比具有更高的执行效率。支持多种数据库，可以在所有的Ado.NET Providers下工作，包括sqlite、oracle、mysql、postgreSQL与SqlServer数据库。可以与数据库进行一对一、一对多、多对多的关系映射。它内部通过Emit反射IDataReder序列队列，来快速得到和产生对象，性能高。支持.NET Core，是一个跨平台的框架。

Dapper框架官网：<https://dapper-tutorial.net/dapper>

# 2.Dapper框架的基本应用

## 2.1.Dapper框架的安装引用

使用Dapper框架首先需要在项目中引入Dapper的安装包，在项目中添加引用浏览外部引用包，选择Dapper进行下载即可将Dapper框架相关的组件引入到解决方案中。具体操作如下：

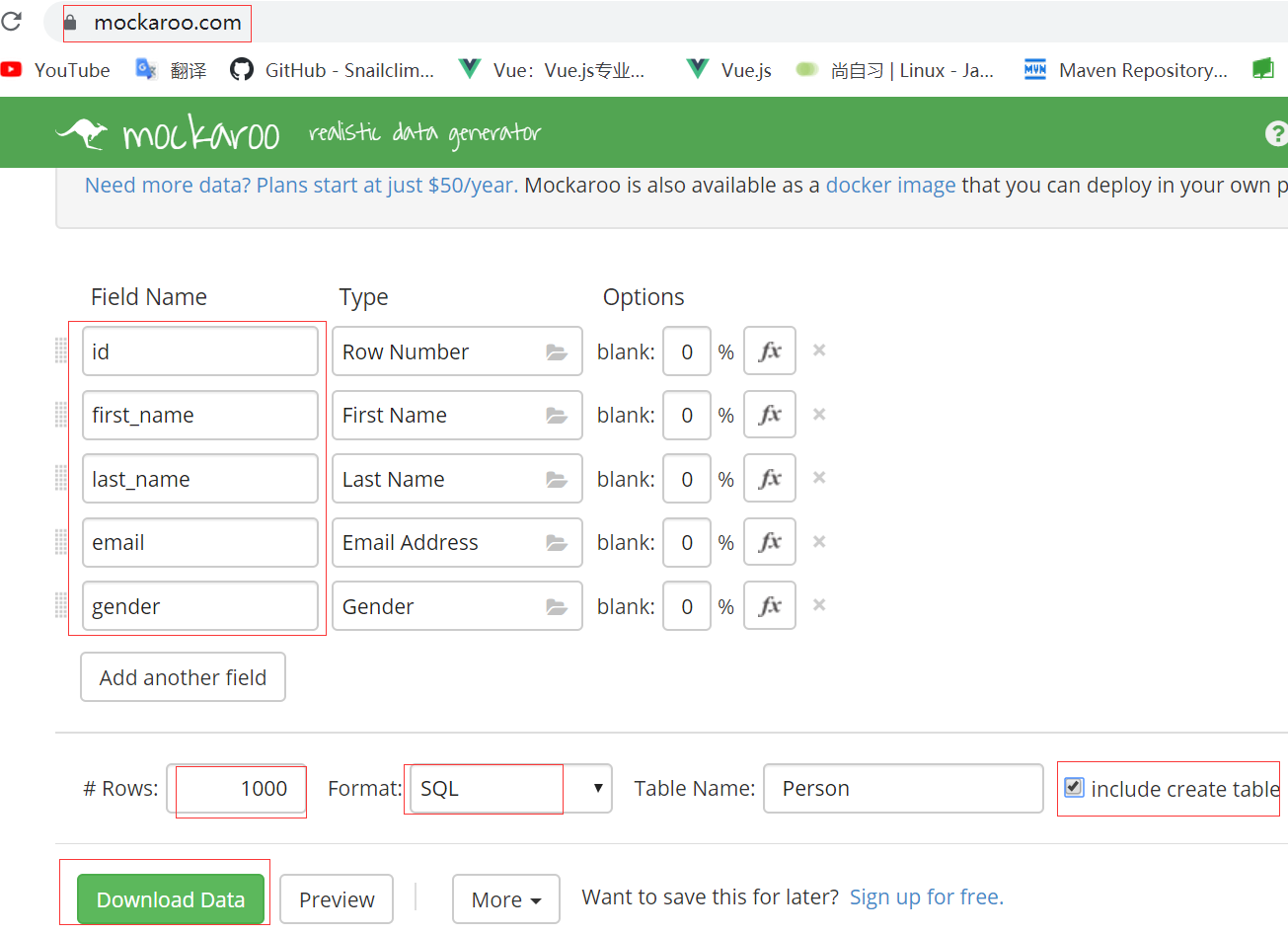


## 2.2.基于mockaroo构造虚拟数据

Mockaroo网址：<https://www.mockaroo.com/>

Mockaroo是一个模拟虚拟数据的一个数据构造网站，可以通过它来构造虚拟的数据库表中的数据，并生成sql文件，非常方便，这里简单地列出构造数据的表与相应的表中的字段列。

