# ado.net概述

## 什么是ado.net

程序要和数据库要通过ADO.Net进行。通过ADO.Net就能在程序中执行SQL了。

ADO.NET就是一组类库，这组类库可以让我们通过程序的方式访问数据库，就像System.IO下的类操作文件一样， System.Data.这组类是用来操作数据库(不光是MSSql Server)，它提供了统一的接口让操作其它数据库(Access、Oracle等)的方式和操作MSSql Server一致.

## ADO.NET组成

### 数据提供程序(常用类)

Connection：用来连接数据库

Command：用来执行SQL语句

DataReader：只读、只进的结果集，一条一条读取数据（StreamReader、XmlReader微软的类库中这些Reader的使用方式都差不多）

DataAdapter，一个封装了上面3个对象的对象

### 数据集(DataSet)

在内存中的临时数据库

断开式数据操作：数据库断开，数据仍然有效

### ADO.NET中的其他常见类

ConnectionStringBuilder//自动生成连接字符串

Parameter//带参数的SQL语句

Transaction//在ADO.NET中使用事务

与DataSet相关的类：

DataView//视图类，DataTable中的数据以不同的视角查看

DataRowView//DataView中的行。

DataTable //DataSet中的数据表

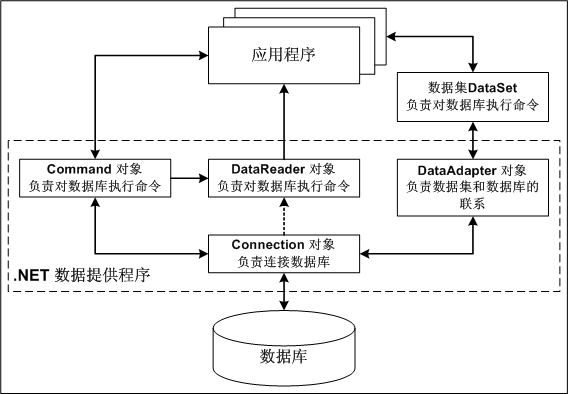
DataRow//DataTable中的行

DataColumn//DataTable中的列

DataRealation//DataTable与DataTable的关系

Constraint//DataTable中建立的约束

### 各类在开发中的职能



## Ado.net访问数据库的方式

* 方式一：
  + 1.连接数据用Connection对象
  + 2.执行SQL语句Command
  + 3.执行完毕之后将结果一条一条返回。DataReader
* 方式二：
  + 使用DataAdapter+DataSet,这种方法本质还是通过Connection、Command、DataReader将数据全部取出来然后放到了DataSet中

# connection类连接数据库

## connection操作数据库的过程

1. 构造连接字符串
2. 创建connection对象
3. 打开连接
4. 执行SQL语句
5. 关闭连接（因为connection对象为非托管资源，所以执行完毕后，需要手动关闭和销毁连接）
6. 释放资源

### 构造连接字符串

|  |
| --- |
| //1.构造连接字符串  *SqlConnectionStringBuilder* connStr = new *SqlConnectionStringBuilder*();  connStr.*DataSource* = "(local)";//SQLserver实例所在的网址或名称  connStr.*InitialCatalog* = "ItCastCn";//要连接的数据库名称  connStr.*UserID* = "sa";//连接数据库的用户ID  connStr.*Password* = "123";//连接数据库的用户密码 |

### 创建Connection对象

|  |
| --- |
| //2. 创建connection对象  SqlConnection conn = new SqlConnection();  //2.1 传入连接字符串。连接数据库  conn.ConnectionString = connStr.*ConnectionString*; |

SqlConnetction类有一个重载的构造函数，可以直接以字符串形式传入连接字符串，不需要提前用SqlConnectionStringBuilder构造连接字符串。

|  |
| --- |
| //2. 创建connection对象,并连接数据库。  SqlConnection conn = new SqlConnection("Data Source=(local);Initial Catalog=ItCastCn;User ID=sa;Password=123"); |

### 打开连接

|  |
| --- |
| //3. 打开连接  conn.Open(); |

### 执行SQL语句

表示向服务器提交的命令。需要通过SqlCommand对象执行。

例子：

|  |
| --- |
| //4.执行SQl语句  *SqlCommand* cmd = new *SqlCommand*(); //4.1 创建SqlCommand对象  cmd.*Connection* = conn;//4.2 SqlCommand要操作的connection对象。  cmd.*CommandText* = "insert into tblClass values ('.net', '激情澎湃的班级')";//4.3 要执行的SQL命令  cmd.*CommandType* = *CommandType*.*Text*;//4.4命令的类型为sql语句，默认就是CommandType.Text可以省略  int i = cmd.*ExecuteNonQuery*();//4.5执行命令。  if(i>0)  {  *Console*.*WriteLine*("插入成功");  }else  {  *Console*.*WriteLine*("插入失败");  } |

#### 创建SqlCommand对象

创建方式有2种：

1. 通过new关键字

|  |
| --- |
| *SqlCommand* cmd = new *SqlCommand*(); //4.1 创建SqlCommand对象  cmd.*Connection* = conn;//4.2 SqlCommand要操作的connection对象。  cmd.*CommandText* = "insert into tblClass values ('"+classID+"', '"+desc+"')"; |

或者

|  |
| --- |
| string sql = "select tClassDesc from tblClass where tClassName = '" + classId + "'";  //在构造函数中传入sql语句与connection对象  *SqlCommand* cmd = new *SqlCommand*(sql, conn); |

1. 通过SqlConnection对象的CreateCommand方法创建。

|  |
| --- |
| SqlCommand cmd = conn.CreateCommand();  //用输入的username过滤查询结果  cmd.CommandText = "select \* from [Table] where Name = '" + username + "'"; |

#### 常用的三个方法

ExecuteNonQuery()：用来对数据库进行增、删、改，返回受影响的行数。适合insert、upadte、delete（对于其他语句返回为-1）.

ExecuteScalar():执行查询，返回首行首列

ExecuteReader():执行查询，返回DataReader对象。

#### 常用的属性

CommandText:要执行的SQL语句，也可以是存储过程名。

CommandType：指定命令的类型，是SQL语句还是存储过程。默认为SQL语句。

### 关闭与释放资源

## 案例：完整的数据库操作过程

|  |
| --- |
| using System;  using System.*Collections*.*Generic*;  using System.Data;  using System.Data.SqlClient;  using System.*Linq*;  using System.*Text*;  using System.*Threading*.*Tasks*;  namespace \_01\_连接数据库  {  class Program  {  static void Main(string[] args)  {  ////1.构造连接字符串  //SqlConnectionStringBuilder connStr = new SqlConnectionStringBuilder();  //connStr.DataSource = "(local)";//SQLserver实例所在的网址或名称  //connStr.InitialCatalog = "ItCastCn";//要连接的数据库名称  //connStr.UserID = "sa";//连接数据库的用户ID  //connStr.Password = "123";//连接数据库的用户密码  //2. 创建connection对象,并连接数据库  SqlConnection conn = new SqlConnection("Data Source=(local);Initial Catalog=ItCastCn;User ID=sa;Password=123");  //2.1 传入连接字符串。连接数据库  // conn.ConnectionString = connStr.ConnectionString  //3. 打开连接  conn.Open();  //输出当前连接状态  *Console*.*WriteLine*(conn.State);  //判断当前连接状态，如果为open则执行。  //4.执行SQl语句  *SqlCommand* cmd = new *SqlCommand*(); //4.1 创建SqlCommand对象  cmd.*Connection* = conn;//4.2 SqlCommand要操作的connection对象。  cmd.*CommandText* = "insert into tblClass values ('.net', '激情澎湃的班级')";//4.3 要执行的SQL命令  cmd.*CommandType* = *CommandType*.*Text*;//4.4命令的类型为sql语句，默认就是CommandType.Text可以省略  int i = cmd.*ExecuteNonQuery*();//4.5执行命令。  if(i>0)  {  *Console*.*WriteLine*("插入成功");  }else  {  *Console*.*WriteLine*("插入失败");  }  //5. 关闭连接  conn.Close();  *Console*.*WriteLine*(conn.State);  //6. 释放资源  conn.Dispose();  *Console*.*WriteLine*(conn.State);  *Console*.*ReadLine*();  }  }  } |

# SQLCommand对象详解

SqlCommand类提供用户一个SQL操作接口,使得用户可以完成设置SQL命令、执行查询操作、获取查询结果等工作。

## 以交互方式执行Sql语句

在《2.2案例：完整的数据库操作过程》中，执行SQL语句时，直接把要添加的数据写入CommandText中，对用户来讲，不可能直接在源代码中插入数据，需要直接在界面中进行数据的操作。这就需要用交互式方式进行数据的操作。

有两种方式：

字符串拼接和参数化查询

### 字符串拼接

例如，在控制台中输入班级名称和班级描述，进行数据库数据插入的操作：

|  |
| --- |
| using System;  using System.*Collections*.*Generic*;  using System.Data;  using System.Data.SqlClient;  using System.*Linq*;  using System.*Text*;  using System.*Threading*.*Tasks*;  namespace \_01\_连接数据库  {  class Program  {  static void Main(string[] args)  {  *Console*.*WriteLine*("输入班级名称：");  string classID= *Console*.*ReadLine*();  *Console*.*WriteLine*("输入描述：");  string desc = *Console*.*ReadLine*();  SqlConnection conn = new SqlConnection("Data Source=(local);Initial Catalog=ItCastCn;User ID=sa;Password=123");  conn.Open();  SqlCommand cmd = new SqlCommand(); //4.1 创建SqlCommand对象  cmd.Connection = conn;//4.2 SqlCommand要操作的connection对象。  cmd.CommandText = "insert into tblClass values ('"+classID+"', '"+desc+"')";  cmd.CommandType = CommandType.*Text*;  int i = cmd.ExecuteNonQuery();  if(i>0)  {  *Console*.*WriteLine*("插入成功！");  }  conn.Close();  conn.Dispose();  *Console*.*ReadLine*();  }  }  } |

定义两个变量接收用户的输入：

|  |
| --- |
| *Console*.*WriteLine*("输入班级名称：");  string classID= *Console*.*ReadLine*();  *Console*.*WriteLine*("输入描述：");  string desc = *Console*.*ReadLine*(); |

在commandText中，把要输入的值替换为变量即可。也就是替换为：”+变量+”

|  |
| --- |
| cmd.CommandText = "insert into tblClass values ('"+classID+"', '"+desc+"')";  // cmd.CommandText = "insert into tblClass values ('值1', '值2')"; |

### 参数化查询

详见第 6 章参数化查询

## ExecuteNonQuery

返回受影响的行数，返回类型为int

|  |
| --- |
| cmd.CommandText = "select Name from T\_Students";  Console.WriteLine(cmd.ExecuteNonQuery());//执行SQL语句并返回受到Insert,Update 和 Delete 操作影响的行数。对于所有其他查询都返回 –1 |
| 输出结果为：-1 |

## ExecuteScalar

返回结果集第一行第一列，返回类型为object。

|  |
| --- |
| *Console*.*WriteLine*("输入班级名称：");  string classId = *Console*.*ReadLine*();  *SqlConnection* conn = new *SqlConnection*("Data Source=(local); Initial Catalog=ItCastCn; User ID=sa; Password=123");  conn.*Open*();  //sql语句  string sql = "select tClassDesc from tblClass where tClassName = '" + classId + "'";  //在构造函数中传入sql语句与connection对象  *SqlCommand* cmd = new *SqlCommand*(sql, conn);  //返回结果集第一行第一列  string desc = cmd.*ExecuteScalar*().*ToString*();  *Console*.*WriteLine*(desc);  conn.*Close*();  conn.*Dispose*();  *Console*.*ReadLine*(); |

不仅能用在select还能用在insert

|  |
| --- |
| //output防止并发。一般用在自动增长主键。这里为了方便没有用自动增长。  cmd.CommandText = "insert into T\_Students(Id,Name,Age) output inserted.Id values(5,'西瓜皮',18)";  int id = (int)cmd.ExecuteScalar();//object转换为int  Console.WriteLine("插入的Id为"+id); |

## ExecuteReader

返回一个SqlDataReader对象（详细用法见第 4 章），该对象提供一种从 SQL Server 数据库读取行的只进流的方式。

# SqlDataReader对象详解

SqlDataReader提供一种从SQL Server 数据库读取行的只进流的方式。

## read()

reade方法完成一行读取，如果连接到数据则返回true，否则返回false。当一行读取完毕后，它会指向下一行。

## 读取行中某列的数据

### 使用索引方式读取

这里的索引，不是根据表的结构来确定的，而是根据ExecuteReader返回的结果集决定的。索引是从0开始的。

使用索引返回的结果时，返回值为object类型，需要类型转换。

|  |
| --- |
| string tblclass = (string)reader[0];  string tblclass1 = (string)reader["tClassName"];  string tblclass2 = (string)reader[reader.GetOrdinal("tClassName")]; |

