using System;

using System.Collections.Generic;

using System.IO;

using System.Linq;

using System.Reflection;

using System.Text;

using System.Threading.Tasks;

using System.Xml.Linq;

namespace IXmlDB

{

public class XmlDbSet<TEntity> where TEntity : class, new()

{

//构造函数

public XmlDbSet(string path = "XmlDb.xml", string nodeName = "Node", string rootName = "Root")

{

defaultProperty = "Id";

if (Connect(path, nodeName, rootName))

{

//链接成功，载入数据

classList = new List<TEntity>();

foreach (var item in AllNodes)

{

classList.Add(XElementToClass(item));

}

}

else

{

throw new Exception("连接数据文件失败!");

}

}

#region 私有字段

private string xmlFilePath;

private string[] xmlProperties;

private string nodeName;

private string defaultProperty;

private XElement xmlRoot;

private List<TEntity> classList;

#endregion

#region 私有属性

//获取新的Id

private int NewId

{

get

{

if (classList.Count() > 0)

{

var lastNode = classList.Select(m => Convert.ToInt32(ReflectionGetProperty(m, defaultProperty)));

return (lastNode.Max() + 1);

}

else

{

return 1;

}

}

}

//获取所有子节点

private IEnumerable<XElement> AllNodes

{

get

{

return xmlRoot.Elements(nodeName);

}

}

#endregion

#region 公有方法

//添加单个实例

public TEntity Add(TEntity entity)

{

if (ReflectionGetProperty(entity, defaultProperty) == "")

{

ReflectionSetProperty(entity, defaultProperty, NewId.ToString());

classList.Add(entity);

}

return entity;

}

//添加多个实例

public IEnumerable<TEntity> AddRange(IEnumerable<TEntity> entities)

{

int id = NewId;

foreach (var entity in entities)

{

if (ReflectionGetProperty(entity, defaultProperty) == "")

{

ReflectionSetProperty(entity, defaultProperty, id.ToString());

classList.Add(entity);

id++;

}

}

return entities;

}

//确定序列中所有元素是否满足条件

public bool All(Func<TEntity, bool> predicate)

{

return classList.All(predicate);

}

//确定序列中是否包含任何元素

public bool Any()

{

return classList.Any();

}

//确定序列中任何元素是否满足条件

public bool Any(Func<TEntity, bool> predicate)

{

return classList.Any(predicate);

}

//从序列中移除所有元素

public void Clear()

{

classList.Clear();

}

//链接两个序列

public IEnumerable<TEntity> Concat(IEnumerable<TEntity> second)

{

return classList.Concat(second);

}

//获取序列中包含的元素数

public int Count()

{

return classList.Count;

}

//返回一个数字，表示在指定序列中满足条件的元素的数量

public int Count(Func<TEntity, bool> predicate)

{

return classList.Count(predicate);

}

//确定指定元素是否在序列中

public bool Contains(TEntity item)

{

return classList.Contains(item);

}

//返回序列中指定索引出的元素

public TEntity ElementAt(int index)

{

return classList.ElementAt(index);

}

//返回序列中指定索引出的元素，如果超出指定范围，则返回默认值（null）

public TEntity ElementAtOrDefault(int index)

{

return classList.ElementAtOrDefault(index);

}

//确定序列中是否包含与指定谓词所定义的条件相匹配的元素

public bool Exists(Predicate<TEntity> match)

{

return classList.Exists(match);

}

//通过使用默认比较器对值进行比较生成两个序列的差集

public IEnumerable<TEntity> Equals(IEnumerable<TEntity> second)

{

return classList.Except(second);

}

//返回序列中满足指定条件的第一个元素;若序列中不包含元素，则返回默认值

public TEntity FirstOrDefault(Func<TEntity, bool> predicate)

{

return classList.FirstOrDefault(predicate);

}

//搜索与指定谓词相匹配的元素，并返回第一个匹配的元素

public TEntity Find(Predicate<TEntity> match)

{

return classList.Find(match);

}

//搜索与指定谓词相匹配的所有元素

public IEnumerable<TEntity> FindAll(Predicate<TEntity> match)

{

return classList.FindAll(match);

}

//搜索与指定谓词相匹配的元素，并返回第一个匹配的元素的从零开始的索引

public int FindIndex(Predicate<TEntity> match)

{

return classList.FindIndex(match);

}

//搜索与指定谓词相匹配的元素，并返回最后一个匹配的元素

public TEntity FindLast(Predicate<TEntity> match)

{

return classList.FindLast(match);

}

//搜索与指定谓词相匹配的元素，并返回最后一个一个匹配的元素的从零开始的索引

public int FindLastIndex(Predicate<TEntity> match)