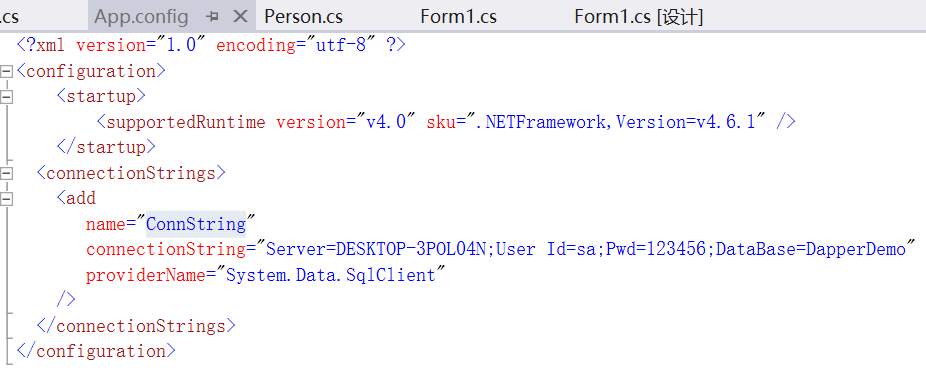
这里使用mockaroo网站构造数据列并下载到对应的sql如下图所示操作即可：



选择Download Data后可以自动生成Person.sql文件。

## 2.3.Dapper与数据库访问操作

使用Dapper连接Sqlserver数据库需要使用到连接字符串，连接字符串是在App.config文件中配置，需要添加如下连接配置：

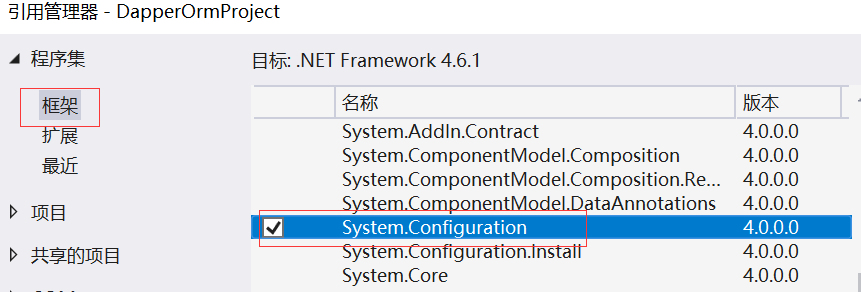


配置如下：

|  |
| --- |
| <connectionStrings>  <add  name="ConnString"  connectionString="Server=DESKTOP-3POL04N;User Id=sa;Pwd=123456;DataBase=DapperDemo"  providerName="System.Data.SqlClient"  />  </connectionStrings> |

构造连接访问助手类DBHelper类：

①首先需要在项目中添加DBHelper的类型引用System.Configuration；



②构建DBHelper数据库访问类：

|  |
| --- |
| //数据访问助手类  public static class DBHelper  {  /// <summary>  /// 从配置文件中读取数据库连接字符串  /// </summary>  public static string ConnString  {  get  {  return ConfigurationManager.ConnectionStrings["ConnString"].ConnectionString;  }  }  } |

### 2.3.1.按条件进行查询

条件查询及Sql注入问题的解决：

Sql注入问题主要是使用SqlConnection对象的Query方法进行匿名参数传递，需要对应sql语句中的变量的值以及使用传入的参数进行赋值。

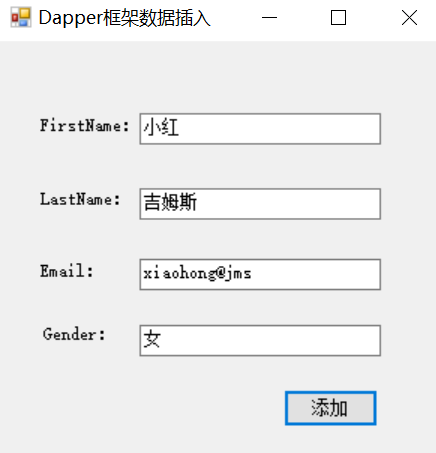
|  |
| --- |
| /// <summary>  /// 根据用户姓氏查询用户集合  /// </summary>  /// <param name="lastName"></param>  /// <returns></returns>  public List<Person> FindListByLastName(string lastName)  {  //IDbConnection由于Dapper ORM的操作实际上是对IDbConnection类的扩展，所有的方法都是该类的扩展方法。  using (IDbConnection db = new SqlConnection(DBHelper.ConnString))  {  //C#6的语法：容易引起sql注入的问题,如：select \* from Person where last\_name = 'Crevy' or '1' = '1';  string sql = $"select \* from Person where last\_name = '{lastName}'";  //解决sql注入的问题，注意以下的参数对应关系  string sqlQuery = $"select \* from Person where last\_name = @tempName";  return db.Query<Person>(sqlQuery, new { tempName = lastName }).ToList();  // return db.Query<Person>(sqlQuery).ToList(); //转化为List的类型返回  }  } |