### 使用方法读取

SqlDataReader提供了很多方法来获取某列的值。用方法获取值不需要进行类型转换。

例如，字段的值为字符串，用GetString方法，如果为整数类型，则可用GetInt32类型。浮点数、布尔、字符等类型都有相应的方法。

|  |
| --- |
| string tblclass = reader.GetString(reader.GetOrdinal("tClassName"));  string tblclass1 = reader.GetString(0); |

## 案例：读取班级名称到comobox中

在数据库中每一行的数据是对tblClass实体的一个对象的属性的描述。因此我们也构建一个类，并用数据库中的数据来初始化对象。最后把对象绑定到列表中即可。

首先创建一个MyClass类来描述班级。

|  |
| --- |
| class MyClass  {  public string tClassName { get; set; }  public string tClassDesc { get; set; }  } |

之后从数据库中读取数据，并用这些数据初始化MyClass对象，最后把MyClass对象绑定到comobox上，并显示其tClassName属性的值。

|  |
| --- |
| using System;  using System.*Collections*.*Generic*;  using System.*ComponentModel*;  using System.Data;  using System.Data.SqlClient;  using System.*Drawing*;  using System.*Linq*;  using System.*Text*;  using System.*Threading*.*Tasks*;  using System.*Windows*.*Forms*;  namespace \_03\_ExecuteReader  {  public partial class Form1 : *Form*  {  public Form1()  {  InitializeComponent();  }  private void button1\_Click(object sender, *EventArgs* e)  {  using (*SqlConnection* conn = new *SqlConnection*("Data Source=(local); Initial Catalog=ItCastCn; User ID=sa; Password=123"))  {  if (conn.*State* == *ConnectionState*.*Open*)  {  return;  }  else  {  conn.*Open*();  string sql = "select tClassName,tClassDesc from tblClass";  using (SqlCommand cmd = new SqlCommand(sql, conn))  {  using (SqlDataReader reader = cmd.ExecuteReader())  {  //清空comboBox1，也就是初始化  comboBox1.*Items*.*Clear*();  while (reader.Read())  { //获取tClassName字段的值  string className = reader.GetString(reader.GetOrdinal("tClassName"));  //获取tClassDesc字段的值  string classDesc = reader.GetString(reader.GetOrdinal("tClassDesc"));  //创建一个tblclass对象  MyClass tblclass = new MyClass();  //用数据库中获得的值来初始化对象的状态。  tblclass.tClassName = className;  tblclass.tClassDesc = classDesc;  //把tblclass对象添加到comobox中  comboBox1.*Items*.*Add*(tblclass);  //comboBox中显示tblclass对象的tClassName属性的值。  comboBox1.*DisplayMember* = "tClassName";  }  }  }  }  }  }  }  } |

案例：绑定班级名称、描述到DataGridView中。

方法与上一个案例类似，区别在于DataGridView与comobox添加数据的方式不同

|  |
| --- |
| private void Form1\_Load(object sender, *EventArgs* e)  {  using (*SqlConnection* conn = new *SqlConnection*("Data Source =(local);Initial Catalog=ItCastCn;User Id=sa;Password=123"))  {  conn.*Open*();  using (*SqlCommand* cmd = conn.*CreateCommand*())  {  string sql = "select tClassName,tClassDesc from tblClass";  cmd.*CommandText* = sql;  using (*SqlDataReader* reader = cmd.*ExecuteReader*())  { //初始化控件  dataGridView1.*Columns*.*Clear*();  dataGridView1.*Rows*.*Clear*();  //添加列名称  dataGridView1.*Columns*.*Add*("name", "名称");  dataGridView1.*Columns*.*Add*("desc", "描述");  while (reader.*Read*())  {  string ClassName = reader.*GetString*(reader.*GetOrdinal*("tClassName"));  string ClassDesc = reader.*GetString*(reader.*GetOrdinal*("tClassDesc"));  MyClass mc = new MyClass(){Name=ClassName,Desc=ClassDesc};  //把数据库中的值添加到行中  dataGridView1.*Rows*.*Add*(mc.Name,mc.Desc);    }  }  }  }  } |

# 连接池

ADO.NET中为了提高性能，所以使用了连接池，这样每个请求就不必都创建一个连接，然后认证，然后执行SQL，而是从连接池中直接取出连接执行SQL，执行完成后也并不是真正关闭连接，而是将该连接重新放回连接池中。如果有100个客户端，每个客户端在使用一段时间后连接池中保存了10个连接，那么在这种情况下，即使不在客户端做任何操作，SQL Server上都有1000个连接，这样不出性能问题才怪。

既然是ADO.NET连接池的问题，针对该问题的2个解决办法：

1.关闭ADO.NET的连接池，每次执行SQL时都是新建一个连接执行，然后关闭。这样做将使数据查询有所减慢(每次都建立连接，每次都认证，当然会慢了)，不过这个慢是毫秒级的，一般感觉不到的，但是如果一个操作就涉及到几百个SQL语句的情况可能会明细感觉到减慢。修改方法特别简单，都不用修改代码，在数据库链接字符串中加入Pooling=False;即可。

2.修改架构，这种CS架构除了性能问题外还会出现其他的比如安全上的问题。可以将直接连数据库的方法改成连接服务，这其中可以使用Remoting、Web服务等，当然现在可以统一用WCF了。这样做就只有服务程序去连接数据库，而客户端只连接服务程序，这样就不会出现连接池造成的瓶颈。不过这样做代码修改量很大，若真要改还是很痛苦的。

介绍ADO.NET连接池

连接池允许应用程序从连接池中获得一个连接并使用这个连接，而不需要为每一个连接请求重新建立一个连接。一旦一个新的连接被创建并且放置在连接池中，应用程序就可以重复使用这个连接而不必实施整个数据库连接创建过程。

当应用程序请求一个连接时，连接池为该应用程序分配一个连接而不是重新建立一个连接;当应用程序使用完连接后，该连接被归还给连接池而不是直接释放。

如何实现ADO.NET连接池

确保你每一次的连接使用相同的连接字符串(和连接池相同);只有连接字符串相同时连接池才会工作。如果连接字符串不相同，应用程序就不会使用连接池而是创建一个新的连接。

优点

使用连接池的最主要的优点是性能。创建一个新的数据库连接所耗费的时间主要取决于网络的速度以及应用程序和数据库服务器的(网络)距离，而且这个过程通常是一个很耗时的过程。而采用数据库连接池后，数据库连接请求可以直接通过连接池满足而不需要为该请求重新连接、认证到数据库服务器，这样就节省了时间。

缺点

数据库连接池中可能存在着多个没有被使用的连接一直连接着数据库(这意味着资源的浪费)。

技巧和提示

1. 当你需要数据库连接时才去创建连接池，而不是提前建立。一旦你使用完连接立即关闭它，不要等到垃圾收集器来处理它。

2. 在关闭数据库连接前确保关闭了所有用户定义的事务。

3. 不要关闭数据库中所有的连接，至少保证连接池中有一个连接可用。如果内存和其他资源是你必须首先考虑的问题，可以关闭所有的连接，然后在下一个请求到来时创建连接池。

## （补充）关于.NET大数据量大并发量的数据连接池管理

首先声明，本人以前做的都是小应用，还没有设计过百万、千万数据量的应用，并发量上万的应用，很多时候都是Applicatiion Server和DB Server在一台机器，或者将DB Server分离出来，单独在一台机器。由于最近的工作需要，故产生很多疑问，想要和大家一起探讨，向大家学习，希望大家不吝赐教，感谢大家的参与。

    首先提一下本文的需求来源和具体的需求。

    需求的来源：

    本文的需求产生于公司最近的产品，是一个互联网应用，考虑到将来的数据量会很大，大概单表上亿条记录，并发量可能也会很大，大概十万、甚至百万的并发访问。

    技术选型：

    数据库：MS SQL Server 2005

    平台选择：ASP.NET 3.5 SP1

    开发工具：VS2008 SP1

    其他类库或框架：JQuery 、NHibernate、Entity Framework

    需求：

    1、数据库架构

    大数据量在数据库的存储问题，单表存放的话，上亿条数据是否太多，有的数据经常使用，有的数据常年不用，是否考虑要分两张表（不过找高人问了一下，自己也找了一些资料看了一下，好像有一种叫做“分区表”，好像就是用来解决类似问题的，具体可以参考微软的官方资料：<http://msdn.microsoft.com/zh-cn/library/ms345146(SQL.90).aspx>）。

    2、应用架构

    这回肯定要将Application Server和DB Server分开，而且数据库可能还要集群，Application Server可能还要做NLB(Network Load Balancing 网络负载均衡)管理。而且在数据库连接方面如何设计更好呢？是否应该使用连接池？连接池如何管理呢？保持固定连接数量，然后做好分配工作？还是每次连接都重新打开，然后关闭，只在服务器IIS和数据库做好设置就可以了呢？

    3、ORM工具选择

    基本选定两种：微软的Entity Framework和NHibernate，不知道那种更合适呢？微软的东西肯定和自己的软硬件搭配更好，可是EF是新东西，不知道应用在我上面提到的大型应用中是否合适？大家请给点提示！！！NHibernate是否更成熟呢？毕竟有Hibernate的多年经验做支持，那个更好呢，希望大家给点提示！！！！

    下面我将给出我的应用架构设计的一些想法，不成熟，不正确的地方希望大家给予指点，稍后我会将大家的补充和指在再写成一篇blog，以供大家参考。

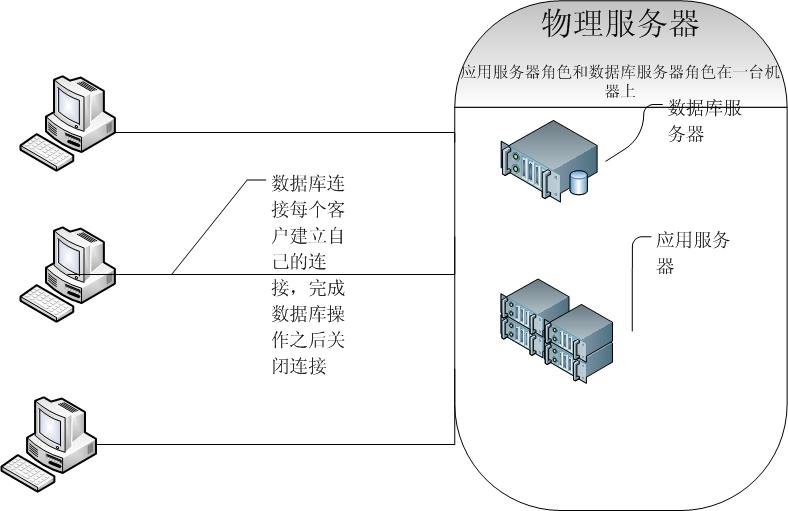
    下面的应用分类只是我自己的一些理解，不合理的地方，也请大家指出来。

    1、小型应用

    这时候Application Server和DB Server在一起服务器上，两个角色和在一起，这种适用于访问量较小，数据量也较小，并发量也较小，甚至没有并发。

    每次的数据连接，是需要访问数据库就打开，用完就关闭，当然了，要设置IIS的并发量和数据库的并发量，还有web.config中的数据库连接池的设置。

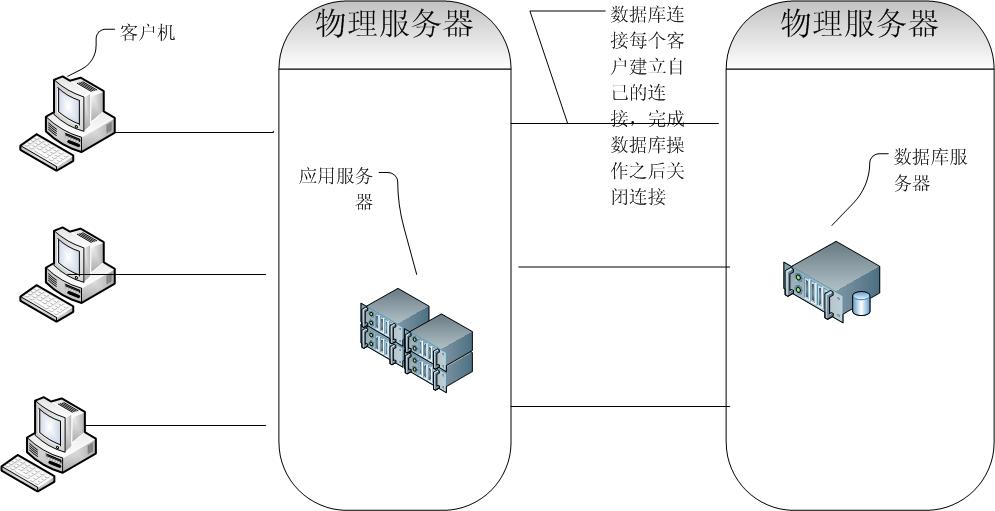
<add name="DefaultConnectionString" connectionString="Data Source=192.168.1.2;Initial Catalog=abc;User ID=sa;Password=sa;Enlist=true;Pooling=true;Max Pool Size=500;Min Pool Size=0"/>



     2、中型应用

    这时候就需要将Application Server和DB  Server分开在两台机器了，可以使得数据库可以充分享受各种IO资源，大内存、大IO，使得数据库发挥最大功效，可以在一定程度提升很多的性能。这种适用于访问量开始增大，数据量也表较大，并发也需要考虑了，可能会上几十、几百之类的。这时候，并发数量的各种设置也是必须的，具体的数值是要根据测试才可以合理的。

