## 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

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

### 集合

### 类

### 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
2. 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)

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。

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;

}

}

2. 见委托4的功能

### 线程

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运行在当前线程上

}

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);**

}

### 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

### 配置文件的读取

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

### XML文件的读取

### 控制反转

避免硬编码，通过字符串形式构造类的实例

<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

### 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;

}

}

## WPF 读书笔记

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