### 2.3.2.按输入进行插入

执行插入数据代码：

|  |
| --- |
| /// <summary>  /// 根据界面输入插入对应数据  /// </summary>  /// <param name="lastName"></param>  /// <returns></returns>  public bool InsertPersonData(Person person)  {  //IDbConnection由于Dapper ORM的操作实际上是对IDbConnection类的扩展，所有的方法都是该类的扩展方法。  using (IDbConnection db = new SqlConnection(DBHelper.ConnString))  {  string insertSql = "insert into Person(first\_name, last\_name, email, gender) values " +  "(@First\_Name, @Last\_Name, @Email, @Gender)";  int resNum = db.Execute(insertSql, new Person  {  First\_Name = person.First\_Name,  Last\_Name = person.Last\_Name,  Email = person.Email,  Gender = person.Gender,  });  return resNum > 0;  }  } |

数据插入：



执行结果：



### 2.3.3.修改特定ID的列

修改首先根据界面上输入的ID进行查询显示，之后在执行对应的修改方法，具体代码如下所示：

|  |
| --- |
| /// <summary>  /// 根据界面输入ID查询出Person信息  /// </summary>  /// <param name="lastName"></param>  /// <returns></returns>  public Person QueryPersonById(int id)  {  Person person = null;  //IDbConnection由于Dapper ORM的操作实际上是对IDbConnection类的扩展，所有的方法都是该类的扩展方法。  using (IDbConnection db = new SqlConnection(DBHelper.ConnString))  {  string querySql = "select \* from Person where id = @tempId";  person = db.Query<Person>(querySql, new { tempId = id }).FirstOrDefault();  }  return person;  }  /// <summary>  /// 根据传入的Person更新数据库表中数据  /// </summary>  /// <param name="person"></param>  public bool UpdatePerson(Person person)  {  //IDbConnection由于Dapper ORM的操作实际上是对IDbConnection类的扩展，所有的方法都是该类的扩展方法。  using (IDbConnection db = new SqlConnection(DBHelper.ConnString))  {  string updateSql = "update Person set " +  "first\_name = @First\_Name, last\_name = @Last\_Name, email = @Email, gender = @Gender " +  "where id = @ID";  //注意：这里的First\_Name、Last\_Name等这些字段都是person赋值的，所以必须和Person  //类中的属性一致  int exeRes = db.Execute(updateSql, person);  return exeRes > 0;  }  } |

界面上的点击按钮事件编写：

|  |
| --- |
| private void searchBtn\_Click(object sender, EventArgs e)  {  PersonService service = new PersonService();  string idStr = this.searchID.Text;  if (!int.TryParse(idStr, out int id))  {  MessageBox.Show("ID必须为正整数", "警告");  return;  }  Person person = service.QueryPersonById(id);  this.tb\_firstName.Text = person.First\_Name;  this.tb\_lastName.Text = person.Last\_Name;  this.tb\_Email.Text = person.Email;  this.tb\_Gender.Text = person.Gender;  }  private void update\_Btn\_Click(object sender, EventArgs e)  {  PersonService service = new PersonService();  bool updateRes = service.UpdatePerson(new Person  {  ID = Convert.ToInt32(this.searchID.Text),  First\_Name = this.tb\_firstName.Text,  Last\_Name = this.tb\_lastName.Text,  Email = this.tb\_Email.Text,  Gender = this.tb\_Gender.Text  });  MessageBox.Show(updateRes ? "数据更新成功" : "数据更新失败");  } |

修改结果：





### 2.3.4.删除特定的数据列

删除数据部分比较简单，下面直接给出删除代码部分：

|  |
| --- |
| /// <summary>  /// 根据传入的ID进行数据删除  /// </summary>  /// <param name="person"></param>  public bool DeleteDataById(int id)  {  //IDbConnection由于Dapper ORM的操作实际上是对IDbConnection类的扩展，所有的方法都是该类的扩展方法。  using (IDbConnection db = new SqlConnection(DBHelper.ConnString))  {  string delSql = "delete from person where id = @ID";  //类中的属性一致  int exeRes = db.Execute(delSql, new { ID = id});  return exeRes > 0;  }  } |

