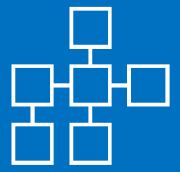
C#高级

郑贵锋

课程目标

通过本次课程，理解类的重要概念和相关知识，理解C#中高级类型（例如可空类型、泛型等），Lambda表达式和LINQ的原理和使用。



大纲

1. 类
2. 高级类型
3. Lambda表达式
4. LINQ

类

类声明

```
//声明非静态类
class NoStaticClass { }
//声明静态类[表示该类能实例化，只能包含静态成员]
static class StaticClass { }
//声明抽象类[该类不可用new直接创建对象]
abstract class AbstractClass { }
//声明封闭类[表示该类不可被继承]
sealed class SealedClass { }

//abstract和sealed不可同时修饰一个类
//static类不可和abstract或sealed同时修饰一个类
```

成员修饰符

```
class Test
{
    //public表示在任何地方都可以访问publicInt字段
    public int publicInt = 0;
    //private表示只能在类内部访问privateInt字段[默认是此项]
    private int privateInt = 0;
    //protected表示在类内部或者Test的子类可访问protectedInt
    protected int protectedInt = 0;
    //在程序集内部可访问，比public低，比protected高
    internal int internalInt=0;
    //protected和internal的合成
    protected internal int protectedInternalInt=0;
}
```

创建类的实例（对象）

```
//声明一个类型
class NoStaticClass {
    public NoStaticClass() { }
    public NoStaticClass(int value) { }
}
public class Test
{
    static void Main()
    {
        //使用new操作符创建一个类的实例（对象）
        //此操作会调用类的无参构造函数
        NoStaticClass myClass1 = new NoStaticClass();
        //调用类的有参构造函数
        NoStaticClass myClass2 = new NoStaticClass(2);
    }
}
```

字段（一）

- 字段是与类或类的实例关联的变量。
- 使用 static 修饰符声明的字段定义了一个静态字段 (static field)。一个静态字段只标识一个存储位置。无论对一个类创建多少个实例，它的静态字段永远都只有一个副本。
- 不使用 static 修饰符声明的字段定义了一个实例字段 (instance field)。类的每个实例都为该类的所有实例字段包含一个单独副本。
- 字段的初始化是在构造函数里面完成的，实例字段在实例构造器内初始化；静态字段在静态构造器初始化。

字段（二）

```
public class Color
{
    //静态字段[通过类型访问]
    public static readonly Color Black = new Color(0, 0, 0);
    //实例字段[通过类型的实例访问]
    private byte r, g, b;
    public Color(byte r, byte g, byte b)
    {
        this.r = r;
        this.g = g;
        this.b = b;
    }
}
```

方法（一）

- 方法 (method) 是一种成员，用于实现可以由对象或类执行的计算或操作。
  - 静态方法 (static method) 通过类来访问；
  - 实例方法 (instance method) 通过类的实例来访问。
- 方法具有一个参数 (parameter) 列表（可以为空），表示传递给该方法的值或变量引用
- 方法还具有一个返回类型 (return type)，指定该方法计算和返回的值的类型。如果方法不返回值，则其返回类型为void。

方法（二）

- 方法的签名 (signature) 在声明该方法的类中必须唯一。方法的签名由方法的名称、类型参数的数目以及该方法的参数的数目、修饰符和类型组成。方法的签名不包含返回类型。
- 方法的几个知识点：
  - 参数；
  - 方法体和局部变量；
  - 静态方法和实例方法；
  - 虚方法、重写方法和抽象方法；
  - 方法重载。

参数（一）

- 参数用于向方法传递值或变量引用。方法的参数从调用该方法时指定的实参 (argument) 获取它们的实际值。
- 四类参数：
  - 值参数：用于传递输入参数。对值参数的修改不影响为该形参传递的实参。
  - 引用参数：用于传递输入和输出参数。引用参数与实参变量表示同一存储位置。引用参数使用 ref 修饰符声明。
  - 输出参数：用于传递输出参数。输出参数与引用参数类似。不同之处是输出参数的调用方可不提供参数初始值。输出参数是用 out 修饰符声明。
  - 参数数组：允许向方法传递可变数量的实参。参数数组使用 params 修饰符声明。只有方法的最后一个参数才可以是参数数组，并且参数数组的类型必须是一维数组类型。

参数（二）

```
class Test
{
    //值参数
    static void ValueParameter(int value)
    {
        value = 100;
    }
    //引用参数
    static void RefParameter(ref int value)
    {
        value = 100;
    }
    //输出参数
    static void OutParameter(out int value)
    {
        value = 100;
    }
    //输出参数
    static void ParameterArray(params int[] array)
    {
        for (int i = 0; i < array.Length; i++)
        {
            Console.WriteLine(array[i]);
        }
    }
}
```

```
static void Main()
    {
        int valueParameter = 0;
        int refParameter = 0;
        int outParameter;
        //调用后valueParameter还是0
        ValueParameter(valueParameter);
        //调用后refParameter[必须初始化]为100
        RefParameter(ref refParameter);
        //调用后outParameter[不用初始化]为100
        OutParameter(out outParameter);
        //可以传任意多个int类型的参数
        ParameterArray(1);//输出1
        ParameterArray(1,3,5,7);//输出1357
    }
```

