## C#读书笔记

### 数据类型

static readonly ,const

static readonly :static readonly是在运行时计算出其值的

const:const的值是在编译期间确定的,编译后值直接存入dll

作用：在线程的同步中用到（锁 、互斥）

Public class P

{

const int A=B\*10;

const int B=10;

public static void Main(string[] args)

{

Console.WriteLine("A is {0},B is {1} ",A,B);

}

}

### 语句

Yield 和 foreach 集合，在feach中返回，进入下个迭代

### 数组

数组派生自 Array 类

Clone 浅复制

深复制

### 集合

数组和集合都支持 foreach 和 下标查找，但是 数组很多lambda表达式不支持，都在Ａrray静态方法中

而集合是实例方法能够支持。

集合的容量是会改变的所以集合不建议使用下标查找

Array.ForEach(chars,(x)=>{Console.WriteLine(x);});

### 类

### 控制反转

1. 利用程序集和类名构造实例

string assemblyName = main.Assembly; //所属程序集

string className = main.ClassName;//类名

Assembly ass = Assembly.Load(assemblyName);

Type pageType = ass.GetType(className);

BusinessBasePage businessPage = (BusinessBasePage)Activator.CreateInstance(pageType);

？程序集并没有引用BusinessBasePage 是哪里来的？// 误区，动态创建对象，在于定义一个方法用于生成某种类型的实例，

不是给个字符串就能生成实例，最后需要强制转换。

2.避免硬编码，通过unity构造类的实例

<configuration>

<configSections>

<section name="unity" type="Microsoft.Practices.Unity.Configuration.UnityConfigurationSection,Microsoft.Practices.Unity.Configuration"/>

</configSections>

<startup>

<supportedRuntime version="v4.0" sku=".NETFramework,Version=v4.5" />

</startup>

<unity xmlns="http://schemas.microsoft.com/practices/2010/unity">

<alias alias="IHuman" type="MyUnityTest.IHuman,MyUnity"></alias>

**<!--type(命名空间+类名，程序集)-->**

<alias alias="Man" type="MyUnityTest.Man,MyUnity"></alias>

<alias alias="People" type="MyUnityTest.People,MyUnity" />

<container name="myunity">

<register type="IHuman" mapTo="Man" name="hu">

<**constructor**>

<**param** name="p">

<**dependency** type="People"></dependency>

</param>

</constructor>

</register>

</container>

</unity>

</configuration>

如果不指定构造函数，系统用无参构造函数控制反转

### 继承

见用户自定义异常。

### 委托

委托首先解决的是方法作为参数调用的问题，与事件相似可以主动调用其他类的委托(不需要关心方法的具体实现)

间接调用一系列事件。

1. 委托的回调：如果想把方法作为参数传递，就定义一个委托类型的形参。

Fun(Action<int> a)

{

a(10); // 实参是方法内部通过一定方式获得的，需要的是a 这个方法

}

1. 委托链：执行多个方法

(此处思考多人订阅同一个事件)

1. 委托和事件结合

事件是种特殊的委托

1. 把一个外部类的方法传递到类的内部，可以在类的内部定义一个委托事件

(这就是用委托不直接使用方法的原因，虽然调用方法f1(),f2()也能够调用方法),可以通过事件的参数将变量从类的内部传到外部去，供外部使用。这种方式是从类的内部像外部传递。A类-->B类-->A类

5.把方法委托到线程执行

6.委托执行一个匿名方法即Lambda表达式

Lambda表达式实质是匿名方法，可以和代理结合，但是本身不是代理。如果参数是Action

New Action(()=>{...})，可以这样传匿名方法，还可以这样写Action a = () => { };。

最好是有名称的，此时不需要确定实参，目的是传递方法，实参在方法内部给定。

Expression 表达式树的作用将传递过来的方法作再次的修改，可以改变方法结构，让方法和变量一样能够被操作。

可以将Lambda表达式传递给表达式树形参。具体实例等看到再记录

1. 委托的协变和逆变

解决了子类，父类类型的相互转换

协变：委托方法的返回值类型直接或间接地继承自委托签名的返回值类型。**↑返回值**

逆变：委托签名的中参数类型继承自委托方法的参数类型。 **↓参数**

从概念可以看出不管是协变还是逆变，都是**子类转到父类**。与向上转型向下转型是两个概念。

class Program

{

static void Main(string[] args)