# 3.Dapper框架的高级应用

## 3.1.Dapper调用存储过程

**存储过程是一组预编译的SQL语句**。

**使用存储过程有以下优点：**

1.允许模块化程序设计，就是说只需要创建一次，在以后的程序中就可以调用该过程任意次。

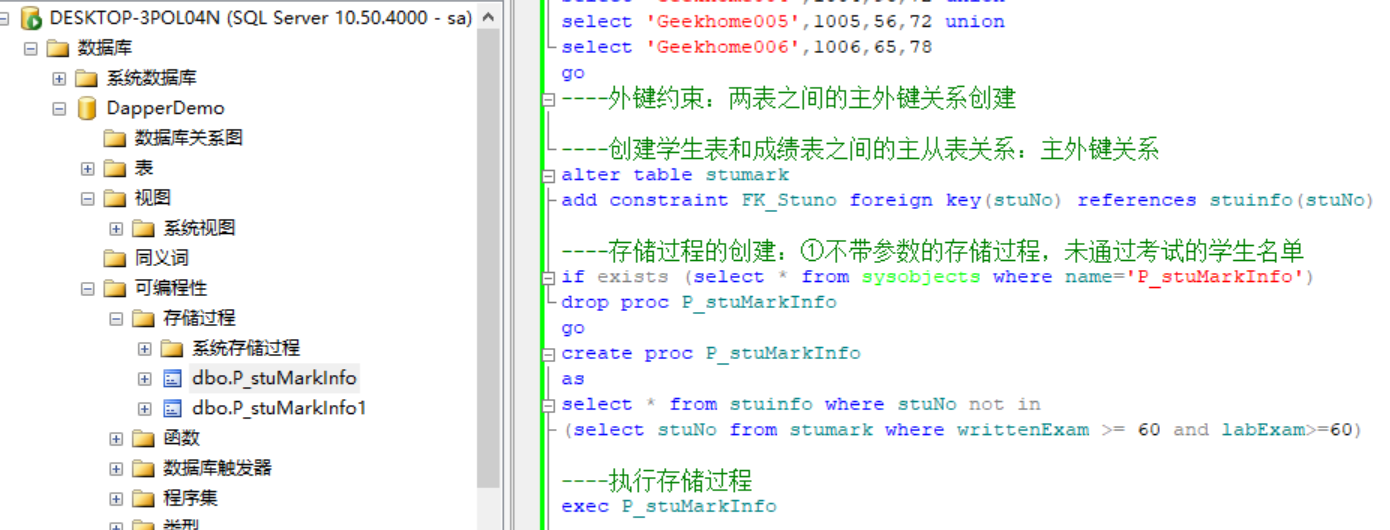
2.允许更快地执行，如果某操作需要大量SQL语句或重复执行对应的SQL语句，存储过程比SQL语句执行的更快。

3.减少网络流量，例如一个需要数百行的SQL代码的操作有一条执行语句完成，不需要在网络中发送数百行代码。

4.更好的安全机制，对于没有权限执行存储过程的用户，也可以授权它们执行存储过程。

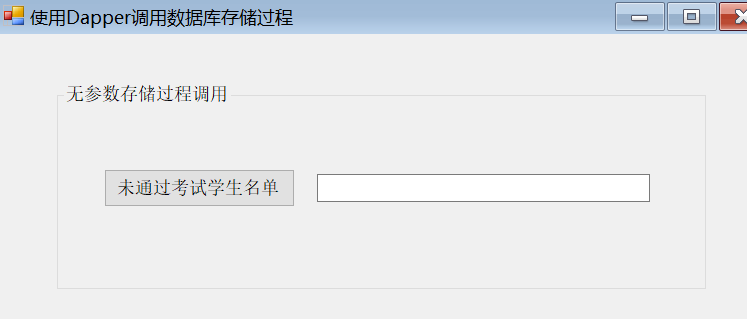
### 3.1.1.使用Dapper调用无参数的存储过程

这里我首先在数据库中创建了一个基于DapperDemo数据库的无参数的存储过程：dbo.P\_stuMarkInfo，具体的存储过程语句已上传至github：<https://github.com/devyf/Dapper-.git>



这里我基于wiform界面按钮操作来调用我的后台存储过程，

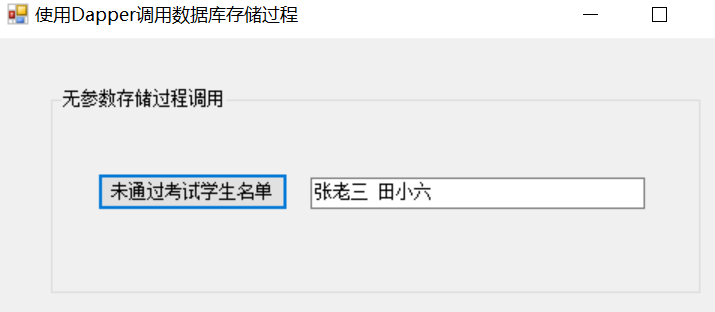
点击“未通过考试学生名单”按钮，会调用后台在数据库对应创建的”dbo. P\_stuMarkInfo”存储过程：