{

return classList.FindLastIndex(match);

}

//返回序列中的第一个元素

public TEntity First()

{

return classList.First();

}

//返回序列中满足指定条件的第一个元素

public TEntity First(Func<TEntity, bool> predicate)

{

return classList.First(predicate);

}

//返回序列中的第一个元素;若序列中不包含元素，则返回默认值

public TEntity FirstOrDefault()

{

return classList.FirstOrDefault();

}

//对序列中的每一个元素执行指定操作

public void ForEach(Action<TEntity> action)

{

classList.ForEach(action);

}

//搜索指定对象，并返回序列中第一个匹配项的从零开始的索引

public int IndexOf(TEntity item)

{

return classList.IndexOf(item);

}

//将元素插入到序列中指定的索引处（索引从0开始）

public void Insert(int index, TEntity item)

{

if (ReflectionGetProperty(item, defaultProperty) == "")

{

ReflectionSetProperty(item, defaultProperty, NewId.ToString());

}

classList.Insert(index, item);

}

//将集合中的元素插入到序列中指定的索引处（索引从0开始）

public void InsertRange(int index, IEnumerable<TEntity> collection)

{

int id = NewId;

foreach (var item in collection)

{

if (ReflectionGetProperty(item, defaultProperty) == "")

{

ReflectionSetProperty(item, defaultProperty, id.ToString());

id++;

}

}

classList.InsertRange(index, collection);

}

//返回序列中的最后一个元素

public TEntity Last()

{

return classList.Last();

}

//返回序列中满足指定条件的最后一个元素

public TEntity Last(Func<TEntity, bool> predicate)

{

return classList.Last(predicate);

}

//搜索指定对象，并返回序列中第一个匹配项的从零开始的索引

public int LastIndexOf(TEntity item)

{

return classList.LastIndexOf(item);

}

//返回序列中的最后一个元素;若序列中不包含元素，则返回默认值

public TEntity LastOrDefault()

{

return classList.LastOrDefault();

}

//调用泛型序列的每一个元素上的转换函数并返回最大结果值

public string Max(Func<TEntity, string> selector)

{

return classList.Max(selector);

}

//调用泛型序列的每一个元素上的转换函数并返回最小结果值

public string Min(Func<TEntity, string> selector)

{

return classList.Min(selector);

}

//根据键按升序对序列的元素排序

public void OrderBy(Func<TEntity, string> keySelector)

{

classList = classList.OrderBy(keySelector).ToList();

}

//根据键按降序对序列的元素排序

public void OrderByDescending(Func<TEntity, string> keySelector)

{

classList = classList.OrderByDescending(keySelector).ToList();

}

//将整个序列中元素顺序逆转

public void Reverse()

{

classList.Reverse();

}

//从classLIst中移除特定对象的第一个匹配项

public TEntity Remove(TEntity entity)

{

classList.Remove(entity);

return entity;

}

//从classLIst中移除一组对象的第一个匹配项

public IEnumerable<TEntity> RemoveRange(IEnumerable<TEntity> entities)

{

foreach (var entity in entities)

{

classList.Remove(entity);

}

return entities;

}

//删除指定谓词所定义的条件匹配的所有元素

public int RemoveAll(Predicate<TEntity> match)

{

return classList.RemoveAll(match);

}

//删除指定索引出的元素

public void RemoveAt(int index)

{

classList.RemoveAt(index);

}

//保存更改(返回值:表示重写以及删除的实例个数)

public void SaveChanges()

{

xmlRoot.RemoveAll();

foreach (var item in classList)

{

xmlRoot.Add(ClassToXElement(item));

};

xmlRoot.Save(xmlFilePath);

}

//保存更改(返回值:表示重写的实例个数)

public int SaveChanges(TEntity entity)

{

if (entity != null)

{

if (ClassModToXElement(entity))

{

xmlRoot.Save(xmlFilePath);

return 1;

}

else

{

return 0;

}

}

else

{

return 0;

}

}

//保存更改(返回值:表示重写实例个数)

public int SaveChanges(IEnumerable<TEntity> entities)

{

if (entities != null)

{

int count = 0;

foreach (var entity in entities)

{

if (ClassModToXElement(entity))

{

count++;

}

}

if (count > 0)

{

xmlRoot.Save(xmlFilePath);

}

return count;

}

else

{

return 0;

}

}

//返回序列中唯一满足条件的元素;如果这类元素不存在，则返回默认值；如果存在多个元素满足条件，此方法将引发异常

public TEntity SingleOrDefault(Func<TEntity, bool> predicate)

{

return classList.SingleOrDefault(predicate);

}

//将序列中的每个元素投影到新表中

public IEnumerable<TResult> Select<TResult>(Func<TEntity, TResult> predicate)