{

Man m = new Man() { MyProperty=1};

Func<Human> h = () => { return m; }; //**Func<Human> h = () =>m; 等价**

**// 如果此处m换成返回方法，看方法是Void ，还是其他数据类型**

Console.WriteLine(h().MyProperty);

Action<Man> a = new Action<Man>(fun);

a(m);//此处注意m为Man类型，因为已经<Man>泛型约束了

Console.Read();

}

public static void fun(Human m)

{

Console.WriteLine(m.MyProperty);

}

}

public class Human

{

public int MyProperty { get; set; }

}

public class Man:Human

{

public override string ToString()

{

return "123";

}

}

------------------------------------------------------------------------------------------------------------------------------------------------------------------

1. ☆ 委托和接口结合此用法比协变，逆变更重要,客户端只依赖接口，不依赖具体的类。

DI依赖注入也是能够达到同样的效果。两者的共同点是调用客户端只需要认识接口，不需要认识具体实现类。

构造函数的注入（至今还未知道怎么在客户端对注入的类赋值，现在客户端能做的就是调用方法。所以构造函数的注入，现在最多注入接口实例）

class Program

{

static void Main(string[] args)

{

Program P = new Program();

**P.fun(x=>x.run("abc"));**

**// 指明了我要调用的是接口的什么方法，实数是什么**

Console.Read();

}

public void fun(Action<IOring> ac)

{

**//ac = x => x.run("xxx");**

**MyClass my=new MyClass(); // 具体实现接口的类**

**ac(my);**

// 此处和协变，逆变无关

}

}

public interface IOring

{

void run(object a);

}

class MyClass:IOring

{

public void run(object a)

{

Console.WriteLine(a);

}

}

------------------------------------------------------------------------------------------------------------------------------------------------------------------

1. 此处的代理和webservice 生成的客户端代理是两回事

class Program

{

static void Main(string[] args)

{

XiaoDong boy = new XiaoDong() { Name = "小东" };

Girl girl = new Girl() {Name="娇娇"};// 可以将Girl隐藏在 Proxy 类里面

Proxy proxy = new Proxy(boy);

proxy.GiveFlows(girl);

Console.Read();

}

}

class XiaoDong:Boy

{

}

此处小东并不认识娇娇，但是代理类认识Boy , Girl 。客户的类继承Boy 后就可以使用代理类。

### 事件

事件最主要的是事件的订阅者和事件的发布者，事件的发布者发布事件的功能。事件的订阅者通过委托链定义多个事件。其他对象也可以订阅。

1. A. public partial class Form1 : Form

{

**public** **EventHandler btn1;**

**public EventHandler btn2; // 可以提供给外部调用**

public Form1()

{

InitializeComponent();

}

private void Form1\_Load(object sender, EventArgs e)

{

**btn1 = new EventHandler(butt1);**

**btn2 = new EventHandler(butt2);**

**this.button3.Click += btn1;**

**this.button3.Click += btn2;**

**this.button4.Click += btn2;**

}

private void butt1(object sender, EventArgs e)

{

this.button1.Visible = false;

}

private void butt2(object sender, EventArgs e)

{

this.button2.Visible = false;

}

}

---------------------------------------------------------------------------------------------------------

B. //btn1+=butt1; 订阅者和发布者都能使用 +=

//btn1+= butt2;

//btn2+= butt2;

//this.button3.Click += btn1;

//this.button4.Click += btn2;

C. //this.button3.Click += butt1; 直接调用

**public EventHandler 同样可以供外部调用，内部触发委托** 难道事件等于委托？？迷茫了

Form2窗口

private void button1\_Click(object sender, EventArgs e)

{

this.F = new Form1();

this.F.Show();

this.button2.Click += this.F.btn1;

this.button3.Click += this.F.btn2;