对应后台的无参存储过程点击调用执行代码：

|  |
| --- |
| /// <summary>  /// 点击按钮，查询未能通过考试的学生名单，调用无参数的存储过程，默认及格线为60分  /// </summary>  /// <param name="sender"></param>  /// <param name="e"></param>  private void button1\_Click(object sender, EventArgs e)  {  List<StuInfo> stuList = new List<StuInfo>();  using (IDbConnection db = new SqlConnection(DBHelper.ConnString))  {  stuList = db.Query<StuInfo>("dbo.P\_stuMarkInfo", //存储过程的名称  null, //存储过程的参数  null, //事务对象  true, //是否缓存  null, //获取或设置在终止执行命令的尝试并生成错误之间的等待时间  CommandType.StoredProcedure //指定的sql语句为存储过程类型  ).ToList();  }  if (stuList.Count > 0)  {  stuList.ForEach(stu => this.textBox1.Text += stu.StuName + " ");  }  } |

执行效果如下：



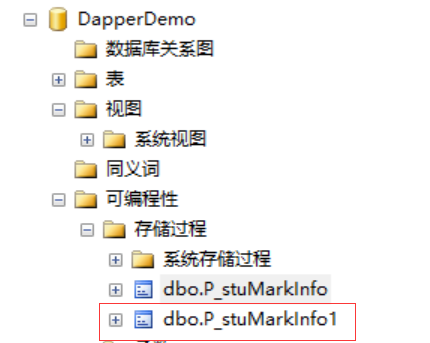
### 3.1.2.使用Dapper调用有参数（有返回值）的存储过程

这里我继续添加sql来创建了一个有参数、有返回值的存储过程，如下：

|  |
| --- |
| ----存储过程的创建：②带参数的存储过程  ----笔试和机试的及格线由用户指定，并且统计出未通过考试的人数  if exists (select \* from sysobjects where name='P\_stuMarkInfo1')  drop proc P\_stuMarkInfo1  go  create proc P\_stuMarkInfo1  @writeLevel int, ----输入参数：笔试及格线  @labLevel int, ----输入参数：机试及格线  @examNum int output ----输出参数：未通过考试的人数  as  select @examNum = count(\*) from stuinfo where stuNo not in  (select stuNo from stumark where writtenExam >= @writeLevel and labExam>=@labLevel)  ----执行存储过程  declare @countNum int  exec P\_stuMarkInfo1 60, 60, @countNum output  print '未通过考试的人数：' + convert(varchar(20), (@countNum)) |

存储过程语句均已上传至github：<https://github.com/devyf/Dapper-.git>

具体存储过程创建时需要对应在当前数据库下执行并保存，如下图：



这里对应前台winform界面上输入和输出参数设置如下，点击”未通过考试学生人数”按钮可以调用后台的存储过程：



后台调用有参数的存储过程代码如下：

|  |
| --- |
| /// <summary>  /// 点击按钮，通过设置的笔试成绩和机试成绩去动态调用带参数的存储过程  /// </summary>  /// <param name="sender"></param>  /// <param name="e"></param>  private void button2\_Click(object sender, EventArgs e)  {  //准备存储过程的三个参数：两个是输入参数，一个是输出参数  var param = new DynamicParameters(); //动态参数类  try  {  param.Add("@writeLevel", int.Parse(this.writeLev.Text)); //存储过程的输入参数赋值  param.Add("@labLevel", int.Parse(this.labLev.Text));  param.Add("@examNum", 0, DbType.Int32, ParameterDirection.Output); //标注为输出参数  }  catch (Exception ex)  {  MessageBox.Show(ex.Message);  }    using (IDbConnection db = new SqlConnection(DBHelper.ConnString))  {  db.Execute("dbo.P\_stuMarkInfo1", //指定存储过程名称  param, //存储过程参数  null, //存储过程事务  null, //执行等待时间  CommandType.StoredProcedure //指定执行为存储过程类型  );  //通过参数调用Get方法来获取返回值  int outNum = param.Get<int>("@examNum");  //放置到文本框中  this.nopassNum.Text = outNum.ToString();  MessageBox.Show("存储过程执行成功！");  }  } |