    这时候的数据库连接可能还是可以采取每次打开，然后关闭的方式。



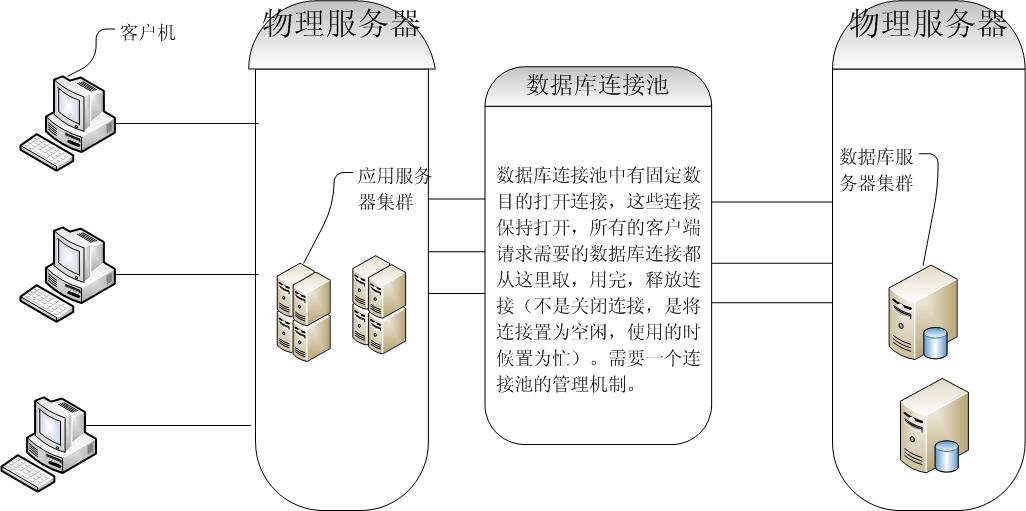
    3、大型设置更大型的应用

    这时候的Application Server和DB Server都不只是一台可以解决的了，Application Server和DB Server都需要做NLB才可以解决的。因为，这时候的数据量可能会上亿，数据库的大小会TB级别，并发量会上万、十万、百万、甚至千万级别。

    这时候的数据库连接就需要仔细考虑了，有两个选择。

* 一种还是前面的办法，设置好各个地方的并发连接量，然后每次还是重新连接，用完关闭，如果超出并发数的访问，就等待一下，等待其他连接关闭。

* 一种就是需要一个连接池，在连接池中保持打开固定数目的连接，如果有需要访问数据库的，就从连接池中找一个没有被使用的连接，给连接的请求者，我考虑，就是在application start的时候就打开固定数目的连接，这些连接需要一个管理机制，因为要标识连接是否空闲，要设置连接的状态，是否正在使用中，但是这些连接不用释放，保持打开，知道应用关闭，连接自动关闭。如果打开的连接被使用完毕，同时又有请求，那就还是要等待。 不知道NHibernate在这方面有没有提供管理共享连接的支持呢？希望知道的老兄可以指点一二！！



    说实话，这两种那个更好，我也说不清楚，还请大家多指教了！！！

　　稍后我还要补充几张图让大家可以更直观的讨论，感谢大家的回复。

Technorati Tags: [ASP.NET](http://technorati.com/tags/ASP.NET),[架构设计，数据库并发，并发量，数据库连接池，IIS](http://technorati.com/tags/æ¶æè®¾è®¡ï¼æ°æ®åºå¹¶åï¼å¹¶åéï¼æ°æ®åºè¿æ¥æ± ï¼IIS),[web.config，数据库设计，分区表，负载均衡，NLB](http://technorati.com/tags/web.configï¼æ°æ®åºè®¾è®¡ï¼ååºè¡¨ï¼è´è½½åè¡¡ï¼NLB),[Network Load Balancing](http://technorati.com/tags/Network+Load+Balancing)

　　2010-1-8下午 15：39补充

　　经过网友的指正，http并发不等同于sql并发，http并发是向应用服务器的请求，sql并发是向数据库服务器的请求。

 　　2010-1-8下午 15：39补充

　　1、并发还要考虑单位，是秒并发还是分钟并发.

　　2、重构也是一个过程，从小数据量考虑起先。参看：<http://www.cnblogs.com/wuxilin/archive/2007/07/17/820482.html>

　　2010-1-9凌晨 0：28补充

　　1、补充一些关于连接池的知识

　　我以前对.NET连接池的认识是错误的，原来以为在web.config中设置了连接池，每次发起的数据库连接也还是会是新的，每个sql请求就是一个连接，需要打开和关闭。因此就想设计一个连接池，然后保持固定的连接数，需要数据库连接就从连接池中取出来一个给请求用，用完毕就设置连接空闲，等待下次请求。这样看来是多余的，ADO.NET已经为我们提供这样的连接池管理，每个连接字符串产生一个连接池，设置了池的最大连接数，尽管你在DBConnection上使用close或者dispose，连接都没有被关闭，其实是返回到池，由池来管理这些连接，等待下次请求。除非连接池中的连接全部关闭，连接池才从内存中释放。多个不同的的连接字符串，产生多个连接池。

　　大家可以参看：

<http://developer.51cto.com/art/200907/140094.htm>

<http://www.xueit.com/html/2009-03/21_852_00.html>

<http://social.microsoft.com/Forums/zh-CN/adonetzhchs/thread/6cd6b103-46bb-49a4-8f61-2e3527628130>

<http://www.cnblogs.com/FallingAutumn/archive/2009/10/30/1593168.html>

<http://www.cnblogs.com/FallingAutumn/archive/2009/10/30/1593151.html>

<http://msdn.microsoft.com/zh-cn/library/aa175863(SQL.80).aspx>

<http://www.cnblogs.com/tuyile006/archive/2008/05/26/1207988.html>

<http://dotnet.itpub.net/post/20969/394416>

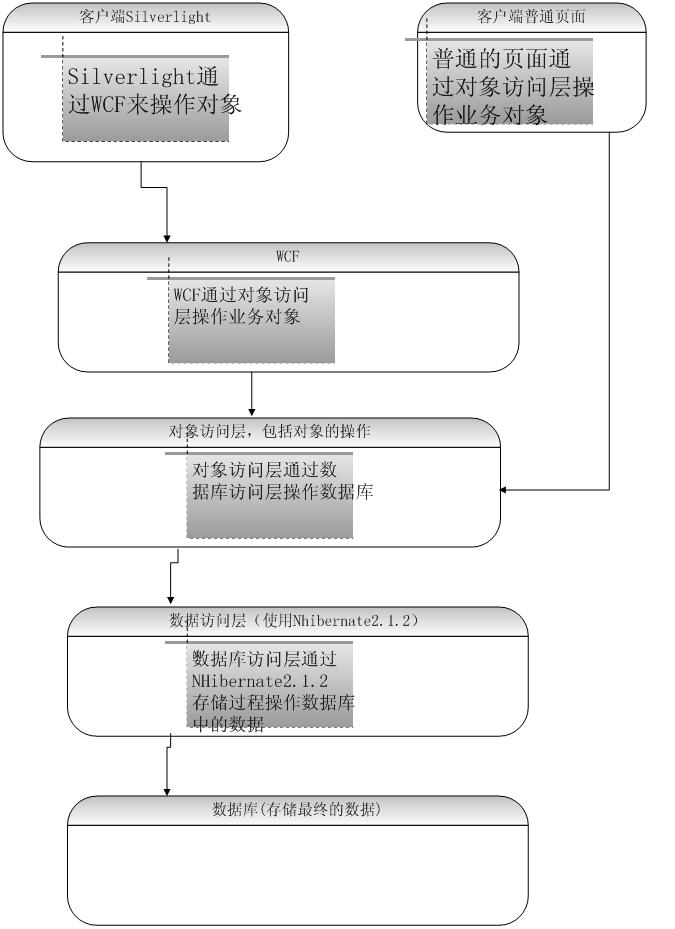
<http://blog.csdn.net/daremeself/archive/2009/04/26/4124886.aspx>

<http://hi.baidu.com/puwenbin/blog/item/d3aeba11d76b4b79ca80c4b8.html>

　　2010-1-20补充，应用架构图

这次的应用有一部分的客户端为了增加交互性，提高体验，使用了Silverlight3，数据库交互部分使用的是WCF，有一部分后台管理还是用ASP.NET页面来做，数据库交互就直接使用ADO.NET，下面是我的一张程序架构图，希望大家可以指出问题。

连接池机制



## 连接池机制

连接到数据库服务器通常需要一定的时间，且服务器也需要一定的资源来处理连接。Web应用程序有时处理成千上万的连接，需要相当多的资源来处理连接。ADO.NET具有连接池的特性，连接池的功能是保留一定数量的连接，当用户使用相同的连接字符串再次连接服务器时，ASO.NET将使用连接池中的连接而不用重新发起一次连接过程。当调用Close方法关闭连接时，ADO.NET将使用连接池中的连接而不用重新发起一次连接过程。当调用Close方法关闭连接时，连接将会返回到连接池中，下次再次调用Open方法时，将从连接池中取出一个连接使用。

（1）默认情况下，连接字符串中Pooling属性为True。如果使用连接池，在创建连接对象时，应该使用同一个连接字符串。ADO.net允许创建多个连接池，并且每个池都与不同的连接字符串关联，打开新连接时，如果连接字符串并非与现有池完全匹配，将创建一个新池。如果 MinPoolSize 在连接字符串中未指定或指定为零，池中的连接将在一段时间不活动后关闭。但是，如果指定的 MinPoolSize 大于零，在 AppDomain 被卸载并且进程结束之前，连接池不会被破坏。非活动或空池的维护只需要最少的系统开销。

　　（2）添加连接

　　连接池是为每个唯一的连接字符串创建的。当创建一个池后，将创建多个连接对象并将其添加到该池中，以满足最小池大小的要求。连接根据需要添加到池中，但是不能超过指定的最大池大小（默认值为 100）。连接在关闭或断开时释放回池中。连接池进程通过在连接释放回池中时重新分配连接，来满足这些连接请求。如果已达到最大池大小且不存在可用的连接，则该请求将会排队。然后，池进程尝试重新建立任何连接，直到到达超时时间（默认值为 15 秒）。如果池进程在连接超时之前无法满足请求，将引发异常。

　　（3）移除连接

　　如果连接长时间空闲，或池进程检测到与服务器的连接已断开，连接池进程会将该连接从池中移除。在使用完连接时一定要关闭连接，以便连接可以返回池。要关闭连接，可以使用 Connection 对象的 Close 或 Dispose 方法，也可以通过在 C# 的 using 语句中。

　　（4）清除池

　　ADO.NET 2.0 引入了两种新的方法来清除池：ClearAllPools 和 ClearPool。ClearAllPools 清除指定提供程序的连接池，ClearPool 清除与特定连接关联的ADO.NET连接池。

　　（5）事务支持

　　连接是根据事务上下文来从池中取出并进行分配的。除非在连接字符串中指定了 Enlist=false，否则ADO.NET连接池将确保连接在 Current 上下文中登记。如果连接使用登记的 System.Transactions 事务关闭并返回到池中，连接将保留在池中，以便使用相同 System.Transactions 事务对该连接池的下一次请求将返回相同的连接（如果可用）。如果发出这样的请求，而没有可用的池连接，则会从池的非事务性部分取出一个连接并登记。如果在池的每个区域都没有可用的连接，则会创建一个新的连接并登记。

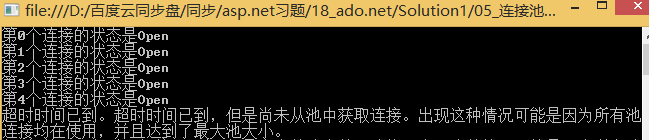
## 每个连接都占用连接池

即时是字符串相同的连接，只要没有关闭，当建立新连接时，会在连接池中增加连接池的size。只有关闭了的连接才会放到连接池中，下次连接时，不需要创建新连接，直接从连接池中取出即可 。

在下面的代码中，设置最大连接池数量为5，在循环中创建新连接，并打开连接。当执行时会发现，第6次执行打开连接的方法conn.Open()时，会抛出异常。这时因为每一个打开的连接都占用一个连接池的容量。

|  |
| --- |
| using *System*;  using *System*.*Collections*.*Generic*;  using *System*.*Data*.*SqlClient*;  using *System*.*Diagnostics*;  using *System*.*Linq*;  using *System*.*Text*;  using *System*.*Threading*.*Tasks*;  namespace \_05\_连接池  {  class Program  {  static void Main(string[] args)  {  *Stopwatch* watch = new *Stopwatch*();  watch.*Start*();  for (int i = 0; i < 1000; i++)  { //设置最大连接池为2。没有关闭连接的情况下，即时字符串完全相同，每次创建对象都会创建连接池  *SqlConnection* conn = new *SqlConnection*("Data Source=(local); Initial Catalog=ItCastCn;User Id=sa;Password=123; pooling= true;Max Pool Size=5;Min Pool Size=1; Connection TimeOut=3");  try  {  conn.*Open*();  *Console*.*WriteLine*("第{0}个连接的状态是{1}", i, conn.*State*.*ToString*());  //conn.Close();  }  catch (*Exception* e)  {  *Console*.*WriteLine*(e.*Message*);  }  finally  { //清空池  //SqlConnection.ClearAllPools();  }  }  watch.*Stop*();  *Console*.*WriteLine*(watch.*Elapsed*);  *Console*.*ReadLine*();  }  }  } |