方法体和局部变量（一）

- 方法体指定了在调用该方法时将执行的语句。
- 方法体可以声明仅用在该方法中使用的变量，这样的变量称为局部变量 (local variable)，局部变量声明指定了类型名称、变量名称，还可指定初始值。
- C# 要求在对局部变量明确赋值之后才能使用。
- 方法可以使用 return 语句将控制返回到它的调用方。在返回 void 的方法中，return 语句不能指定表达式。在返回非 void 的方法中，return 语句必须含有一个计算返回值的表达式。

方法体和局部变量（二）

```
using System;
class Test
{
    static void Main()
    {
        //声明并赋值局部变量
        int i = 0;
        //只声明局部变量
        int j;
        Console.WriteLine(i);//0
        //编译报错：使用了未赋值的局部变量“j”
        Console.WriteLine(j);
        //方法返回:void方法可以省略return
        return;
    }
}
```

静态方法和实例方法（一）

- 使用 static 修饰符声明的方法为静态方法 (static method)。静态方法不对特定实例进行操作，并且只能直接访问静态成员。通过类型进行调用。

- 不使用 static 修饰符声明的方法为实例方法 (instance method)。实例方法对特定实例进行操作，并且能够访问静态成员和实例成员。在调用实例方法的实例上，可以通过 this 显式地访问该实例。通过对象进行调用。

静态方法和实例方法（二）

```
using System;
class Test
{
    public static void StaticMethod()
    {
        Console.WriteLine("我是静态方法");
    }
    public void InstanceMethod()
    {
        Console.WriteLine("我是实例方法");
    }
    static void Main()
    {
        //调用静态方法
        Test.StaticMethod();
        //调用实例方法
        Test test = new Test();
        test.InstanceMethod();
    }
}
```

扩展方法

- “扩展方法”使你能向现有的类型“动态”添加方法，却不需要创建新的派生类型、重新编译或以直接修改原始类型的源代码。

- 扩展方法是一种特殊的静态方法，但可以其调用方式与调用普通的实例方法一样。

- 扩展方法中第一个参数为this，专用于指明此扩展方法所“适用”的类型，在扩展方法的实现代码中，可以用也可以不用。

扩展方法说明

- 扩展方法之所以能动态扩充现有数据类型的功能 而又无需对原有数据类型进行“伤筋动骨”的改造，其关键在于编译器在后台所完成的大量工作。

- 在编译使用了扩展方法的代码时，编译器会将这些调用代码转换为对相应静态方法的调用。因此， 在编译生成的程序集（DLL或EXE）中并不存在着一种特殊的调用扩展方法的指令。

- 与原始类型的方法具有相同名称和签名的扩展方法永远不会被调用。

扩展方法使用

- 一般情况下，多将扩展方法集中存入静态类中。

- 在需要使用扩展方法的地方，将保存扩展方法的静态类所在的命名空间引入：
  - using 扩展方法所在的命名空间;
  - 之后就可以直接使用扩展方法了。

扩展方法示例

```
public static class MyExtensions
{
    public static string UpperLower(this string str, bool upperFirst)
    {
        StringBuilder newString = new StringBuilder(str.Length);
        for (int i = 0; i < str.Length; i++)
        {
            newString.Append(upperFirst ? char.ToUpper(str[i]) :
                char.ToLower(str[i]));
            upperFirst = !upperFirst;
        }
        return newString.ToString();
    }
}
string input = Console.ReadLine();
Console.WriteLine("calling extension method for {0}: {1}", input,input.UpperLower(true));
```

虚方法，抽象方法和重写（一）

- 虚方法：若一个实例方法的声明中含有virtual 修饰符，则称该方法为虚方法 (virtual method)。

- 抽象方法：使用 abstract 修饰符进行声明，该方法没有方法体，并且只允许出现在同样被声明为abstract 的类中。抽象方法必须在每个非抽象派生类中重写。

- 重写：虚方法和抽象方法可以在派生类中重写 (override)。当某个实例方法声明包括 override 修饰符时，该方法将重写所继承的具有相同签名的虚方法。在调用一个虚方法或抽象方法是，该调用所涉及的实例的运行时类型 (runtime type) 确定了要实际调用的方法实现。而在非虚方法调用中，实例的编译时类型 (compile-time type) 负责做出此决定。