## 3.2.Dapper执行事务操作

事务的概念：在关系型数据库中，一个事务可以是一条SQL语句，一组SQL语句或者整个程序。

事务特性：

事务是恢复和并发控制的基本单位。

事务应该具有4个属性：原子性、一致性、隔离性、持久性。这四个属性通常称为ACID特性。

原子性（atomicity）：一个事务是一个不可分割的工作单位，事务中包括的所有操作要么都做，要么都不做。

一致性（consistency）：事务必须是使数据库从一个一致性状态变到另一个一致性状态。一致性与原子性是密切相关的。

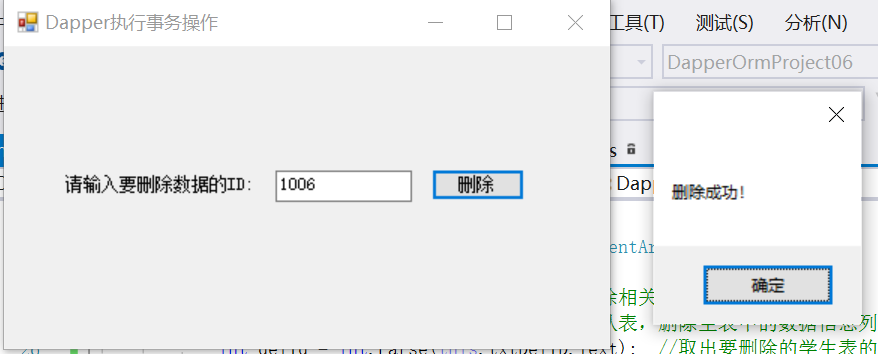
隔离性（isolation）：一个事务的执行不能被其它事务干扰。即一个事务内部的操作及使用的数据对并发的其它事务是隔离的，并发执行的各个事务之间不能互相干扰。

持久性（durability）：持久性也称为永久性（permanence），指一个事务一旦提交，它对数据库中数据的改变就应该是永久性的。接下来的其它操作或故障不应该对其有任何影响。

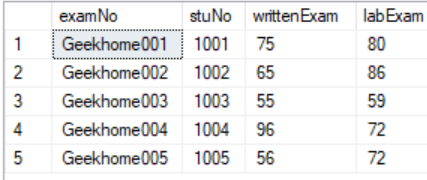
使用Dapper操作数据库事务，这里基于winform界面执行数据库中两张表（主表stuinfo、从表stumark）进行删除操作，使用Dapper操作事务进行提交、删除、回滚操作，具体后台代码如下：

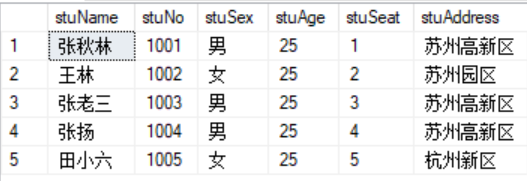
|  |
| --- |
| /// <summary>  /// 删除按钮点击执行事务操作  /// </summary>  /// <param name="sender"></param>  /// <param name="e"></param>  private void delBtn\_Click(object sender, EventArgs e)  {  //执行界面操作，根据主表学生表的ID去删除相关联的信息  //事务操作：根据输入的学生ID编号，删除从表，删除主表中的数据信息列  int delId = int.Parse(this.txtDelID.Text); //取出要删除的学生表的学号  using (IDbConnection db = new SqlConnection(DBHelper.ConnString))  {  db.Open(); //基于事务操作的特殊性：执行事务之前需要优先开启连接  //try catch语句使用外侧代码：快捷键：ctrl+k、ctrl+s  //创建事务对象  IDbTransaction transaction = db.BeginTransaction(); //开启数据库的事务  try  {  //根据用户输入的学号ID进行删除的操作：先删除从表，再删除主表的信息  string delSql1 = "delete from stuinfo where stuNo = @stuNo"; //主表  string delSql2 = "delete from stumark where stuNo = @stuNo"; //从表  //执行删除操作  db.Execute(delSql2, new { stuNo = delId }, transaction, null, null);  db.Execute(delSql1, new { stuNo = delId }, transaction, null, null);  //提交事务  transaction.Commit();  MessageBox.Show("删除成功！");  }  catch (Exception ex)  {  //出现异常，需要回滚事务  transaction.Rollback();  MessageBox.Show("出现异常：" + ex.Message);  }  finally  {  db.Close();  }  }  } |