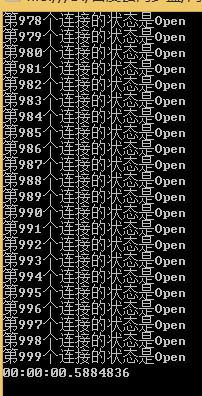
执行结果：



在代码中关闭连接：

|  |
| --- |
| try  {  conn.*Open*();  *Console*.*WriteLine*("第{0}个连接的状态是{1}", i, conn.*State*.*ToString*());  conn.*Close*();  } |

执行不会报错，而且执行时间很短



最后，关闭连接池测试一下速度：

|  |
| --- |
| *SqlConnection* conn = new *SqlConnection*("Data Source=(local); Initial Catalog=ItCastCn;User Id=sa;Password=123; pooling= false;Max Pool Size=5;Min Pool Size=1; Connection TimeOut=3"); |

执行结果：



效率满了6倍左右。

# 参数化查询

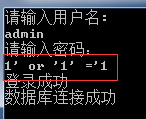
## 注入漏洞

注入漏洞是因为字符过滤不严谨所造成的，可以在帐号密码不正确的情况下进入程序。例如有一个帐号的名称与密码都是admin，利用漏洞，输入任意的用户名，密码用“1’ or ‘1’ = ‘1”（不包含“”）代替，即可成功登录。

代码如下：

|  |
| --- |
| cmd.CommandText = "select count(\*) from [Table] where Name = '" + username + "' and Password = '" + password + "'";    int number = (int)cmd.ExecuteScalar();  if(number>0)  {  Console.WriteLine("登录成功");  }  else  {  Console.WriteLine("帐号名或密码错误");  } |

在控制台中的输入：



这是因为上面的句子

|  |
| --- |
| "select count(\*) from [Table] where Name = '" + username + "' and Password = '" + password + "'"; |

提交到sql server变成：

|  |
| --- |
| Select count(\*) from [Table] where Name = ‘admin ’ and Password = ‘1’ or ‘1’ = ‘1’ |

为什么要多用一个or呢？解释一下，在逻辑运算符中，and的优先级别高于or, 所以会一直为真。

那么如何防止注入漏洞呢？那就是使用上一章的参数化查询。

参数化查询把输入作为参数传入值，这样即使输入 “1’ or ‘1’ = ‘1”，也会把它作为一个参数进行赋值，

### 案例：注入漏洞

|  |
| --- |
| using *System*;  using *System*.*Collections*.*Generic*;  using *System*.*Data*.*SqlClient*;  using *System*.*Linq*;  using *System*.*Text*;  using *System*.*Threading*.*Tasks*;  namespace \_05\_注入漏洞  {  class Program  {  static void Main(string[] args)  {  using (*SqlConnection* conn = new *SqlConnection*("Data Source=(local); Initial CataLog=MyFirstDB;User Id= sa; Password=123"))  {  conn.*Open*();  using (*SqlCommand* cmd = conn.*CreateCommand*())  {  try  {  string UTable = "create table Utable (id int identity(1,1) primary key not null, uid nvarchar(10) not null, password varchar(20) not null)";  cmd.*CommandText* = UTable;  int nums = cmd.*ExecuteNonQuery*();  }  catch (*Exception* e)  {  *Console*.*WriteLine*(e.*Message*);  }  string insertInfo = "insert into Utable(uid,password) values('hao','888')";  string selectUid = "select uid from Utable where uid='hao'";  cmd.*CommandText* = selectUid;  object obj = cmd.*ExecuteScalar*();  if (obj != null)  {  *Console*.*WriteLine*("用户已经存在");  }  else  {  cmd.*CommandText* = insertInfo;  int inums = cmd.*ExecuteNonQuery*();  if (inums > 0)  {  *Console*.*WriteLine*("插入成功");  }  }  *Console*.*WriteLine*("输入用户名：");  string uid = *Console*.*ReadLine*();  *Console*.*WriteLine*("输入密码");  string psd = *Console*.*ReadLine*();  string sqlInsertBug = "select \* from Utable where uid='" + uid + "' and password='" + psd + "'";  cmd.*CommandText* = sqlInsertBug;  using (*SqlDataReader* reader = cmd.*ExecuteReader*())  {  if (reader.*HasRows*)  {  *Console*.*WriteLine*("登录成功！");  }  else  {  *Console*.*WriteLine*("用户名或密码错误");  }  }  }  *Console*.*ReadLine*();  }  }  }  } |

随便输入一个用户名，密码用”1’ or 1=1 --”，登录成功。”1’ or 1=1 --”就是注入。



## 参数化查询

参数化查询（Parameterized Query 或 Parameterized Statement）是指在设计与[数据库](http://baike.baidu.com/view/1088.htm)链接并访问数据时，在需要填入数值或数据的地方，使用参数 (Parameter) 来给值，这个方法目前已被视为最有效可预防SQL注入攻击 (SQL Injection) 的攻击手法的防御方式。

参数化查询可以分为以下步骤：

* 用变量保存输入的值

|  |
| --- |
| Console.WriteLine("请输入用户名:");  string username = Console.ReadLine();  Console.WriteLine("请输入密码：");  string password = Console.ReadLine(); |

* 在SQL语句中使用参数：

参数格式为：@参数名

* 关联参数与变量

|  |
| --- |
| // 传值username，指定参数长度为20、类型为NVarChar  cmd.Parameters.Add(new SqlParameter("@Pname", System.Data.SqlDbType.NVarChar, 20) {Value = username });  // 传值password，不指定参数类型、长度  cmd.Parameters.Add(new SqlParameter("@PPassword", password)); |

完整的步骤如下：

|  |
| --- |
| cmd.CommandText = "select count(\*) from [Table] where Name = @Pname and Password = @PPassword";  cmd.Parameters.Add(new SqlParameter("@Pname", System.Data.SqlDbType.NVarChar, 20) {Value = username });  cmd.Parameters.Add(new SqlParameter("@PPassword", password));  int number = (int)cmd.ExecuteScalar();  if(number>0)  {  Console.WriteLine("登录成功");  }  else  {  Console.WriteLine("帐号名或密码错误");  } |

# DBNull与可空数据类型

Null，不是空，是“不知道”。数据库中int是可以为null的，但是C#中int不可以为null，存在一个不匹配的问题。

string、类等引用类型是可null的，但是int、bool、datetime等值类型是不可为null的，在不可为空的类型后加“？”就是可空类型了。

在C#中，如果值为null，直接写null的话用SqlParameter提交到服务器上是报错的（未提供值）。需要些DBNull.Value。

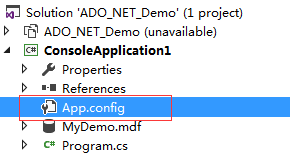
|  |
| --- |
| SqlParameter sp = new SqlParameter(“@id”,model.id==null?(object)DBNull.Value:model.id); |

# 配置文件

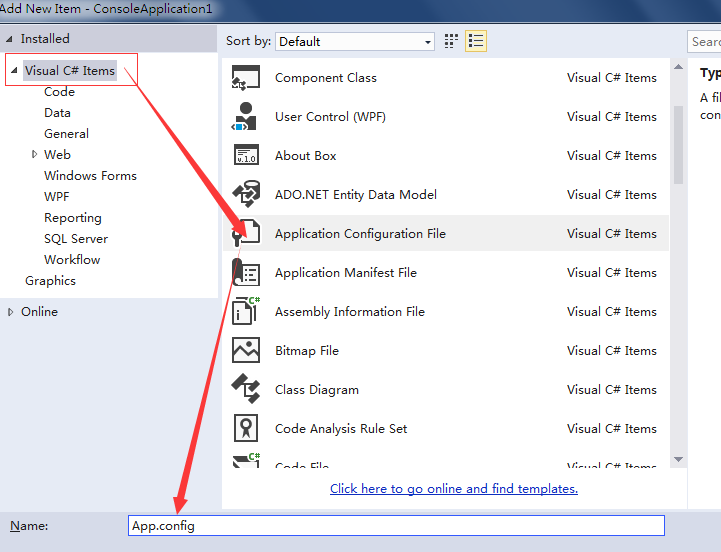
数据库连接的字符串一般是写在配置文件中的这样比较灵活.当数据库发生改变时,只需要修改配置文件而不必,修改程序,再重新编译了.

设置的步骤：

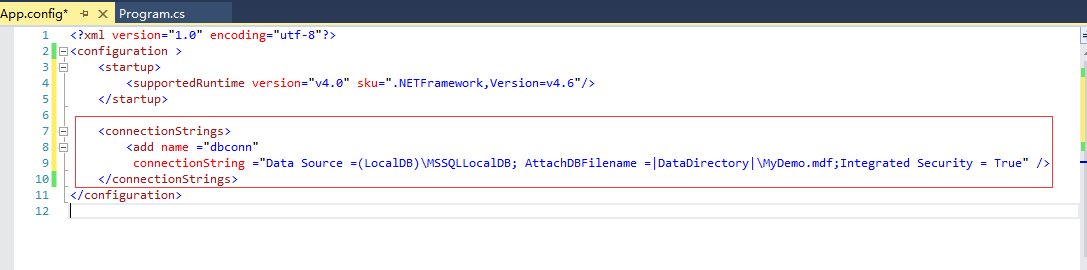
1. 首先要建立app.config(winForm应用程序)或者web.config(webForm应用程序)文件。一般会自动生成，



没有自动生成则在项目上右击“add→new item→Application Configuration File”,创建配置文件

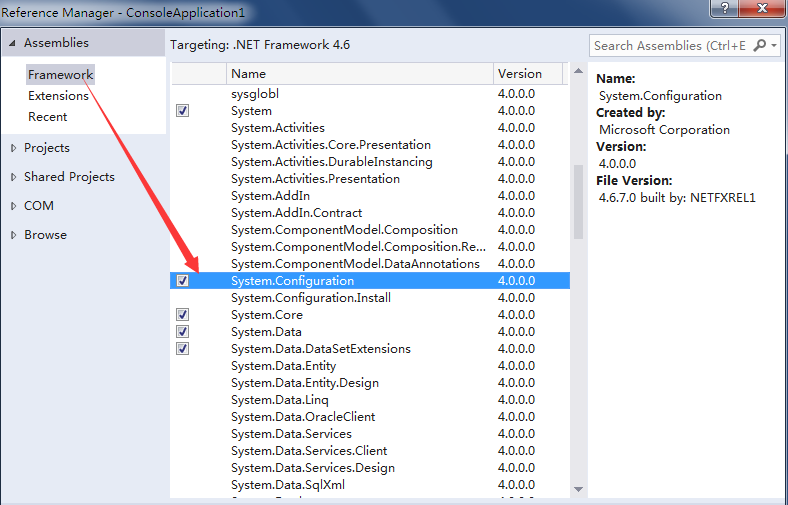


1. 在配置文件中设置



1. 添加System.Configuration程序集

在项目的Reference上右击，选择“Add Reference”，在弹出的Reference Manager中设置



1. 获取配置文件中的字符串

|  |
| --- |
| // 获得配置文件中的connectionString  string connStr = System.Configuration.ConfigurationManager.ConnectionStrings["dbconn"].ConnectionString; |

1. 修改数据库连接的字符串

|  |
| --- |
| //字符串连接数据库  SqlConnection conn = new SqlConnection(connStr) |

1. 在调整时，只需要修改配置文件即可



完整的代码如下：

|  |
| --- |
| using System;  using System.Collections.Generic;  using System.Data.SqlClient; // 连接数据库需要的命名空间  using System.Linq;  using System.Text;  using System.Threading.Tasks;  namespace ConsoleApplication1  {  class Program  {  static void Main(string[] args)  {  //下面的代码获取当前应用程序所在的目录  string dataDir = AppDomain.CurrentDomain.BaseDirectory;//取得当前程序的路径(即xxx.exe所在的文件夹)  if (dataDir.EndsWith(@"\bin\Debug\") || dataDir.EndsWith(@"\bin\Release\"))  {  dataDir = System.IO.Directory.GetParent(dataDir).Parent.Parent.FullName;//获取当前应用程序域的路径上一层再上一层的文件夹路径  AppDomain.CurrentDomain.SetData("DataDirectory", dataDir);//为指定的应用程序域的DataDirectory属性分配指定值。 以后可以用GetData取到值  }  // 获得配置文件中的connectionString  string connStr = System.Configuration.ConfigurationManager.ConnectionStrings["dbconn"].ConnectionString;  using  (  //字符串连接数据库  SqlConnection conn = new SqlConnection(connStr)  )  {  conn.Open();  using  (  //表示要对SQLServer执行SQL语句  SqlCommand cmd = conn.CreateCommand()  )  {  //要执行的SQL语句  cmd.CommandText = "insert into T\_Students(Id,Name,Age) values (4,'hyfdb', 20) ";  cmd.ExecuteNonQuery();//执行SQL语句并且返回受到Insert,Update 和 Delete 操作影响的行数。对于所有其他查询都返回 –1，；  Console.WriteLine("插入成功");  }  }  Console.WriteLine("数据库连接成功");  Console.ReadKey();  }  }  } |