虚方法，抽象方法和重写（二）

```
abstract class BaseClass
{
    //虚方法
    public virtual void VirtualMethod()
    {
        Console.WriteLine("BaseClass.VirtualMethod");
    }
    //抽象方法
    public abstract void AbstractMethod();
}
class SubClass : BaseClass
{
    //重写虚方法
    public override void VirtualMethod()
    {
        Console.WriteLine("SubClass.VirtualMethod");
    }
    public override void AbstractMethod()
    {
        Console.WriteLine("重写抽象方法");
    }
}
```

```
class Test
{
    static void Main()
    {
        //声明类型为基类，实际类型为子类
        BaseClass baseClass =
                        new SubClass();
        //由实际类型决定调用子类还是父类方法
        //实际是SubClass类的对象。
        //SubClass.VirtualMethod
        baseClass.VirtualMethod();
        //重写抽象方法
        baseClass.AbstractMethod();
    }
}
```

方法重载（一）

- 方法重载 (overloading) 允许同一类中的多个方法具有相同名称，条件是这些方法具有唯一的签名。在编译一个重载方法的调用时，编译器使用重载决策 (overload resolution) 确定要调用的特定方法。重载决策将查找与参数最佳匹配的方法，如果没有找到任何最佳匹配的方法则报告错误。

- 也就是除了参数类型，参数数目或者参数排列顺序不同之外其他都声明信息都相同的两个方法构成重载。

方法重载（二）

```
class Test
{
    static void F()
    {
        Console.WriteLine("F()");
    }
    static void F(int x)
    {
        Console.WriteLine("F(int)");
    }
    static void F(double x)
    {
        Console.WriteLine("F(double)");
    }
    static void F(object x)
    {
        Console.WriteLine("F(object)");
    }
    static void F(double x, double y)
    {
        Console.WriteLine("F(double, double)");
    }
}
```

```
static void Main()
    {
        //F()
        F();

        //F(int)
        F(1);

        //F(double)
        F(1.0);

        //F(object)
        F("abc");

        //F(double, double)
        F(1, 1);
    }
```

实例构造器

```
public class Test
{
    /*声明一个无参的实例构造器
     *如果你写的类里面没有一个实例构造器
     *那么编译器会为你生成一个默认的无参构造器
     *实例构造器是一个特殊的方法，没有返回类型，方法名与类名相同，
     *在new一个对象的时候调用构造器*/
    public Test() { }

    //有一个参数的实例构造器
    public Test(int value) { }

    //:this(参数)可以调用已存在的构造器
    public Test():this(0) { }
}
```

析构造器

```
class A{
    //定义析构器，不能加访问修饰符
    //语法格式：~开头+类名+无参
    //析构函数是自动调用的，它不能被显式调用。
    //当任何代码都不再可能使用一个实例时，该实例就符合被销毁的条件。
    //此后，它所对应的实例析构函数随时均可能被调用
    ~A(){
        System.Console.WriteLine("析构器执行");
    }
}
class Test{
    static void Main(){
        A a = new A();
        //符合被销毁的条件
        a = null;
        //输出：析构器执行
    }
}
```

静态构造器

```
public class Test
{
    /*静态构造器
     * 用static修饰，不能加任何访问修饰符并且必须是无参数
的
     * 静态构造器的调用由CLR在合适的时间调用，
     * 不能手动调用
     */
    static Test()
    {
        //....初始化静态成员代码
    }
}
```

运算符重载（一）

```
class Point
{
    public int X;
    public int Y;
    public Point(int x, int y)
    {
        this.X = x;
        this.Y = y;
    }
    //在Point类上重载+号操作符,使Point可以出现在+号的两端
    public static Point operator +(Point p1, Point p2)
    {
        return new Point(p1.X + p2.X, p1.Y + p2.Y);
    }
    public override string ToString()
    {
        return string.Format("[{0},{1}]", this.X, this.Y);
    }
}
```

运算符重载（二）

```
class Test
{
    static void Main()
    {
        Point point1 = new Point(10,20);
        Point point2 = new Point(1,2);
        //可以直接用+号操作两个Point对象
        //point1 - point2;这样是非法的，因为并没重载-号操作符
        Point temp = point1 + point2;
        //输出: [11,22]
        System.Console.WriteLine(temp);
    }
}
```

属性（一）

- 属性 (property) 是字段的扩展。对字段的读取和存储加以控制，隐藏字段，暴露出属性供外部使用。面向对象封装特征的体现。

- 属性有访问器 (accessor)，这些访问器指定在读取或写入它们的值时需执行的语句。包括get 和 set访问器。同时具有 get 访问器和 set 访问器的属性是读写属性 (read-write property)；只有 get 访问器的属性是只读属性 (read-only property)；只有 set 访问器的属性是只写属性(write-only property)。

- get 访问器相当于一个具有属性类型返回值的无形参方法。
- set 访问器相当于具有一个名为 value（上下文关键字）的参数并且没有返回类型的方法。

属性（二）

```
class Person
{
    private String _name = String.Empty;//空
    public String Name
    {
        get
        {
            //对取_name加以判断，为空则返回"未知"
            if (String.IsNullOrEmpty(this._name))
            {
                return "未’知a";
            }
            return this._name;
        }
        set { this._name = value; }
    }
```