{

return classList.Select(predicate);

}

//将序列中的每个元素投影到IEnumerable<out T> 并将结果序列合并为一个序列

public IEnumerable<TResult> SelectMany<TResult>(Func<TEntity, IEnumerable<TResult>> selector)

{

return classList.SelectMany(selector);

}

//跳过序列中指定数量的元素，然后返回剩余元素

public IEnumerable<TEntity> Skip(int index)

{

return classList.Skip(index);

}

//只要满足指定的条件，就跳过序列中的元素，然后返回剩余元素

public IEnumerable<TEntity> SkipWhile(Func<TEntity, bool> predicate)

{

return classList.SkipWhile(predicate);

}

//计算int值序列的和，这些值是通过对输入序列中的每一个元素调用转换函数得到的

public int Sum(Func<TEntity, int> selector)

{

return classList.Sum(selector);

}

public long Sum(Func<TEntity, long> selector)

{

return classList.Sum(selector);

}

public float Sum(Func<TEntity, float> selector)

{

return classList.Sum(selector);

}

public double Sum(Func<TEntity, double> selector)

{

return classList.Sum(selector);

}

public decimal Sum(Func<TEntity, decimal> selector)

{

return classList.Sum(selector);

}

//从序列的开头返回指定数量的连续元素

public IEnumerable<TEntity> Take(int index)

{

return classList.Take(index);

}

//只要满足指定条件，就会返回序列中的元素

public IEnumerable<TEntity> TakeWhile(Func<TEntity, bool> predicate)

{

return classList.TakeWhile(predicate);

}

//确定是否序列中每一个元素都与指定的谓词所定义的条件相匹配

public bool TrueForAll(Predicate<TEntity> match)

{

return classList.TrueForAll(match);

}

//通过使用默认的相等比较器生成两个序列的并集

public IEnumerable<TEntity> Union(IEnumerable<TEntity> second)

{

return classList.Union(second);

}

//基于谓此筛选值序列

public IEnumerable<TEntity> Where(Func<TEntity, bool> predicate)

{

return classList.Where(predicate);

}

#endregion

#region 私有方法

//连接数据文件

private bool Connect(string path, string nodeName, string rootName)

{

try

{

//检查参数是否为null或为空字符串

if (string.IsNullOrWhiteSpace(path) || string.IsNullOrWhiteSpace(nodeName) || string.IsNullOrWhiteSpace(rootName))

{

return false;

}

//匹配xml文件路径

if (path.IndexOf("\\") == -1)

{

return false;

//path = ConfigurationManager.ConnectionStrings["xmlPath"].ConnectionString + "\\App\_Data\\" + path;

}

//if (!Regex.IsMatch(path, @"^(?<fpath>([a-zA-Z]:\\)([\s\.\-\w]+\\)\*)(?<fname>[\w]+.[\w]+)") || path.Length < 5 || path.Substring(path.Length - 4).ToLower() != ".xml")

//{

// return false;

//}

//检查属性是否合法

TEntity objClass = new TEntity();

PropertyInfo[] infos = objClass.GetType().GetProperties();

if (infos.Length == 0 || infos.Count(m => m.Name == defaultProperty) == 0)

{

return false;

}

xmlProperties = new string[infos.Length];

int i = 0;

foreach (var info in infos)

{

if (string.IsNullOrWhiteSpace(info.Name) || infos.Count(m => m.Name == info.Name) > 1)

{

return false;

}

else

{

xmlProperties[i] = info.Name;

i++;

}

}

this.nodeName = nodeName;

xmlFilePath = path;

//判断xml文件是否存在，若不存在则创建

if (path.LastIndexOf("\\") > 0)

{

path = path.Substring(0, path.LastIndexOf("\\"));

}

else

{

path = "";

}

string quote = "\"";

if (path != "" && !Directory.Exists(path))

{

Directory.CreateDirectory(path);

var xmlFile = new StreamWriter(xmlFilePath);

xmlFile.WriteLine("<?xml version=" + quote + "1.0" + quote + " encoding=" + quote + "utf-8" + quote + "?>");

xmlFile.WriteLine("<" + rootName + ">");

xmlFile.WriteLine("</" + rootName + ">");

xmlFile.Close();

}

else

{

if (!File.Exists(xmlFilePath))

{

var xmlFile = new StreamWriter(xmlFilePath);

xmlFile.WriteLine("<?xml version=" + quote + "1.0" + quote + " encoding=" + quote + "utf-8" + quote + "?>");

xmlFile.WriteLine("<" + rootName + ">");

xmlFile.WriteLine("</" + rootName + ">");

xmlFile.Close();