}

如果是事件btn1 ，事件无法在类外部被调用，即使为Public 也无效。如果是委托内外部内部都能调用。

如A,B（B属于A）窗体，A中想改变B 用委托，B中想改变A用事件。委托可以用=或者+= -= ，修改委托链中的事件

但是事件只能用+= -= ，这样防止用 =时将其他方法覆盖。

一个**EventHandler （注意该类是一个委托，不是事件）对应一个事件订阅者，事件发布者通过委托链连接委托EventHandler 。**

1. 见委托4的功能，区别很大
2. 事件参数

( Object sender ，EventArgs e ) , 在自定义事件中sender 可以自己指定为声明该事件的类，也可以是类内部自己启动事件时传递的其他类，e 为实现了EventArgs的子类。

传统的 sender，e 区别就是引发事件的对象，和额外参数。

WPF 路由事件中 sender 为捕捉事件的类，也就是捕捉事件的类绑定在的对象上比如Grid，e.source 为 真正发出事件的类 比如 Grid 内部的 Button 。

### 用户自定义异常

1. 继承Exception 类
2. 在派生类中指定基类的构造函数。

class \_UserException : Exception

{

public string ex\_message = string.Empty;

public \_UserException(): base()

{

}

public \_UserException(string message)

: base(message)

{

this.ex\_message = "用户自定义异常：" + message;

}

}

### 验证

### 线程

线程同步

线程异步

1. private void button2\_Click(object sender, EventArgs e)

{

//public delegate void run();

//public delegate void mydele(string s);

**mydele dele = new mydele(temp);**

**dele.BeginInvoke("开启线程异步计算...", null, null);**

// Delegate.BeginInvoke 会从线程池取一个线程异步执行,Delegate.InInvoke运行在当前线程

}

public void temp(string s)

{

MessageBox.Show(s);

for (int i = 0; i < 2000; i++)

{

for (int j = 0; j < 100000; j++)

{

Count += j;

}

}

**run r = new run(fun);**

**this.BeginInvoke(r, null);**

MessageBox.Show("123");

// Control.BeginInvoke ,Control.InInvoke 都只是运行在UI线程上，

// Control.BeginInvoke 异步指的是MessageBox.Show("123")会BeginInvoke 后立刻执行

}

private void fun()

{

this.textBox3.Text = Count.ToString();

}

--------------------------------------------------------------------------------------------------------------

2. private void button2\_Click(object sender, EventArgs e)

{

//public delegate void mydele(string s);

**mydele dele = new mydele(temp);**

**dele.BeginInvoke("开启线程异步计算.....",fun,null);**

}

public void temp(string s)

{

MessageBox.Show(s);

for (int i = 0; i < 2000; i++)

{

for (int j = 0; j < 100000; j++)

{

Count += j;

}

}

}

private void fun(**IAsyncResult** ar)

{

**Action a = new Action(() =>**

**{**

**this.textBox3.Text = Count.ToString();**

**});**

**this.BeginInvoke(a, null);**

}

**MarshalByRefObject**

跨应用程序域边界

### Async Await Task

Asnyc 和 await 是成对使用的。被标记为asnyc 方法的内部必须有 await ,不然此方法不能被异步执行。

await 时，如果被await 的Task方法还没有执行完，那么控制权交给主线程，同时记录下await 的点，此时主线程

执行其他代码，当被 await 标记的 Task 方法执行完之后（其他线程中执行的），主线程回来，执行await 后面的代码 。

返回值有 Void Task Task<T> , Void Task 为空返回值，区别在于 Task 类型可以被接受并且ContinueWith 继续其他任务。

private async Task read(int A) { } 返回空值

private async Task<int> read(int A) { return A }

**Task**

任务的意义：在一个总任务下挂多个子任务

Task 开启线程，此时会从线程池取一个线程用来执行代码。而async 标记的方法在主线程运行。

与 ThreaPool 的区别在于，Task 运行的线程各个环节的状态能够控制。Task 在NET版本中出现的时间比 async 早。

CancellationTokenSource tokenSource;

CancellationToken token;

tokenSource = new CancellationTokenSource();