属性（三）

```
private Int32 _age = 0;
    public Int32 Age
    {
        get { return this._age; }
        set
        {
            //对设置年龄加限定[0-150]
            if (value < 0 || value > 150)
            {
                throw new ArgumentException("年龄必须在0-150之间");
            }
            this._age = value;
        }
    }
}
```

属性（四）

```
class Test
{
    static void Main()
    {
        Person person = new Person();

        //调用Name的get访问器；输出"未知"
        Console.WriteLine(person.Name);
        //调用Name的set访问器；存入"小明"
        person.Name = "小明";
        //调用Name的get访问器；输出"小明"
        Console.WriteLine(person.Name);

        //调用Age的set访问器；存入-2,由于存在验证，则会报错
        person.Age = -2;
        //正常设置[0-150]
        person.Age = 15;
    }
}
```

索引器

```
public class Test
{
    int[] array = new int[] { 1, 2, 3, 4, 5, 6, 7, 8, 9, 0 };
    /*索引器
     * 类似于属性的语法，格式是：索引器修饰符 返回类型 this[参数列表]*/
    public int this[int index]
    {
        get { return this.array[index]; }
        set { this.array[index] = value; }
    }
    static void Main()
    {
        Test test = new Test();
        //调用索引器的get访问器，输出：2
        System.Console.WriteLine(test[1]);
        //调用set访问器
        test[3] = 100;
    }
}
```

继承（一）

- 一个类继承 (inherit) 它的直接基类类型的成员。意味着一个类隐式地将它的直接基类类型的所有成员当作自己的成员。
- 继承有一些重要的性质如下：
  - 继承是可传递的：如果C继承B，而B继承A，那么C就会既继承在B中声明的成员，又继承在A中声明的成员。
  - 派生类扩展 它的直接基类。派生类能够在继承基类的基础上添加新的成员。但不可以移除基类成员。
  - 实例构造函数、析构函数和静态构造函数是不可继承的，但所有其他成员是可继承的。
  - 派生类可以通过声明具有相同名称或签名的新成员来隐藏那个被继承的成员。
  - 类可以声明虚的方法、属性和索引器，而派生类可以重写这些函数成员的实现

继承（二）

```
using System;
class BaseClass
{
    public String Name = String.Empty;//空
    public Int32 Age = 0;
}
//":"表示SubClass继承BaseClass
class SubClass : BaseClass
{
    public void ShowData()
    {
        //base关键字表示访问基类的成员
        //当然你可以使用this，因为基类成员已被继承到子类来了
        Console.WriteLine("姓名：{0}\n年龄：{1}",base.Name,base.Age);
    }
}
```

继承（三）

```
class Test
{
    static void Main()
    {
        SubClass subClass = new SubClass();

        subClass.Name = "张三";
        subClass.Age = 20;

        //输出
        //姓名: 张三
        //年龄: 20
        subClass.ShowData();
    }
}
```

接口（一）

- 一个接口定义一个协定。
- 实现某接口的类或结构必须遵守该接口定义的协定。一个接口可以从多个基接口继承，而一个类或结构可以实现多个接口。
- 接口可以包含方法、属性、事件和索引器。接口本身不提供它所定义的成员的实现。接口只指定实现该接口的类或结构必须提供的成员。
- 所有接口成员都隐式地具有 public 访问属性。接口成员声明中包含任何修饰符都属于编译时错误。具体来说，不能使用修饰符 abstract、public、protected、internal、private、virtual、override 或 static 来声明接口成员

接口（二）

```
using System;

public delegate void StringListEvent(IStringList sender);

public interface IStringList
{
    //方法
    void Add(String s);
    //只读属性
    int Count { get; }
    //事件
    event StringListEvent Changed;
    //索引器
    String this[int index] { get; set; }
}
```

多态（一）

- 多态是指为同名的方法提供不同的实现的能力，它使得我们不用关心方法的具体实现而仅仅依靠其名称来进行调用操作。是面向对象编程的最重要特征（其他两个为封装，继承）。

- C#提供三种多态能力：
  - 接口多态：依赖实现接口来提供多态能力
  - 继承多态：依赖继承来提供多态能力
  - 抽象多态：依赖抽象类来提供多态能力

多态（二）

```
using System;
//声明一个接口
public interface IAnimal
{
    //打印出动物的叫声
    void Call();
}
//猫类实现接口
public class Cat : IAnimal
{
    //实现Call方法
    public void Call()
    {
        Console.WriteLine("我是小猫：喵喵~~~~~");
    }
}
```

多态（三）

```
//狗类实现接口
public class Dog : IAnimal
{
    //实现Call方法
    public void Call()
    {
        Console.WriteLine("我是小狗：汪汪~~~~~");
    }
}
public class Test
{
    static void Main()
    {
        IAnimal animal = new Cat();
        //我是小猫：喵喵~~~~~
        animal.Call();

        animal = new Dog();
        //我是小狗：汪汪~~~~~
        animal.Call();
    }
}
```