}

}

xmlRoot = XElement.Load(xmlFilePath);//载入数据文件

//自检数据文件

if (NodesPropertiesIsValid())

{

return true;

}

else

{

throw new Exception("数据文件不完整或损坏!");

}

}

catch (Exception e)

{

throw e;

}

}

//检查节点属性是否合法

private bool NodePropertiesIsValid(XElement targetNode)

{

try

{

if (targetNode.Name.ToString() != nodeName)

{

return false;

}

for (int i = 0; i < xmlProperties.Length; i++)

{

if (targetNode.Element(xmlProperties[i]) == null)

{

return false;

}

}

return true;

}

catch

{

return false;

}

}

//检查整个xml文件属性和Id是否合法(加载自检)

private bool NodesPropertiesIsValid()

{

try

{

if (AllNodes.Count() == 0)

{

return true;

}

var strs = AllNodes.Select(m => m.Element(defaultProperty).Value).Distinct();

if (strs.Count() != AllNodes.Count() || AllNodes.Count(m => !NodePropertiesIsValid(m)) > 0)

{

return false;

}

else

{

return true;

}

}

catch

{

return false;

}

}

//将xml元素转化为对应对象(新实例)

private TEntity XElementToClass(XElement targetNode)

{

if (targetNode == null)

{

return null;

}

else

{

TEntity objClass = new TEntity();

for (int i = 0; i < xmlProperties.Length; i++)

{

ReflectionSetProperty(objClass, xmlProperties[i], targetNode.Element(xmlProperties[i]).Value);

}

return objClass;

}

}

//将对象转化为对应的xml元素新实例)

private XElement ClassToXElement(TEntity objClass)

{

if (objClass == null)

{

return null;

}

else

{

XElement newNode = new XElement(nodeName);

for (int i = 0; i < xmlProperties.Length; i++)

{

newNode.Add(new XElement(xmlProperties[i], ReflectionGetProperty(objClass, xmlProperties[i])));

}

return newNode;

}

}

//将对象的值传给对应的xml元素，或直接添加

//private void ClassSaveToXElement(TEntity objClass)

//{

// string id = ReflectionGetProperty(objClass, defaultProperty);

// var targetNode = AllNodes.SingleOrDefault(m => m.Element(defaultProperty).Value == id);

// if (targetNode != null)

// {

// for (int i = 0; i < xmlProperties.Length; i++)

// {

// targetNode.Element(xmlProperties[i]).Value = ReflectionGetProperty(objClass, xmlProperties[i]);

// }

// }

// else

// {

// xmlRoot.Add(ClassToXElement(objClass));

// }

//}

//将对象的值传给对应的xml元素

private bool ClassModToXElement(TEntity objClass)

{

string id = ReflectionGetProperty(objClass, defaultProperty);

var targetNode = AllNodes.SingleOrDefault(m => m.Element(defaultProperty).Value == id);

if (targetNode != null)

{

for (int i = 0; i < xmlProperties.Length; i++)

{

targetNode.Element(xmlProperties[i]).Value = ReflectionGetProperty(objClass, xmlProperties[i]);

}

return true;

}

else

{

return false;

}

}

////动态编译类

//private Assembly NewAssembly()

//{

// //创建编译器实例。

// CSharpCodeProvider provider = new CSharpCodeProvider();

// //设置编译参数。

// CompilerParameters paras = new CompilerParameters();

// paras.GenerateExecutable = false;

// paras.GenerateInMemory = true;

// //创建动态代码。

// StringBuilder classSource = new StringBuilder();

// classSource.Append("public class DynamicClass \n");

// classSource.Append("{\n");

// //创建属性。

// for (int i = 0; i < xmlProperties.Length; i++)

// {

// classSource.Append(" public string " + xmlProperties[i] + " { get; set; } \n");

// }

// classSource.Append("}");

// System.Diagnostics.Debug.WriteLine(classSource.ToString());

// //编译代码。

// CompilerResults result = provider.CompileAssemblyFromSource(paras, classSource.ToString());

// //获取编译后的程序集。

// Assembly assembly = result.CompiledAssembly;

// return assembly;

//}

//反射设置动态类的实例对象的指定的属性值

private void ReflectionSetProperty(TEntity objClass, string propertyName, string value)

{

objClass.GetType().GetProperty(propertyName, BindingFlags.Public | BindingFlags.Instance).SetValue(objClass, value ?? "", null);

}

//反射返回动态类的实例对象的指定的属性值

private string ReflectionGetProperty(TEntity objClass, string propertyName)

{

try

{

return objClass.GetType().GetProperty(propertyName, BindingFlags.Public | BindingFlags.Instance).GetValue(objClass, null).ToString();

}

catch

{

return "";

}

}

#endregion

}

}