## 案例：下拉列表联动

首先创建OrgStruc类来接受数据库中的字段

|  |
| --- |
| class OrgStruc  {  public int Oid { get; set; }  public string ODepartment { get; set; }  public int OParent { get; set; }  } |

创建绑定办法

|  |
| --- |
| private void BindDepartment(int connId, *ComboBox* cb)  {  cb.*Items*.*Clear*();  cb.*Items*.*Add*("--请选择--");  cb.*SelectedIndex* = 0;  string connStr = System.*Configuration*.*ConfigurationManager*.*ConnectionStrings*["UnivecityBind"].*ConnectionString*;  using (*SqlConnection* conn = new *SqlConnection*(connStr))  {  using (*SqlCommand* cmd = conn.*CreateCommand*())  {  cmd.*CommandText* = "select \* from OrgStructure where OrgParent=@id";  cmd.*Parameters*.*Add*(new SqlParameter("@id", connId));  conn.*Open*();  using (*SqlDataReader* reader = cmd.*ExecuteReader*())  {  while (reader.*Read*())  {  OrgStruc orgs = new OrgStruc()  {  Oid = reader.*GetInt32*(reader.*GetOrdinal*("Orgid")),  ODepartment = reader.*GetString*(reader.*GetOrdinal*("OrgDepartment")),  OParent = reader.*GetInt32*(reader.*GetOrdinal*("OrgParent"))  };  cb.*Items*.*Add*(orgs);  cb.*DisplayMember* = "ODepartment";  }  }  }  }  } |

当下拉列表选项改变后，第二个下拉列表绑定对应的信息

|  |
| --- |
| private void comboBox1\_SelectedIndexChanged(object sender, *EventArgs* e)  {  OrgStruc org = comboBox1.*SelectedItem* as OrgStruc;  if (org != null)  {  BindDepartment(org.Oid, comboBox2);  }  else  {  comboBox2.*Items*.*Clear*();  }  } |

# 存储过程

在数据库中有以下的存储过程。输入小写会转换为大写。

|  |
| --- |
| create proc ConvertToUpper  @str varchar(20) out  as  set @str = UPPER(@str) |

如何在程序中执行sql存储过程呢？

执行存储过程步骤如下：

* 设置CommandText为存储过程的名称
* 设置CommandType的类型为StoredProcedure
* 通过SqlParameter传递值到存储过程的参数中，如存储过程有输出值，需要设置SqlParameter的方向为InputOutput
* 传递进数据库以及从数据库传递出来的值存储在SqlParameter的value属性中。

|  |
| --- |
| string inputTxt = textBox1.*Text*.*Trim*();  string connstr = *System*.*Configuration*.ConfigurationManager.ConnectionStrings["MyDb"].*ConnectionString*;  using (*SqlConnection* conn = new *SqlConnection*(connstr))  {  conn.*Open*();  using (*SqlCommand* cmd = conn.*CreateCommand*())  {  //1.执行语句为存储过程的名称  cmd.*CommandText* = "ConvertToUpper";  //2/执行类型为StoredProcedure。默认为sql语句  cmd.*CommandType* = *CommandType*.*StoredProcedure*;  //3.提供存储过程的参数  *SqlParameter* sp1 = new *SqlParameter*("@str", inputTxt);  //3.1数为双向的。会输出值  sp1.*Direction*= *ParameterDirection*.*InputOutput*;  //4.添加参数  cmd.*Parameters*.*Add*(sp1);  cmd.*ExecuteNonQuery*();  //5.通过参数获得存储过程的返回值  textBox1.*Text* = sp1.*Value*.*ToString*();  }  } |

# DataSet与Adapter

## DataSet数据集

DataSet是存在于数据库中的临时数据库，支持断开式操作。它是DataTable的集合。

Tables属性：获取DataTable集合。

## SqlDataAdapter

adapter 负责数据库和数据集的联系。

Fill方法：用于查询，将数据库中的信息查询出来，放到内存中的对象DataSet内。

Update方法：用于将DataSet中的数据更新到数据库中。更新包括：insert、update、delete，分别用insertCommand, updateCommand,deleteCommand属性实现。

## 实例：把数据库中信息展示到GridView中

|  |
| --- |
| private void Form1\_Load(object sender, *EventArgs* e)  {  BindData();  }  /// <summary>  /// 绑定数据  /// </summary>  private void BindData()  {  string connStr = System.*Configuration*.*ConfigurationManager*.*ConnectionStrings*["MyDB"].ConnectionString;  using (SqlConnection conn = new SqlConnection(connStr))  {  string sql = "select \* from Orders";  //选择语句可以直接在构造函数中进行。如果是修改则需要创建SqlCommand对象。  *SqlDataAdapter* adapter = new *SqlDataAdapter*(sql, conn);  DataSet ds = new DataSet();  conn.Open();  adapter.*Fill*(ds); //本句才真正的执行sql语句，将取得的数据放到ds中。  //把ds数据集中的第一个表绑定到Datagridview上。  dataGridView1.DataSource = ds.Tables[0];  }  } |

## DataSet实现原理

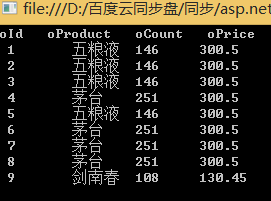
当执行SqlDataAdapter的Fill方法时:

* 按行读取所有的数据（SqlDataReader）
* 取得列的数量和类型，添加到DataColumns对象中。
* 遍历每一行的数据，添加到DataRows对象中。
* 把DataColumns和DataRows组成的表添加到DataTable中。
* 创建Dataset对象，把DataTable添加到Dataset对象中。
* 如果有多个表，按照上面的步骤循环，DateSet最后可以持有DataTable的集合。

实际的底层运行方式如下面代码所示。

|  |
| --- |
| using System;  using System.*Collections*.*Generic*;  using System.Data;  using System.Data.SqlClient;  using System.*Linq*;  using System.*Text*;  using System.*Threading*.*Tasks*;  namespace \_09\_dataset原理  {  class Program  {  static void Main(string[] args)  {  DataSet ds = DataToTable();  for (int i = 0; i < ds.Tables[0].*Columns*.*Count*; i++)  {  *DataColumn* dc = ds.Tables[0].*Columns*[i];  *Console*.*Write*(dc + " ");  }  *Console*.*WriteLine*();  for (int i = 0; i < ds.Tables[0].*Rows*.*Count*; i++)  {  *DataRow* drc = ds.Tables[0].*Rows*[i];  for (int j = 0; j < ds.Tables[0].*Columns*.*Count*; j++)  {  *Console*.*Write*(" " + drc[j] + "\t");  }  *Console*.*WriteLine*();  }  *Console*.*WriteLine*();  *Console*.*Read*();  }  /// <summary>  /// DataSet原理  /// </summary>  /// <returns></returns>  static DataSet DataToTable()  {  string connStr = System.*Configuration*.*ConfigurationManager*.*ConnectionStrings*["MyDB"].ConnectionString;  using (SqlConnection conn = new SqlConnection(connStr))  {  using (*SqlCommand* cmd = conn.CreateCommand())  {  cmd.*CommandText* = "select \* from Orders";  conn.Open();  //1. SqlDataReader读取所有数据  using (SqlDataReader reader = cmd.*ExecuteReader*())  {  //2. 创建DataTable对象。  *DataTable* table1 = new *DataTable*();  int columnsNums = reader.FieldCount;  //3. 循环列，把列添加到DataTalbe对象中。  for (int i = 0; i < columnsNums; i++)  {  //3.1 从reader中获取列名称与类型，赋值给DataColumn对象  *DataColumn* dc = new *DataColumn*();  dc.*ColumnName* = reader.GetName(i);  dc.*DataType* = reader.GetFieldType(i);  //3.2 把列添加给DataTalbe对象  table1.*Columns*.*Add*(dc);  }  //4.为DataTable对象添加行  while (reader.Read())  {  //4.1创建新行  *DataRow* dr = table1.*NewRow*();  //4.2 循环遍历，为每一行赋值。  for (int i = 0; i < columnsNums; i++)  {  dr[i] = reader[i];  }  //4.3 把行对象数据添加到table对象中。  table1.*Rows*.*Add*(dr);  }  //5.创建Dataset对象  DataSet ds = new DataSet();  //6.把DataTable对象添加到dataset对象中。  ds.Tables.*Add*(table1);  return ds;  }  }  }  }  }  } |

效果如下：



## 数据更新

### updateCommand

当数据库中的数据和DataSet中的数据不一样时，使用updateCommand可以将DataSet中修改了的数据更新到数据库中。

|  |
| --- |
| private void button1\_Click(object sender, *EventArgs* e)  {  //1. 转换dataGridview绑定的数据源为Datatable类型  *DataTable* dt = dataGridView1.DataSource as *DataTable*;  if (dt != null)  {  //2.连接数据库  string connStr = System.*Configuration*.*ConfigurationManager*.*ConnectionStrings*["MyDB"].ConnectionString;  using (SqlConnection conn = new SqlConnection(connStr))  {  //3.创建适配器  *SqlDataAdapter* adapter = new *SqlDataAdapter*();  //4. 创建SqlCommand对象。  *SqlCommand* cmdUpate = conn.CreateCommand();  //4.1 更新语句  cmdUpate.*CommandText* = "update Orders set oCount=@Count, oPrice=@price where oId=@id";  //4.2 设置数据库参数的类型，长度以及DataTable中字段的名称。  cmdUpate.*Parameters*.*Add*("@Count", *SqlDbType*.*Int*, 4, "OCount");  cmdUpate.*Parameters*.*Add*("@price", *SqlDbType*.*Float*, 8, "oPrice");  cmdUpate.*Parameters*.*Add*(new *SqlParameter*("@id", *SqlDbType*.*Int*, 4, "oId"));  //5. UpdateCommand为要执行的语句对象。  adapter.*UpdateCommand* = cmdUpate;  conn.Open();  //6. 开始执行SQL插入、删除或更新语句。  adapter.*Update*(dt);  }  }  } |

### insertCommand

当DataSet中有的数据，在数据库中没有时，使用insertCommand在数据库中插入数据。

|  |
| --- |
| private void button2\_Click(object sender, *EventArgs* e)  {  //1. 转换dataGridview绑定的数据源为Datatable类型  *DataTable* dt = dataGridView1.DataSource as *DataTable*;  if (dt != null)  {  //2.连接数据库  string connStr = System.*Configuration*.*ConfigurationManager*.*ConnectionStrings*["MyDB"].ConnectionString;  using (SqlConnection conn = new SqlConnection(connStr))  {  //3.创建适配器  *SqlDataAdapter* adapter = new *SqlDataAdapter*();  //4. 创建SqlCommand对象。  *SqlCommand* cmdInsert = conn.CreateCommand();  //4.1 更新语句  cmdInsert.*CommandText* = "insert into Orders values(@product,@Count,@price)";  //4.2 设置数据库参数的类型，长度以及DataTable中字段的名称。  cmdInsert.*Parameters*.*Add*("@product", *SqlDbType*.*NVarChar*, 20, "oProduct");  cmdInsert.*Parameters*.*Add*("@Count", *SqlDbType*.*Int*, 4, "OCount");  cmdInsert.*Parameters*.*Add*("@price", *SqlDbType*.*Float*, 8, "oPrice");  //5. InsertCommand为要执行的语句对象。  adapter.*InsertCommand* = cmdInsert;  conn.Open();  //6. 开始执行SQL插入、删除或更新语句。  adapter.*Update*(dt);  *MessageBox*.*Show*("插入成功");  }  }  } |

### deleteCommand

当数据库中有的数据，在DataSet中不存在时，使用deleteCommand删除数据库中存在的数据。

|  |
| --- |
| private void button3\_Click(object sender, *EventArgs* e)  {  //1. 转换dataGridview绑定的数据源为Datatable类型  *DataTable* dt = dataGridView1.DataSource as *DataTable*;  if (dt != null)  {  //2.连接数据库  string connStr = System.*Configuration*.*ConfigurationManager*.*ConnectionStrings*["MyDB"].ConnectionString;  using (SqlConnection conn = new SqlConnection(connStr))  {  //3.创建适配器  *SqlDataAdapter* adapter = new *SqlDataAdapter*();  //4. 创建SqlCommand对象。  *SqlCommand* cmdDelete = conn.CreateCommand();  //4.1 更新语句  cmdDelete.*CommandText* = "delete from Orders where oId=@id";  //4.2 设置数据库参数的类型，长度以及及DataTable中字段的名称。  cmdDelete.*Parameters*.*Add*("@id", *SqlDbType*.*Int*, 4, "oId");  //5. InsertCommand为要执行的语句对象。  adapter.*DeleteCommand* = cmdDelete;  conn.Open();  //6. 开始执行SQL插入、删除或更新语句。  adapter.*Update*(dt);  *MessageBox*.*Show*("删除成功");  }  }  } |