**token = tokenSource.Token;**

task = Task.Run(() => { Thread.Sleep(2000); }, token);

await task;

this.label2.Text = "第1阶段";

await task.ContinueWith((x) => { Thread.Sleep(3000); }, **token**);

this.label2.Text += "——第2阶段";

await task.ContinueWith((x) => { Thread.Sleep(3000); }, **token**);

this.label2.Text += "——第3阶段";

耗时要写在Task 或者 Task<T> 返回值得方法中。在第一阶段之后取消任务，因为第二阶段已经开始了，所以第3阶段才真正停止任务。

Task.Delay(); 任务等待

### 接口

见 委托 第8点委托

协变

逆变

### 泛型

泛型方法

Myprint.Print<int>(20) 等价 Myprint.Print(20)

IAsyncProxy<ICommonService> asyncProxy = await Task.Run(()=>WcfClientProxy.CreateAsyncProxy<ICommonService>

(c=>c.SetEndpoint(CreateBinding(),newEndpointAddress(ClientHelper.GetEndpointAddress(ServiceCode.CommonService));

此处 Run 简写，原型Run<IAsyncProxy<ICommonService> >

CreateAsyncProxy<TServiceInterface>(Action<IRetryingProxyConfigurator> configurator) 构造实例，

此处构造channel 信道

Lambda 中 c 为 IRetryingProxyConfigurator 类型

void SetEndpoint(Binding binding, EndpointAddress endpointAddress); 指明绑定和总结点

有个问题：SetEndpoint为什么是实例，见 8.委托和接口结合

### 流和文件

TeaxReader ，XmlReader 和 BinaryReader 格式的读写器

TeaxReader 是 StringRrader 和 StreamReader 的抽象父类

XmlReader 是 XmlTextReader 和 XmlNodeReader 的抽象父类

从两个子类的构造函数和方法看出StringRrader不能操作流（很奇怪）

其他几个格式的读写器可以

File FileInfo 能够提供 FileStream 流供读写器读写

Sw.Flush() 将缓冲区数据写入基础流

Sw.Close() 关闭当前sw对象，关闭基础流同时写入缓存区数据

string str = "ABCDEFG";

byte[] strByte = System.Text.Encoding.Default.GetBytes(str);

FileInfo file = new FileInfo(@"D://2.txt");

**FileStream fs = file.Open(FileMode.OpenOrCreate,****FileAccess.ReadWrite);**

StreamWriter sw = new StreamWriter(fs);

sw.Write(str);

**sw.Flush();**  //这一步才将缓冲区的数据推送到基础流

char[] ch = new char[strByte.Length];

fs.Position = 0;

StreamReader sr = new StreamReader(fs);

sr.Read(ch, 0, strByte.Length);

Console.WriteLine(ch);

Console.Read();

sw.Close();

sr.Close();

使用 **FileAccess.ReadWrite** 模式下支持文件的读写，最后结束了再将流关掉，sw.WriteLine() 支持写换行。

FileStream 有自己的缓冲区  private byte[] \_buffer;

public override void Flush()  
         {  
             this.Flush(false); // 已经重写  
         }

NetWorkStream 没有缓冲区 无private byte[] \_buffer;

public override void Flush()  
        {

没有重写为空  
       }

BufferedStream 为其他流构建缓冲区 private byte[] \_buffer;

public override void Flush()  
         {

重写了父类的方法

}

MemoryStream 开辟内存一块空间 有private byte[] \_buffer;

public override void Flush()  
         {

没有重写为空  
        }

**用各种基础流自身的读写，一次读写一个比特，但是可以设置读写比特个数**

**读写器一个可以读写几个比特（int ,double）或者字符，然后推送到基础流。**

读写器和序列化默认编码方式是UTF8

### Encoding 编码和解码抽象类

字符串或 char[] 转byte数组 System.Text.Encoding.ASCII.GetBytes(s) 编码 // 从字符或者字节到比特

byte数组转字符串 System.Text.Encoding.ASCII.GetString(b) 解码 // 从比特读取到其他形式

char[] chars = Encoding.ASCII.GetChars(bytes);

字符串转char数组 s.ToCharArray()

char数组转字符串 new( ch ) // char[] ch=new char[10];