高级类型

值类型&引用类型（一）

- C#数据类型分为两大类：值类型和引用类型。两者最大的区别是值类型的变量直接代表着其存储的数据，而引用类型的变量则是一个引用，此引用指向托管堆上的一块内存，这块内存中存的才是变量代表的真正的数据。

  - 由于这些区别，在很多地方两者表现出完全不同的效果。
    - 方法传参：C#默认是按值传递，所以值类型在传参时是传递的数据的副本；引用类型则是引用的副本，调用方和被调方法会共享引用类型的数据。
    - 方法内部：值类型的数据直接存储在方法执行栈上，而引用则是在托管堆上。

值类型&引用类型（二）

- 装箱和拆箱：值类型向引用类型的转型称作装箱；引用类型向值类型的转换称作拆箱。装箱和拆箱的概念是 C# 的类型系统的核心。它值类型和引用类型之间架起了一座桥梁。

  - 装箱包括以下操作：分配一个对象实例，然后将值类型的值复制到该实例中。装箱操作大都以隐式方式转换，不需要额外操作。
  - 拆箱包括下列操作：首先检查对象实例是否是给定值类型 的装箱值，然后将该值从实例中复制出来。

- 装箱和拆箱操作因为存在内存分配操作，所以会有性能损耗。能避免的就应该避免掉。

值类型&引用类型（三）

```
class Test {
    //演示装箱和拆箱操作
    static void Main() {
        int i = 10;
        //装箱操作
        Object o = i;
        //拆箱并把拆到的值赋给j
        int j = (int)o;
        Console.WriteLine(j);
        //上面的比较明显，下面这个装箱就不容易被发现了
        //+号实际是对String.Concat方法的调用，这个方法参数类型是object
        //"1="是引用类型，只需隐式转型就可以了，不用装箱，
        //i则是int类型，所以需要装箱成object方能调用
        Console.WriteLine("i=" + i);
    }
}
```

泛型（一）-简介

- 泛型是一种类型的”多态“，通过把类型参数化来实现在同一份代码上操作多种数据类型。泛型编程是一种编程范式，它利用“参数化类型”将类型抽象化，从而实现更为灵活的复用。

- 泛型通常用与集合以及作用于集合的方法一起使用。.NET2.0 版类库提供一个新的命名空间 System.Collections.Generic，其中包含几个新的基于泛型的集合类。建议面向 2.0 版的所有应用程序都使用新的泛型集合类，而不要使用旧的非泛型集合类。

泛型（二）-特点

- C#泛型赋予了代码更强的类型安全，更好的复用，更高的效率，更清晰的约束。
- 如果实例化泛型类型的参数相同，那么CLR会重复使用该类型。
- C#泛型类型携带有丰富的元数据，因此C#的泛型类型可以应用于强大的反射技术。
- C#的泛型采用“基类, 接口, 构造器, 值类型/引用类型”的约束方式来实现对类型参数的“显式约束”。

泛型（三）-泛型类型

```
//T代表类型参数
class List<T>
{
    private T[] list;
    private int size;
    public List()
    {
        list = new T[10];
        size = 0;
    }
    public void Add(T t)
    {
        list[size++] = t;
    }
    public T Find(int index)
    {
        return list[index];
    }
}
```

使用泛型

```
//除了class之外
//C#支持另外三种（struct,delegate interface）泛型
class Test{
    static void Main()
    {
        //使用时提供类型参数<int>
        List<int> ilist = new List<int>();
        //强类型方法，只需添加int型数据
        ilist.Add(100);
        //类型参数换成string
        List<string> slist = new List<string>();
        //则此处只能添加string类型的数据
        slist.Add("string");

        Console.WriteLine(ilist.Find(0));//100

Console.WriteLine(slist.Find(0));//string
    }
}
```

泛型（四）-泛型方法

```
class Test{
    //泛型方法,可以出现在非泛型的类中
    static int Find<T>(T[] items, T t)
    {
        for (int i = 0; i < items.Length; i++)
        {
            if (items[i].Equals(t))
            {
                return i;
            }
        }
        return -1;
    }
    static void Main()
    {
        int[] iarray = new int[] { 1, 2, 3, 4, 5 };
        string[] sarray = new string[] { "A", "B", "C" };
        Console.WriteLine(Find<int>(iarray, 3));//2
        Console.WriteLine(Find<string>(sarray, "A"));//0
    }
}
```

泛型（五）-泛型约束简介

- C#泛型要求对“所有泛型类型或泛型方法的类型参数”的任何假定，都要基于显式约束，以维护C#所要求的类型安全。
- 显式约束由where子句表达，可以指定：基类约束；接口约束；构造器约束；值类型/引用类型约束；共四种约束。
- 显式约束并非必须，如果没有指定显式约束，泛型类型参数将只能访问System.Object类型中的公有方法