执行删除从表、主表操作，成功从前台界面上删除ID为”1006”的数据：

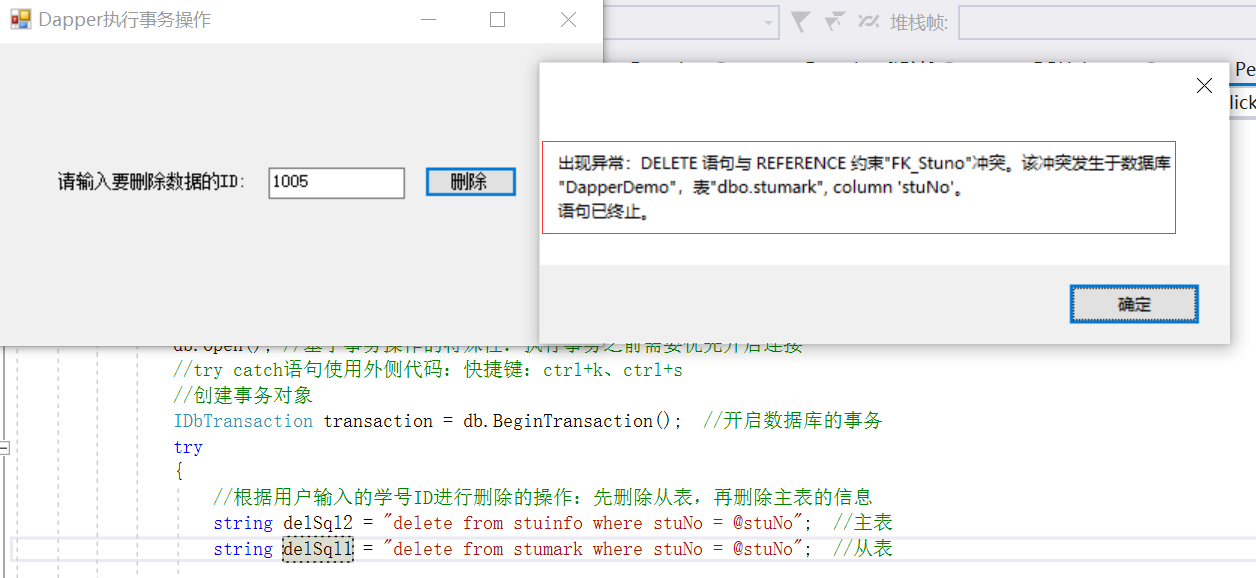


对应数据库中也随即删除：

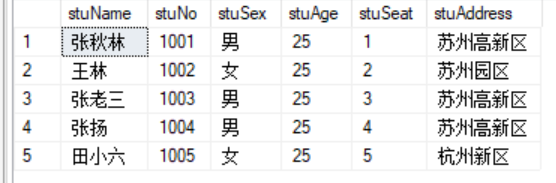




对应如果交换delSql1与delSql2的执行顺序，这时因为先删除主表，而在从表上有数据外键的关联，会执行数据回滚rollback操作，无法进行数据的删除操作：