//char[] cChar = new char[5] { 'a', 'b', 'c', 'd', 'e' };

//string sData = "abc";

//byte[] byteData = Encoding.Default.GetBytes(cChar);

//byte[] stringData = Encoding.Default.GetBytes(sData);

//Console.WriteLine(byteData[0]);

//Console.WriteLine(stringData[0]);

FileInfo f = new FileInfo("D://3.b");

BinaryWriter bw = new BinaryWriter(f.Open(FileMode.OpenOrCreate));

bw.Write(97);

bw.Write('a');

bw.Write(98);

bw.Write(99);

bw.Write(100);

bw.Close();

Console.Read();

字符会转成 ASII 编码后的二进制码存储 97 0x61 a 等价

序列化 默认 UTF-8 编码？

### 配置文件的读取

1. <appSettings>
2. <connectionStrings>
3. <configsections>

### XML文件的读取

### 事务

1. c# 中的事务(分布式事务？)

TransactionScope

TransactionScope实现了IDispose接口。除非显示调用ts.Complete()方法。否则，系统不会自动提交这个事务。如果在代码运行退出这个block后，还未调用Complete()，那么事务自动回滚了。

1. SQL Server 中的事务

### 简单加密

### 递归菜单[Telerik]

### GridView与分页[Telerik]

1. 自定义分页控件的关键点事，封装好的分页控件，只对外开放页码改变事件，供页码改变时Gridview 绑定当前页码，外部需要初始化：

总数据条数 ItemCount

每页显示条数 PageSize

当前页码索引 PageIndex 这3个属性可以封装成一个类传到自定义控件的内部去

自定义控件内部需要实现：

public int **PageCount** { get; set; }

public int **TotalItemCount** { get; set; }

public int **PageNumber** { get; set; }

public bool **HasPreviousPage** { get; set; }

public bool **HasNextPage** { get; set; }

public bool **IsFirstPage** { get; set; }

public bool **IsLastPage** { get; set; }

PageCount = (int)Math.Ceiling(totalItemCount / (float)pageSize);

PageNumber = pageNumber;

TotalItemCount = totalItemCount;

HasPreviousPage = pageNumber > 1;

HasNextPage = pageNumber < PageCount;

private void pagePlanList\_PageChanged(object sender, EventArgs e)

{

\_paging.PageNumber = pagePlanList.CurrentPage;

BindWarehouseTranHeadList();

}

var data = ctx.Set<T>().Where(filter.Compile());

int totalRecords = data.Count();

result.Entities = data.AsQueryable()

.OrderBy(paging.SortBy + " " + paging.SortDirection)

.Skip((paging.PageNumber - 1) \* paging.PageSize)

.Take(paging.PageSize)

.AsEnumerable();

## WPF

## WCF

容器

public static class ServiceTestHelper

{

private static IUnityContainer \_aisinoMesContainer;

public static IUnityContainer AisinoMesContainer

{

get { return \_aisinoMesContainer; }

set { \_aisinoMesContainer = value; }

}

public static void GetUnitContainer()

{

**AisinoMesContainer = new UnityContainer();**

**UnityConfigurationSection sysManagerSection = (UnityConfigurationSection)GetSysManagerUnityConfig().GetSection("MesManagerUnity");**

sysManagerSection.Configure(AisinoMesContainer, "AisinoMesContainer");

//其余配置文件可以陆续添加

}

public static Configuration GetSysManagerUnityConfig()

{

ExeConfigurationFileMap fileMap = new ExeConfigurationFileMap();

fileMap.ExeConfigFilename = AppDomain.CurrentDomain.BaseDirectory + @"\MesManagerUnity.config";

Configuration config = ConfigurationManager.OpenMappedExeConfiguration(fileMap, ConfigurationUserLevel.None);

return config;

}

public static T GetServiceInstance<T>(string name)

{

T aisinoServiceInstance = AisinoMesContainer.Resolve<T>(name);

return aisinoServiceInstance;

}

public \_UserException(string message,OtherException ex):base(message,ex)

{

}

此处可以catch一段代码抓取OtherException ( 可以为 OtherException 1，OtherException 2 )，

然后统一throw UserException供外部抓取