### SqlCommandBuilder

SqlCommandBuilder可以自动创建Sql语句，无论是更新、删除还是插入数据都可以用这一种方式。

|  |
| --- |
| using (SqlConnection conn = new SqlConnection(directory))  {  conn.Open();  using (SqlCommand cmd = conn.CreateCommand())  {  cmd.CommandText = "select \* from T\_users";  DataSet dt = new DataSet();  SqlDataAdapter adapter = new SqlDataAdapter(cmd);  adapter.Fill(dt);  DataTable talbe = dt.Tables[0];  DataRow row = talbe.Rows[0];  //----------------修改DataSet中数据，并提交到数据哭---------------------  row["Name"] = "jason";  SqlCommandBuilder builder = new SqlCommandBuilder(adapter);//自动创建sql语句  adapter.Update(dt);//提交修改的数据到数据库  }  } |

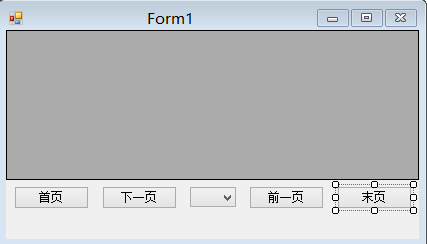
# 分页查询

## 语句分页查询

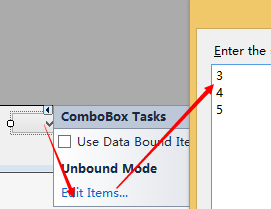
1. 首先，在数据库中创建一个视图，按照oId编号，以方便在程序中进行查询。

|  |
| --- |
| create view wineOrderWithIndex  as  select \*,ROW\_NUMBER() over(order by oId desc) as rowIndex from Orders |

1. 设计winfrom界面



1. 设置每页可以显示多少条数据



在窗口加载时，默认选中第一项:

|  |
| --- |
| private void Form1\_Load(object sender, *EventArgs* e)  {  comboBox1.*SelectedIndex* = 0;  } |

1. 设置页面导航的四个按钮为同一个方法pagination。封装的GetRowNums用来返回数据库中数据的行数。

|  |
| --- |
| //定义页码  int PageIndex = 1;  int CountofEveryPage;  private void pagination(object sender, *EventArgs* e)  {  *Button* btn = sender as *Button*;  //最后一页的索引：数据总行数除每一页的数据条数。  int lastPageIndex = (int)*Math*.*Ceiling*(GetRowNums() \* 1.0 / CountofEveryPage);  switch (btn.*Text*)  {  case "首页":  PageIndex = 1;  break;  case "末页":  PageIndex = lastPageIndex;  break;  case "前一页":  PageIndex--;  if (PageIndex <= 0)  {  PageIndex = 1;  }  break;  case "下一页":  PageIndex++;  if (PageIndex >= lastPageIndex)  {  PageIndex = lastPageIndex;  }  break;  }  }  /// <summary>  /// 获取数据库中的总行数  /// </summary>  /// <returns></returns>  private int GetRowNums()  {  int rowCount = 0;  string connStr = *System*.*Configuration*.*ConfigurationManager*.*ConnectionStrings*["MyDB"].*ConnectionString*;  using (*SqlConnection* conn = new *SqlConnection*(connStr))  {  using (*SqlCommand* cmd = conn.*CreateCommand*())  {  //wineOrderWithIndex为视图的名称  cmd.*CommandText* = "select count(\*) from wineOrderWithIndex";  //ExecuteScalar取得ount(\*)函数的值，并被转换为int类型。  conn.*Open*();  rowCount = (int)cmd.*ExecuteScalar*();  }  }  //返回行数。  return rowCount;  } |

1. 窗口加载时，加载数据

|  |
| --- |
| private void Form1\_Load(object sender, *EventArgs* e)  {  comboBox1.*SelectedIndex* = 0;  //调用方法绑定数据  BindDateByPaging();  }  private void BindDateByPaging()  {  int CountofEveryPage = *Convert*.*ToInt32*(comboBox1.*Text*);  string connStr = *System*.*Configuration*.*ConfigurationManager*.*ConnectionStrings*["MyDB"].*ConnectionString*;  using (*SqlConnection* conn = new *SqlConnection*(connStr))  {  using (*SqlCommand* cmd = conn.*CreateCommand*())  {  cmd.*CommandText* = "select rowIndex,oProduct,oCount,oPrice from wineOrderWithIndex where rowIndex between @StartIndex and @EndIndex";  //起始索引： 前一页乘以每页的数据+1。即是这一页的开始  cmd.*Parameters*.*AddWithValue*("@StartIndex", (PageIndex - 1) \* CountofEveryPage + 1);  //结束索引：页乘以每页数据的条数。  cmd.*Parameters*.*AddWithValue*("@EndIndex", PageIndex \* CountofEveryPage);  *SqlDataAdapter* adapter = new *SqlDataAdapter*();  conn.*Open*();  //SelectCommand获取要执行的Select语句。  adapter.*SelectCommand* = cmd;  *DataSet* ds = new *DataSet*();  //开始执行sql语句，并填充到Dataset中  adapter.*Fill*(ds);  //DataSet中的表绑定到DataGridView中  dataGridView1.*DataSource* = ds.*Tables*[0];  }  }  } |

1. 最后，当改变每页显示的数据条数时，也调用绑定方法

|  |
| --- |
| private void comboBox1\_SelectedIndexChanged(object sender, *EventArgs* e)  {  BindDateByPaging();  } |

## 存储过程分页查询

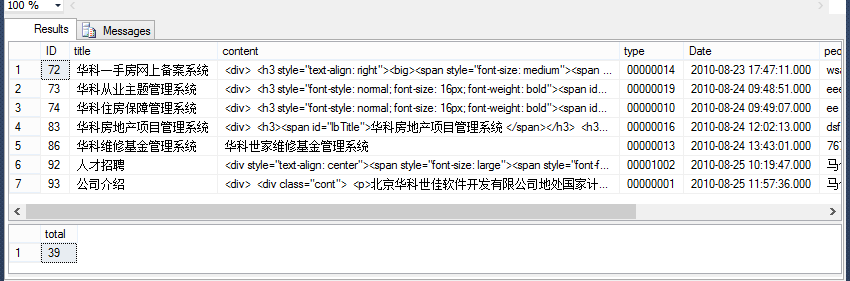
### 数据库中创建存储过程

|  |
| --- |
| create procedure P\_Paging  --参数  @pageIndex int,  @pageSize int,  @total int out  as  --代码:分页原理：越过多少条，取多少条  select top(@pageSize) \* from dbo.HKSJ\_Main where id not in  (  select top((@pageIndex-1)\*@pageSize) ID from dbo.HKSJ\_Main order by ID  )  order by ID  select @total = COUNT(id) from dbo.HKSJ\_Main |

测试

|  |
| --- |
| declare @total int  exec P\_Paging 5,7,@total out  select @total as total |

结果：



### ado.net执行存储过程

|  |
| --- |
| public List<Model.HKSJ\_Main> getPaging(int pageIndex, int pageSize, out int total)  {  List<Model.HKSJ\_Main> model = new List<Model.HKSJ\_Main>();  string connStr = System.Configuration.ConfigurationManager.ConnectionStrings["Mydb"].ConnectionString;  DataSet ds = new DataSet();  using (SqlConnection conn = new SqlConnection(connStr))  {  using (SqlDataAdapter adapter = new SqlDataAdapter("P\_Paging", conn))  {  //参数赋值  adapter.SelectCommand.Parameters.Add(new SqlParameter("@pageIndex", pageIndex));  adapter.SelectCommand.Parameters.Add(new SqlParameter("@pageSize", pageSize));  //输出参数的用法  SqlParameter totalParameter = new SqlParameter("@total", SqlDbType.Int);  totalParameter.Direction = ParameterDirection.Output;  adapter.SelectCommand.Parameters.Add(totalParameter);  //设置执行方式为存储过程  adapter.SelectCommand.CommandType = CommandType.StoredProcedure;  adapter.Fill(ds);  //获取输出out值  total = (int)totalParameter.Value;  }  }  //返回强类型  return DataSetToList(ds);  } |

转换为强类型的方法

|  |
| --- |
| private List<Model.HKSJ\_Main> DataSetToList(DataSet ds)  {  List<Model.HKSJ\_Main> modelList = new List<Model.HKSJ\_Main>();  for (int i = 0; i < ds.Tables.Count; i++)  {  int rowsCount = ds.Tables[i].Rows.Count;  if (rowsCount > 0)  {  Model.HKSJ\_Main model;  for (int j = 0; j < rowsCount; j++)  {  model = this.DataRowToModel(ds.Tables[i].Rows[j]);  if (model != null)  {  modelList.Add(model);  }  }  }  }  return modelList;  } |

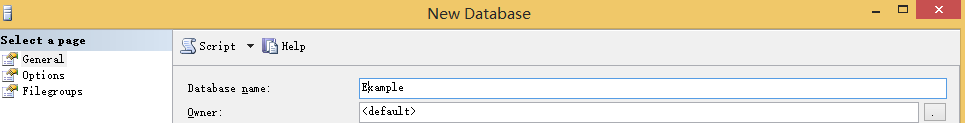
填充数据到model中的方法

|  |
| --- |
| public Hyf.News.Model.HKSJ\_Main DataRowToModel(DataRow row)  {  Hyf.News.Model.HKSJ\_Main model = new Hyf.News.Model.HKSJ\_Main();  if (row != null)  {  if (row["ID"] != null && row["ID"].ToString() != "")  {  model.ID = int.Parse(row["ID"].ToString());  }  if (row["title"] != null)  {  model.title = row["title"].ToString();  }  if (row["content"] != null)  {  model.content = row["content"].ToString();  }  if (row["type"] != null)  {  model.type = row["type"].ToString();  }  if (row["Date"] != null && row["Date"].ToString() != "")  {  model.Date = DateTime.Parse(row["Date"].ToString());  }  if (row["people"] != null)  {  model.people = row["people"].ToString();  }  if (row["picUrl"] != null)  {  model.picUrl = row["picUrl"].ToString();  }  }  return model;  } |

# 综合案例：用户管理

## 创建数据库

首先创建名为Example的数据库：



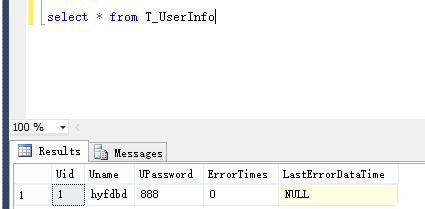
创建T\_UserInfo表

|  |
| --- |
| use Example  create Table T\_UserInfo  (  Uid int identity(1,1) not null primary key ,  Uname nvarchar(16) not null,  UPassword varchar(20) not null,  ErrorTimes int default(0),  LastErrorDataTime datetime  ) |

添加一个测试信息：

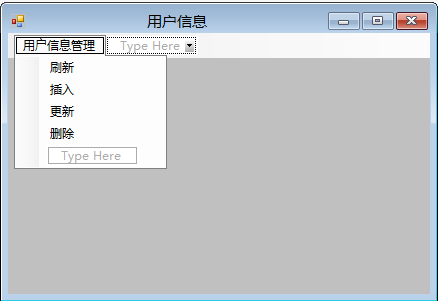
|  |
| --- |
| insert into T\_UserInfo(Uname,UPassword) values('hyfdbd','888') |

查看表中信息：



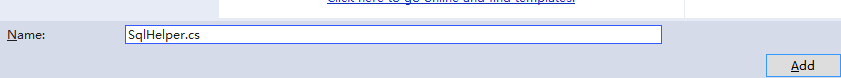
## 创建界面

一个DataGridVIew和一个MenuStrip



## 创建SqlHelper类

sqlHelper类封装了sql的连接与读取。



代码如下：

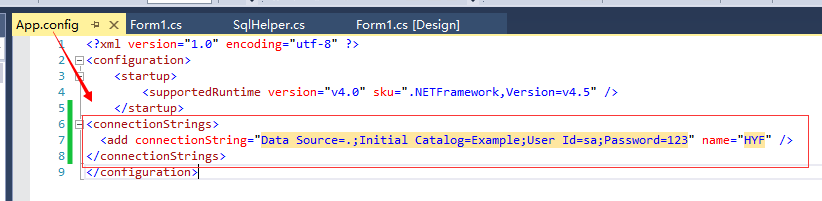
|  |
| --- |
| using *System*;  using *System*.*Collections*.*Generic*;  using *System*.*Data*;  using *System*.*Data*.*SqlClient*;  using *System*.*Linq*;  using *System*.*Text*;  using *System*.*Threading*.*Tasks*;  namespace \_01\_用户信息管理  {  public static class SqlHelper  {  private static *SqlConnection* getSqlConn()  {  string connStr = *System*.*Configuration*.*ConfigurationManager*.*ConnectionStrings*["HYF"].*ConnectionString*;  return new *SqlConnection*(connStr);  }  public static *DataTable* getDataTalbe(string sql, params *SqlParameter*[] sp)  {  *DataTable* dt = new *DataTable*();  using (*SqlConnection* conn = getSqlConn())  {  try  {  conn.*Open*();  *SqlDataAdapter* adapter = new *SqlDataAdapter*(sql, conn);  if (sp.*Length* > 0)  {  adapter.*SelectCommand*.*Parameters*.*AddRange*(sp);  }  adapter.*Fill*(dt);  }  catch (*Exception* e)  {  throw e;  }  }  return dt;  }  }  } |