泛型（六）-泛型约束

```
class BaseClass { }
interface IPrint { }

//基类约束[类型参数必须是BaseClass或者是其子类]
class A<T> where T : BaseClass { }
//接口约束[类型参数必须是实现了IPrint接口的类型]
class B<T> where T : IPrint { }
//构造器约束[类型参数必须是有一个无参的构造器的类型]
class C<T> where T : new() { }
//值类型/引用类型约束[类型参数必须是引用类型]
class D<T> where T : class { }
//值类型/引用类型约束[类型参数必须是值类型]
class F<T> where T : struct { }
```

可空类型（一）

- 在可空类型出先以前，值类型是不允许为null的，值类型的性质决定它默认值只能是0的表示形式。

- 可空类型的引入解决了这个问题，泛型机制为可空类型提供了很大的灵活性，C#中的可空类型也是依赖泛型的。

- 这个泛型类的完整定义如下：
- ```
  public struct Nullable<T> where T : struct
  ```

- Nullable<T>是个值类型，类型参数也约束为struct，表示只能接受值类型。此类为所有值类型提供了一个“可空（即可为null）”的能力。
  ```

可空类型（二）

```
class Test {
    static void Main() {
        //声明一个可空类型，等同于：int? nullInt=100;
        //类型?这种写法是种语法简化，实际是完全一样的
        Nullable<int> nullInt = new Nullable<int>(100);
        //HasValue用来检查null是否有值
        if (nullInt.HasValue){
            //通过Value属性来访问实际存储的值
            //Value属性的类型取决于<类型参数>，这里是int
            Console.WriteLine(nullInt.Value);//100
        }
        nullInt = null;
        //已赋值为null，所以HasValue属性会返回false的
        Console.WriteLine(nullInt.HasValue);//false
        //在HasValue属性为false情况下访问Value属性会报错的
        Console.WriteLine(nullInt.Value);
    }
}
```

匿名方法

```
//匿名方法实际所指的并不是方法，而是和委托相关的一种语法糖。
class Test {
    //声明一个委托
    delegate void ShowDelegate(int x, int y);
    static void Main() {
        //没有匿名方法时,必须事先定义一个方法作为参数传给委托
        ShowDelegate a = new ShowDelegate(Show);
        a(1, 2);
        //有了匿名方法之后，可以省掉Show方法的定义而直接书写
        ShowDelegate b = delegate(int x, int y)
        {
            Console.WriteLine(x);
            Console.WriteLine(y);
        };
    }
    static void Show(int x, int y) {
        Console.WriteLine(x);
        Console.WriteLine(y);
    }
}
```

类型转换（一）

- **C#支持三种方式的类型转换，各有千秋。如下：**
  - (Type)：使用(Type)进行强制转换。转换不成功则报异常。
  - is：用is关键字进行检测，类型兼容则返回true，否则返回false，不兼容也不会报异常。
  - as：用as关键字进行类型转换，可转换时会返回转换后的引用，不可转换时会返回null，不会报异常。只能用在引用类型上。

类型转换（二）

```
class A { }
class B : A { }
class Test {
    static void Main() {
        B b = new B();
        Object o = new Object();
        //把o强制转换为A类型的
        //由于Object并不是A或者其子类型，所以会报异常
        // A a1 = (A)o;
        //用is检测o和b是否兼容于A
        Console.WriteLine(o is A);//false
        Console.WriteLine(b is A);//true
        //用as对o做向A的转型,即使不兼容，也不会报异常会把a2置为null
        A a2 = o as A;
        Console.WriteLine(a2==null);//true
    }
}
```

部分类（一）

- 用partial修饰的类型称作部分类，只支持类，结构和接口类型。部分类允许我们将一个类型分成几个部分，分别实现在几个不同的.cs文件中。partial 是一个上下文关键字，只有和class、struct、interface放在一起时才有关键字的含义。
- 部分类适用于以下情况：类型特别大，不宜放在一个文件中实现；一个类型中一部分代码为自动化工具生成的代码，不宜与我们自己编写的代码混合在一起。
- 部分类是一个纯语言层的编译处理，不影响任何执行机制——事实上C#编译器在编译的时候仍将各个部分的局部类型合并成一个完整的类。

部分类（二）

A1.cs文件

```
partial class A {
    public void MethodA() {

Console.WriteLine("MethodA");
    }
}
```

完全等于写在一个文件的效果

A2.cs文件

```
partial class A {
    public void MethodA() {

Console.WriteLine("MethodA");
    }
}
```

A.cs文件

```
class A {
    public void MethodA() {

Console.WriteLine("MethodA");
    }
    public void MethodB() {

Console.WriteLine("MethodB");
    }
}
```

泛型委托

泛型委托的使用与普通委托类似，不同之处在于使用泛型委托时需要指定类型参数

- 声明泛型委托

```
public delegate Tresult  MyTDelegate<T, Tresult >(T obj1, T obj2);
```

- 定义泛型委托变量