可以看到catch异常之后，数据库中数据”1005”并未执行数据列的删除操作：



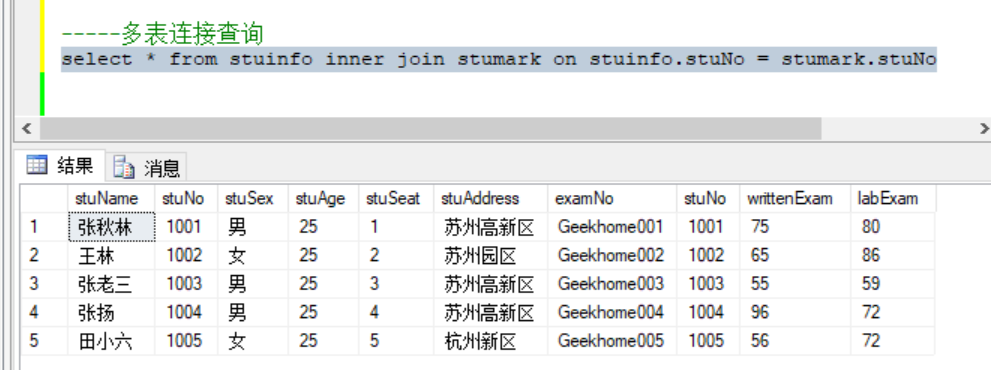
## 3.3.Dapper进行多表查询

Dapper框架可以基于数据库sql语句进行数据库字段映射，并使用splitOn进行类型与返回值之间的划分。

下面基于两个实例来简单介绍一下Dapper进行多表查询返回值划分案例：

实例一：

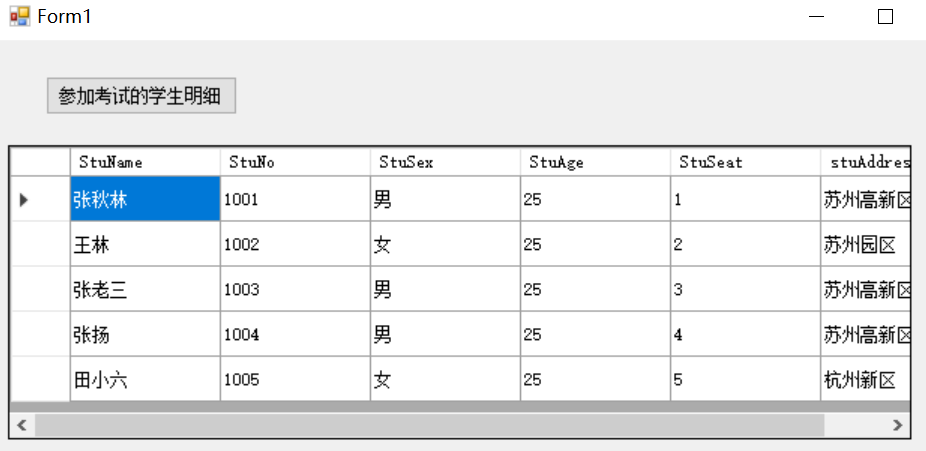
①首先基于之前的dbo.stuinfo表与dbo.stumark表进行多表连接查询，对应sql语句与查询结果如下：



②基于这个连接查询的结果，使用Dapper框架进行Query查询封装查询映射，具体代码如下：

|  |
| --- |
| private void button1\_Click(object sender, EventArgs e)  {  using (IDbConnection db = new SqlConnection(DBHelper.ConnString))  {  var sql = "select \* from stuinfo inner join stumark on stuinfo.stuNo = stumark.stuNo";  //执行查询：多表(类型一，类型二，返回值)  var list = db.Query<StuInfo, StuMark, StuInfo>(  sql,  (students, score) => { return students; }, //变量students对应的StuInfo类型，scores对应StuMark类型  null, //存储过程的参数  null, //事务  true, //缓存  splitOn: "stuNo" //该参数是用来划分查询中的字段是属于哪个表的 splitOn可以省略  );  /\*splitOn:stuNo 划分查询中的字段是属于哪个表的，也就是查询结构映射到哪个实体，上边的sql运行时，会从查询结果所有  字段列表的最后一个字段进行匹配，一直找到stuNo这个字段（大小写不计），找到的第一个stuNo字段匹配就是Query参数中  StuInfo类的stuNo属性，那么从stuNo到最后一个字段都属于StuInfo，StuNo以前的字段都被映射到StuMark这张表  通过(T,P)=>(return T)把两个类的实例解析出来\*/  this.dgvContent.DataSource = list;  }  } |

执行界面点击操作，加载数据库查询映射结果如下图所示：



实例二：

继续使用查询语句进行数据分割操作，这里同样创建了两个数据点击显示按钮界面进行查询操作：



查看数据后台代码：

|  |
| --- |
| private void button1\_Click(object sender, EventArgs e)  {  using (IDbConnection db = new SqlConnection(DBHelper.ConnString))  {  List<Customer> userList = new List<Customer>();  string sql = @"select u.\*, r.\* from UserInfo u  inner join UserRole ur on ur.UserId = u.UserId  inner join Role r on r.RoleId = ur.RoleId";  userList = db.Query<Customer, Role, Customer>( //第三个参数是返回值类型  sql,  (user, role) => { user.Role = role; return user; },  null,  null,  true,  "RoleId", //分割数据列字符串  null,  null  ).ToList();  this.dataGridView1.DataSource = userList;  //打印其中单个字符  if (userList.Count > 0)  {  userList.ForEach(item => Console.WriteLine("userName:" + item.Username + "passWord:" + item.PasswordHash +  "Role:" + item.Role));  }  }  }  private void button2\_Click(object sender, EventArgs e)  {  using (IDbConnection db = new SqlConnection(DBHelper.ConnString))  {  List<User> userList = new List<User>();  string sql = @"select u.UserId, u.UserName, u.PasswordHash, r.RoleId, r.RoleName from UserInfo u  inner join UserRole ur on ur.UserId = u.UserId  inner join Role r on r.RoleId = ur.RoleId";  Dictionary<int, User> dic = new Dictionary<int, User>();    userList = db.Query<User, Role, User>(  sql,  (user, role) =>  {  User tempUser;  if (!dic.TryGetValue(user.UserId, out tempUser))  {  tempUser = user;  dic.Add(user.UserId, tempUser);  }  tempUser.Role.Add(role);  return user;  },  null,  null,  true,  "RoleId",  null,  null  ).ToList();  this.dataGridView2.DataSource = userList;  //打印其中单个字符  if (userList.Count > 0)  {  userList.ForEach(item => Console.WriteLine("userName:" + item.Username + "passWord:" + item.PasswordHash +  "Role:" + item.Role.First().RoleName));  }  }  } |

查询结果如下图所示：