## 窗口加载时加载数据

设计一个绑定方法，在窗口加载时调用：

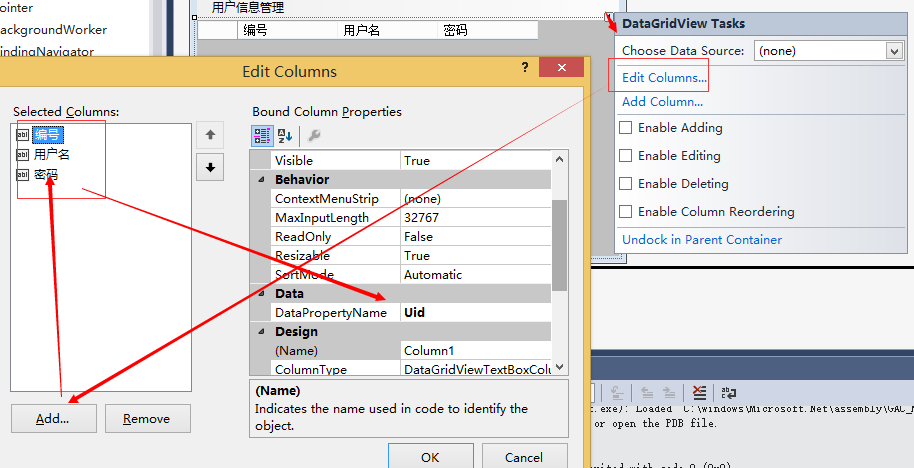
|  |
| --- |
| private void Form1\_Load(object sender, *EventArgs* e)  {  string sql = "select \* from T\_UserInfo";  BindData(sql);  }  void BindData(string sql)  {  *DataTable* dt = SqlHelper.getDataTalbe(sql);  dataGridView1.*DataSource* = dt;  } |

## 设置配置文件

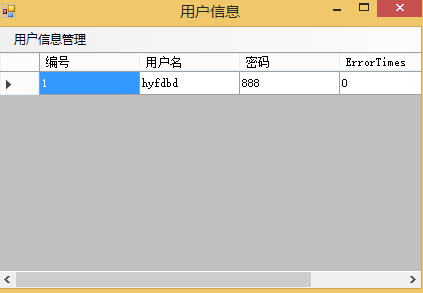


## DataGridView编辑列

添加3个列，分布设置DataPropertyName与Uid、UName、Upassword绑定

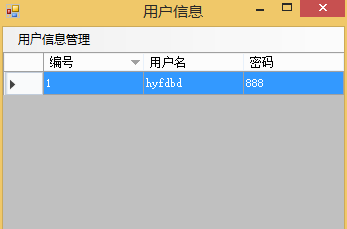


运行，效果如下：



如果不想显示ErrorTImes可以设置AutoGenerateColumns为false：

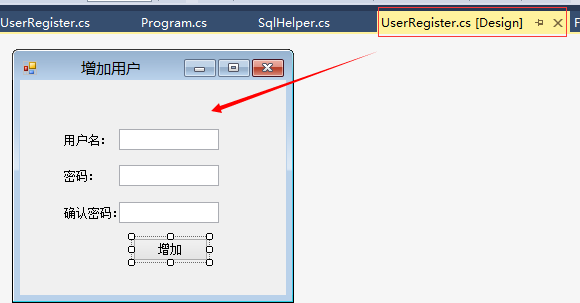
|  |
| --- |
|  |



## 增加用户数据

### 创建用户注册界面

新建winform窗体，添加lable、textbox以及button



### 查询数据库中用户名是否存在

在sqlHelper中创建查询方法，用来判断用户名是否已经存在

|  |
| --- |
| /// <summary>  /// 执行sql语句，返回首行首列  /// </summary>  /// <param name="sql"></param>  /// <returns></returns>  public static object *ExecuteScalar*(string sql)  {  object obj = null;  using (*SqlConnection* conn = getSqlConn())  {  *SqlCommand* cmd = new *SqlCommand*(sql, conn);  try  {  conn.*Open*();  }  catch  {  return null;  }  obj = cmd.*ExecuteScalar*();  }  return obj;  }  /// <summary>  /// 调用ExecuteScalar，判断是否有返回数据，有则返回true  /// </summary>  /// <param name="sql"></param>  /// <returns></returns>  public static bool IsExist(string sql)  {  if (SqlHelper.*ExecuteScalar*(sql) != null)  {  return true;  }  else  {  return false;  }  } |

### 添加插入数据的方法

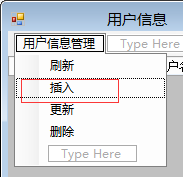
在Sqlhelper中创建添加数据的方法

|  |
| --- |
| public static int *ExecuteNonQuery*(string sql, *Dictionary*<string, string> dc)  {  int nums = -1;  using (*SqlConnection* conn = getSqlConn())  {  *SqlCommand* cmd = new *SqlCommand*(sql, conn);  *SqlParameter*[] sp = new *SqlParameter*[dc.*Count*];  int index = 0;  foreach (var item in dc)  { //字典中的键作为参数名，值作为参数值  sp[index++] = new *SqlParameter*(item.*Key*, item.*Value*);  }  cmd.*Parameters*.*AddRange*(sp);  try  {  conn.*Open*();  }  catch (*Exception* e)  {  return -1;  }  nums = cmd.*ExecuteNonQuery*();  }  return nums;  } |

### 点击按钮，增加用户

|  |
| --- |
| private void button1\_Click(object sender, *EventArgs* e)  { //采集textbox中的数据  string uid = txtUid.*Text*.*Trim*();  string pwd = txtPwd.*Text*;  string pwdAgain = txtpwdagain.*Text*;  //密码是否相等，不相等提示  if (pwd != pwdAgain)  {  *MessageBox*.*Show*("密码不一致");  return;  }  //判断用户名是否存在，调用sqlHelper中的IsExist方法  string sql = "select count(\*) from T\_UserInfo";  if (!SqlHelper.IsExist(sql))  {  *MessageBox*.*Show*("用户名已存在！");  }  else  {  //插入数据。  *Dictionary*<string, string> dc = new *Dictionary*<string, string>();  string sqlInsert = "insert into T\_UserInfo(Uname,UPassword) values (@name,@pwd)";  dc.*Add*("@name", uid);  dc.*Add*("@pwd", pwd);  int num = SqlHelper.*ExecuteNonQuery*(sqlInsert, dc);  if (num == 1)  {  *MessageBox*.*Show*("注册成功");  //设置这个窗口的DialogResult。设置后窗口后隐藏  this.*DialogResult* = *DialogResult*.*Abort*;  }  }  } |

### 点击主窗口中的“插入”菜单项，打开用户注册界面



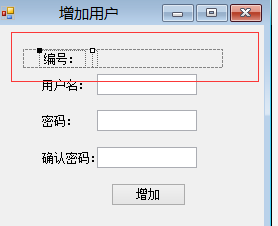
事件如下：

|  |
| --- |
| private void 插入ToolStripMenuItem\_Click(object sender, *EventArgs* e)  {  //创建用户注册界面  UserRegister urFrom = new UserRegister();  //显示用户注册界面  urFrom.*ShowDialog*();  //如果用户注册界面的DialogResult值为abort，执行if语句块中内容  if (urFrom.*DialogResult* == *DialogResult*.*Abort*)  {  //这是刷新dategridview  string sql = "select \* from T\_UserInfo order by uid desc";  BindData(sql);  }  } |

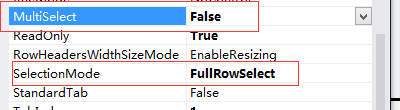
## 修改用户数据

### 修改用户注册界面

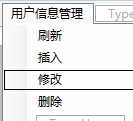
为了重用，减少对象的创建，对用户注册界面进行修改。增加一个panle里面放两个标签



### 取消选择多行



### 点击修改菜单项时，创建界面



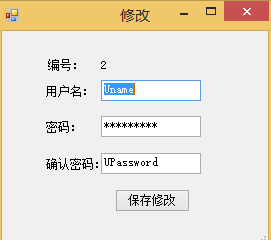
把当前选择行的id放到tag中

|  |
| --- |
| private void 更新ToolStripMenuItem\_Click(object sender, *EventArgs* e)  {  UserRegister urForm = new UserRegister();  if(dataGridView1.*SelectedRows*.*Count*<=0)  {  *MessageBox*.*Show*("请选择要修改的用户");  return;  }  //获取用户ID，放到tag中  urForm.*Tag* =dataGridView1.*SelectedRows*[0].*Cells*[0].*Value*;  *DialogResult* dr = urForm.*ShowDialog*();  if(dr == *DialogResult*.*OK*)  {  //这是刷新dategridview  string sql = "select \* from T\_UserInfo order by uid desc";  BindData(sql);  }  } |

### 初始化界面

用户注册界面加载时，初始化界面信息。

|  |
| --- |
| private void UserRegister\_Load(object sender, *EventArgs* e)  {  //tag不为空则是修改用户信息  if(this.*Tag* !=null)  {  //初始化界面  int tag = *Convert*.*ToInt32*(this.*Tag*);  lblUid.*Text* = *Convert*.*ToString*(tag);  this.*Text* = "修改";  button1.*Text* = "保存修改";  string sql = "select \* from T\_UserInfo where @id = uid";  *SqlParameter* sp = new *SqlParameter*("@id",tag);  *DataTable* dt = SqlHelper.getDataTalbe(sql, sp);  txtUid.*Text* = dt.*Rows*[0]["Uname"].*ToString*();  txtPwd.*Text* = dt.*Rows*[0]["UPassword"].*ToString*();  txtpwdagain.*Text* = dt.*Rows*[0]["UPassword"].*ToString*();    }else  {  lblforUid.*Text* = string.*Empty*;  }  } |



### 点击保存修改，修改用户信息

|  |
| --- |
| private void button1\_Click(object sender, *EventArgs* e)  {  //如果编号不为空，则调用修改方法  if (lblUid.*Text* != string.*Empty*)  {  UpdateUserInfo();  }  else  {  insertUserInfo();  }  } |

UpdateUserInfo方法如下：

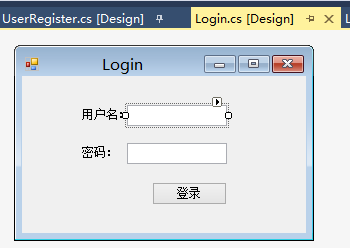
|  |
| --- |
| private void UpdateUserInfo()  {  //采集textbox中的数据  string uname = txtUid.*Text*.*Trim*();  string pwd = txtPwd.*Text*;  string pwdAgain = txtpwdagain.*Text*;  int id = *Convert*.*ToInt32*(this.*Tag*);  //密码是否相等，不相等提示  if (pwd != pwdAgain)  {  *MessageBox*.*Show*("密码不一致");  return;  }  //判断用户名是否已经存在，并且不包含本行的用户名。  string sql = "select count(\*) from T\_UserInfo where uname=@uname and uid != @uid";  *SqlParameter*[] sp = new *SqlParameter*[2];  sp[0] = new *SqlParameter*("@uname", uname);  sp[1] = new *SqlParameter*("@uid", id);  *DataTable* dt = SqlHelper.getDataTalbe(sql, sp);  //如果不存在，则修改数据  if ((int)dt.*Rows*[0][0] == 0)  {  sql = "update T\_UserInfo set Uname= @name, UPassword=@password where uid=@id";  *Dictionary*<string, string> dc = new *Dictionary*<string, string>();  dc.*Add*("@name", uname);  dc.*Add*("@password", pwd);  dc.*Add*("@id", id.*ToString*());  int rows = SqlHelper.ExecuteNonQuery(sql, dc);  if (rows == 1)  {  *MessageBox*.*Show*("修改成功!");  }  else  {  *MessageBox*.*Show*("修改失败!");  }  }  else  {  *MessageBox*.*Show*("用户名已经存在");  }  this.*DialogResult* = *DialogResult*.*OK*;  } |