```
MyTDelegate<object, bool>  myTDelegate=new MyTDelegate<object, bool>(FunctionName);
```

  - 这里 T=object, Tresult=bool
  - 返回值为bool类型，参数为两个object类型的

```
bool FunctionName(object obj1,object obj2);
```

Lambda表达式

认识Lambda表达式

- 定义一个委托：

```
public delegate string SomeDelegateType(int arguments);
```

- 使用匿名方法给委托变量赋值

```
SomeDelegateType del1 = delegate(int arguments)
 {
     return arguments.ToString();
 };
```

- 使用Lambda表达式达到同样的目的：

```
SomeDelegateType del2 = arguments => {
   return arguments.ToString();
};
```

- 简化

```
SomeDelegateType del3 = arguments => arguments.ToString();
```

Lambda表达式定义

- Lambda表达式其实就是匿名方法的进一步简化， 可以用于定义一个匿名函数，并将其传送给一个委托变量。
- 直观来看，使用Lambda表达式可以大大简化委托的编码。
- 但事实上，Lambda表达式是LINQ（ .NET 3.5 引入）的技术基础。远非“是对匿名方法的进一 步简化”这么简单。

- 两种基本格式：
  - (input parameters) =>表达式
  - (input parameters) => {语句;}

Lambda表达式示例

- 只有一个输入参数时，括号是可选的：

```
Func<int, bool> del1 = x => x > 0;
```

- 相当于

```
Func<int, bool> del1 = (x) => { return x > 0; };
```

- Lambda表达式中的return关键字可省。
- 两个或更多输入参数由括在括号中的逗号分隔：

```
Func<int, int, bool> del2 = (x, y) => x == y;
```

类型推断

- 对于Lambda表达式的参数，在声明时可以不指定数据类型，而由编译器来动态地进行类型推断
- 有时，编译器难于或无法推断输入类型。如果出现这种情况，必须显式指定类型：

```
Func<int,string,bool> del3= (int x, string s) => s.Length > x;
```

LINQ

什么是LINQ

- LINQ：Language-Integrated Query（语言集成的查询），是.NET 3.5引入的一项重要技术。它第一次尝试着 在标准的编程语言中引入了SQL编程语言的特性，大大简化了数据存取工作。是一项很有应用前景的技术创新。

- LINQ技术在.NET整个技术领域中的重要地位。

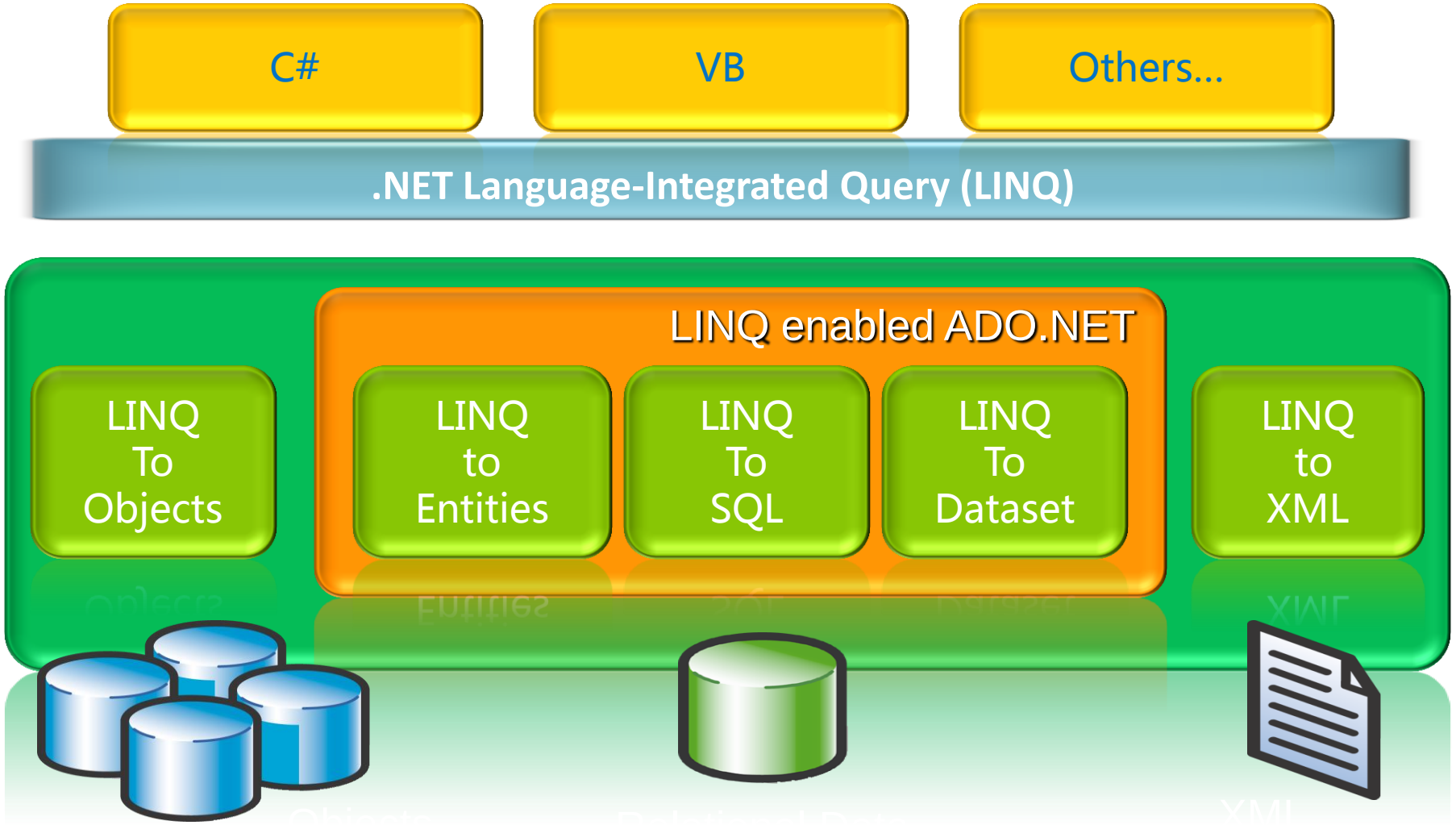
背景

- 当前，软件系统多采用基于组件的多层分布式系统架构，存在着以下问题：
  - 在软件系统的界面层与中间层，数据通常以对象和对象集合的形式表达，而数据却往往自于不同的地方，具有不同的数据格式，因此，数据的转换变得很麻烦。
  - 由于数据来源与数据格式的多样性，因此，同时存在多种数据访问技术，这些技术的集成变得很麻烦
  - 在现代软件系统中，大量在中间层组件中使用SQL命令来存取数据库，由于编译器无法检查SQL，所以，在开发中SQL命令出错的排除是非常麻烦的。
- LINQ技术就是是为了解决这些麻烦

数据源

- 存储于SQL Server数据库的关系型数据
- 存储于XML文档中的层次型数据
- ADO.NET数据集（DataSet）
- 实现IEnumerable 或泛型 IEnumerable<T> 接口的任意对象（即“可查询类型”）集合。

LINQ架构


C#
VB
Others...
.NET Language-Integrated Query (LINQ)
LINQ enabled ADO.NET
LINQ To Objects
LINQ to Entities
LINQ To SQL
LINQ To Dataset
LINQ to XML

基本语法

- 基于扩展方法的语法
- 关键字
  - where, orderby, select, group, ...
- 辅助语法功能的使用
  - 自动属性
  - 匿名类型
  - 对象的初始化

查询语法

```
var result = from p in Persons
     where p.FirstName.StartsWith("B")
     orderby p.LastName
     select new { p.FirstName, p.LastName, p.Age};
```

```
var result = Customers.Where( p =>
     p.FirstName.StartsWith("B") )
     .OrderBy( p => p.LastName )
     .Select( p => new {p.FirstName, p.LastName, p.Age} );
```

LINQ To Objects

LINQ To Object 并不是可以查询任何对象。他需要集合的支持。对象LINQ可以查询的集合必须实现了IEnumerable<T>接口，在LINQ词汇表中，实现此接口的集合，我们称之为序列。

- 与集合结合工作
- using System.Linq
- System.Core.Dll assembly

LINQ To SQL

LINQ TO SQL是LINQ技术在数据库方面的应用。
LINQ不仅仅可以对数据库进行查询，同样CUID(Create,Update, Insert,Delete)都可以实现

- DataContext 类型
- 自定义属性 (Table, Column)
- 不仅仅是 “Query”
- 可以使用存储过程(stored procedures)
- using System.Data.Linq
- System.Data.Linq.Dll Assembly

LINQ To XML

LINQ to XML 是一种启用了 LINQ 的内存 XML 编程接口，使用它，可以在 .NET Framework 编程语言中处理 XML。

- 新的对象模型
  - 没有必要创建一个文件
  - 非常直观和灵活
- using System.Xml.Linq
- System.Xml.Linq.Dll Assembly
- 容易与其他LINQ结合
  - 如:LINQ to SQL

总结

- 类：类声明，成员修饰符，创建类的实例，字段，方法，实例构造器，析构器，静态构造器，运算符重载，属性，索引器，继承，接口，多态。
- 高级类型：值类型、引用类型、泛型、可空类型、类型转换
- Lambda表达式
- LINQ、LINQ to Objects、LINQ to SQL、LINQ to XML

资源

- *Visual C#*

  http://msdn.microsoft.com/zh-cn/library/kx37x362.aspx
- *《C#高级编程》*

  http://product.china-pub.com/197224
- *《CLR 框架设计》*

  http://product.china-pub.com/28146
- *《深入理解C#》*

  http://product.china-pub.com/198866
- *Windows 8 Sample Code*
- *《Programming Windows Sixth Edition》*

  http://shop.oreilly.com/product/0790145369079.do

Q&A

Microsoft®