## 删除用户数据

|  |
| --- |
| private void 删除ToolStripMenuItem\_Click(object sender, *EventArgs* e)  {  if (dataGridView1.*SelectedRows*.*Count* > 0)  {  *DialogResult* dr = *MessageBox*.*Show*("确定要删除吗？");  *StringBuilder* sb = new *StringBuilder*();  if (dr == *System*.*Windows*.*Forms*.*DialogResult*.*OK*)  {  for (int i = 0; i < dataGridView1.*SelectedRows*.*Count*; i++)  {  sb.*Append*(dataGridView1.*SelectedRows*[i].*Cells*[0].*Value* + ",");  }  sb.*Remove*(sb.*Length* - 1, 1);  }  string sql = "delete from T\_UserInfo where uid in(" + sb.*ToString*() + ")";  int num = SqlHelper.*ExecuteNonQuery*(sql, null);  if(num >0)  {  *MessageBox*.*Show*("删除成功！");  //这是刷新dategridview  sql = "select \* from T\_UserInfo order by uid desc";  BindData(sql);  }  }  } |

## 用户登录

### 新建一个winform窗体



### 修改入口方法使login在程序打开时加载

|  |
| --- |
| static void *Main*()  {  *Application*.*EnableVisualStyles*();  *Application*.*SetCompatibleTextRenderingDefault*(false);  //创建注册窗口  Login login = new Login();  //显示注册窗口  *DialogResult* dr = login.*ShowDialog*();  //注册窗口DialogResult状态改变后，启动From1  if(dr == *DialogResult*.*OK*)  {  *Application*.*Run*(new Form1());  }  } |

### 点击登陆按钮时，进行注册

|  |
| --- |
| private void button1\_Click(object sender, *EventArgs* e)  {  //1.采集用户输入的信息  string pwd = txtPwd.*Text*;  string userName = txtname.*Text*.*Trim*();  //2.根据用户名进行查询  string sql = "select UPassword,ErrorTimes, DATEDIFF(MINUTE,LastErrorDataTime,GETDATE()) as ErrorLoginTime from T\_UserInfo where Uname=@name";  *SqlParameter*[] sp = new *SqlParameter*[1];  sp[0] = new *SqlParameter*("@name", userName);  *DataTable* dt = SqlHelper.getDataTalbe(sql, sp);  //获取最后一次错误登录时间  int LastErrorLoginTime = *Convert*.*ToInt32*(dt.*Rows*[0]["ErrorLoginTime"]);  //获取错误次数  int errortimes = *Convert*.*ToInt32*(dt.*Rows*[0]["ErrorTimes"]);  //账户锁定后多久后可以登录  int CanLogin = 4;  //如果table中没有数据表示没有该用户  if (dt.*Rows*.*Count* <= 0)  {  *MessageBox*.*Show*("用户名不存在");  }  //如果有该账户，则判断密码、错误次数、错误的时间  else  {  //错误次数小于3次或者最后的错误时间大于定义的锁定多久后可以登录的时间。  if (errortimes != 3 || LastErrorLoginTime > CanLogin)  {  //密码相等，则成功登陆  if (dt.*Rows*[0]["UPassword"].*ToString*() == pwd)  {  //初始化错误次数为0  sql = "update T\_UserInfo set ErrorTimes=0 where Uname=@Dname";  *Dictionary*<string, string> dc = new *Dictionary*<string, string>();  dc.*Add*("@Dname", txtname.*Text*.*ToString*());  SqlHelper.ExecuteNonQuery(sql, dc);  //设置login的状态，登录界面会隐藏。  this.*DialogResult* = *DialogResult*.*OK*;  }  //密码错误  else  {  //最后的错误时间小于定义的锁定多久后可以登录的时间。初始化错误次数为0  if (LastErrorLoginTime > CanLogin)  {  sql = "update T\_UserInfo set ErrorTimes=0 where Uname=@Dname";  *Dictionary*<string, string> dc = new *Dictionary*<string, string>();  dc.*Add*("@Dname", txtname.*Text*.*ToString*());  int i = SqlHelper.ExecuteNonQuery(sql, dc);  }  //如果错误小于3，每错误一次ErrorTimes加1.  if (errortimes < 3)  {  sql = "update T\_UserInfo set ErrorTimes=ErrorTimes+1,LastErrorDataTime=getdate()where Uname=@Dname";  *Dictionary*<string, string> dc = new *Dictionary*<string, string>();  dc.*Add*("@Dname", txtname.*Text*.*ToString*());  int i = SqlHelper.ExecuteNonQuery(sql, dc);  *MessageBox*.*Show*("密码错误！");  }  }  }  //错误次数大于3，并且在账户锁定时间内，进行提示。  else  {  *MessageBox*.*Show*("错误次数太多，请" + (CanLogin - LastErrorLoginTime).*ToString*() + "分后再登陆");  }  }  } |

### 关闭主界面时退出程序

点击右上角的“X”按钮时，调用FormClosed事件。关闭程序

|  |
| --- |
| private void Form1\_FormClosed(object sender, *FormClosedEventArgs* e)  {  *Application*.*Exit*();  } |

## 导出数据

### 新建导出方法

新建一个静态类，写一个导出方法：

|  |
| --- |
| public static class HyfTools  {  /// <summary>  /// 导出文件  /// </summary>  /// <param name="path">导出文件的保存地址</param>  public static void ExportData(string path)  {  string sql = "select Uid,Uname,uPassword from T\_userInfo";  *DataTable* dt = SqlHelper.getDataTalbe(sql);  //导出文件为UserInfo.csv。可以是任何文本类型。  path = *Path*.*Combine*(path, "UserInfo.csv");  using (*StreamWriter* writer = new *StreamWriter*(path))  {  for (int i = 0; i < dt.*Rows*.*Count*; i++)  { //csv文件两个值中间用，号隔开  writer.*WriteLine*("{0},{1},{2}", dt.*Rows*[i][0].*ToString*(), dt.*Rows*[i][1].*ToString*(), dt.*Rows*[i][2].*ToString*());  }  }  }  } |

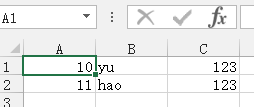
### 创建导出菜单项



### 点击导出按钮，选择导出位置，进行导出

|  |
| --- |
| private void 导出ToolStripMenuItem\_Click(object sender, *EventArgs* e)  {  //创建一个文件浏览器窗口，可以选择文件要保存的位置。  *FolderBrowserDialog* fd = new *FolderBrowserDialog*();  *DialogResult* dr = fd.*ShowDialog*();  //点击确定按钮，调用 HyfTools.ExportData方法  if(dr==*DialogResult*.*OK*)  {  //文件浏览器窗口所选择的路径  string path = fd.*SelectedPath*;  HyfTools.ExportData(path);  }  } |

保存结果如下：

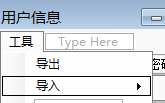


## 导入数据

### 创建导入方法

|  |
| --- |
| public static void ImportData(string file)  {  string sql = "insert into T\_UserInfo(Uname,UPassword) ";  using (*StreamReader* reader = new *StreamReader*(file))  {  while (!reader.*EndOfStream*)  {  string[] strs = reader.*ReadLine*().*Split*(',');  string *name* = strs[1];  string pwd = strs[2];  //判断是否包含名称。  if (!SqlHelper.IsExist("select \* from T\_userinfo where Uname='" + *name* + "'"))  {  //拼接字符串。结果类似：insert into T\_UserInfo(Uname,UPassword) select '值','值' union select '值','值' union  sql += " select '" + *name* + "','" + pwd + "' union ";  }  }  // 去掉最后多余的union  sql = sql.*ToString*().*Substring*(0, sql.*Length* - 6);  try  { //当添加的文件名字全部重复时，会出错。这里只是简单的返回  int nums = SqlHelper.ExecuteNonQuery(sql, null);  }  catch  {  return;  }  }  } |

### 创建导入菜单项



### 点击导入按钮，选择文件，导入数据

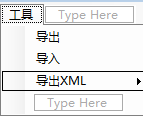
|  |
| --- |
| private void 导入ToolStripMenuItem\_Click(object sender, *EventArgs* e)  {  *OpenFileDialog* ofd = new *OpenFileDialog*();  *DialogResult* dr = ofd.*ShowDialog*();  if (dr == *DialogResult*.*OK*)  {  HyfTools.ImportData(ofd.*FileName*);  //这是刷新dategridview  string sql = "select \* from T\_UserInfo order by uid desc";  BindData(sql);  }  } |

## 导出xml文件

### 创建导出xml文件的方法

|  |
| --- |
| public static void ExportXML(string tableName,string path)  {  string sql = "select \* from " + tableName;  //创建文档对象  *XDocument* document = new *XDocument*();  //设置文档声明  document.*Declaration* = new *XDeclaration*("1.0", "utf-8", "yes");  //添加根节点  *XElement* root = new *XElement*("T" + tableName);  document.*Add*(root);  //从数据库读取数据  *DataTable* dt = SqlHelper.getDataTalbe(sql);  foreach (*DataRow* item in dt.*Rows*)  { //设置元素节点  *XElement* element = new *XElement*(tableName);  //设置属性  element.*SetAttributeValue*("id", item[0]);  for (int i = 1; i < dt.*Columns*.*Count*; i++)  {  element.*SetElementValue*(dt.*Columns*[i].*ColumnName*, item[i]);  }  root.*Add*(element);  }    document.*Save*(path);    } |

### 创建菜单项



### 点击导出XML导出为xml文件

|  |
| --- |
| private void 导出XMLToolStripMenuItem\_Click(object sender, *EventArgs* e)  {  SaveFileDialog sdf = new SaveFileDialog();  sdf.Filter = "xml文件(\*.xml)|\*.xml|所有文件(\*.\*)|\*.\*";  sdf.DefaultExt = "xml";  sdf.AddExtension = true;  *DialogResult* dr = sdf.*ShowDialog*();  if (dr == *DialogResult*.*OK*)  {  string filename = sdf.FileName;  HyfTools.ExportXML("T\_UserInfo", filename);  }  } |

# 总结

两大部分：数据提供程序Data Provider，数据集DataSet

操作SqlServer数据库：SqlConnection SqlCommand SqlDataReader SqlAdapter

数据集：DataSet DataTable DataRow DataColumn

定义一个静态类：SqlHelper，封装所有的数据库操作

Connection:连接,Open(),Close(),Dispose()

配置连接字符串，推荐写到配置文件中

System.Configuration.ConfigurationManager.ConnectionStrings[name].ConnectionString;

Command：执行命令对象

CommandText

CommandType

ExecuteNonQuery():执行非查询语句

ExecuteScalar():执行查询，返回首行首列的值

ExecuteReader():执行查询，返回DataReader，需要保持连接才可以访问

DataReader：用于读取查询到的数据

Read()

Get\*\*\*(列索引)

DataAdapter：进行断开式的访问

Fill()：执行查询，将结果放到DataSet中

Update()：将DataTable中的信息更新到真实的数据库中

DataSet:

Tables

DataTable：内存中的二维的数据表，对应着数据库中的某个数据表

Rows:DataRow

Columns:DataColumn

将sql语句进行参数化，防止sql注入

# sqlhelper

|  |
| --- |
| using System;  using System.Collections.Generic;  using System.Linq;  using System.Text;  using System.Configuration;  using System.Data;  using System.Data.SqlClient;  namespace \_01通过SqlHelper实现登录  {  public static class SqlHelper  {  //连接字符串  private static readonly string connStr = ConfigurationManager.ConnectionStrings["mssqlserver"].ConnectionString;  //封装常用方法  //1.执行insert/delete/update的方法  public static int ExecuteNonQuery(string sql, CommandType cmdType, params SqlParameter[] pms)  {  using (SqlConnection con = new SqlConnection(connStr))  {  using (SqlCommand cmd = new SqlCommand(sql, con))  {  //设置当前执行的是存储过程还是带参数的Sql语句  cmd.CommandType = cmdType;  if (pms != null)  {  cmd.Parameters.AddRange(pms);  }  con.Open();  return cmd.ExecuteNonQuery();  }  }  }  //2.执行返回单个值的方法  public static object ExecuteScalar(string sql, CommandType cmdType, params SqlParameter[] pms)  {  using (SqlConnection con = new SqlConnection(connStr))  {  using (SqlCommand cmd = new SqlCommand(sql, con))  {  //设置当前执行的是存储过程还是带参数的Sql语句  cmd.CommandType = cmdType;  if (pms != null)  {  cmd.Parameters.AddRange(pms);  }  con.Open();  return cmd.ExecuteScalar();  }  }  }  //3.返回SqlDataReader的方法  public static SqlDataReader ExecuteReader(string sql, CommandType cmdType, params SqlParameter[] pms)  {  SqlConnection con = new SqlConnection(connStr);  using (SqlCommand cmd = new SqlCommand(sql, con))  {  cmd.CommandType = cmdType;  if (pms != null)  {  cmd.Parameters.AddRange(pms);  }  try  {  con.Open();  return cmd.ExecuteReader(CommandBehavior.CloseConnection);  }  catch  {  con.Close();  con.Dispose();  throw;  }  }  }  //4.返回DataTable  public static DataTable ExecuteDataTable(string sql, CommandType cmdType, params SqlParameter[] pms)  {  DataTable dt = new DataTable();  using (SqlDataAdapter adapter = new SqlDataAdapter(sql, connStr))  {  adapter.SelectCommand.CommandType = cmdType;  if (pms != null)  {  adapter.SelectCommand.Parameters.AddRange(pms);  }  adapter.Fill(dt);  return dt;  }  }  }  } |