# 反射

## dll-IL-metadata-反射



C#->编译器编译->dll/exe->CLR/JIT->机器码（01010101）

Metadata IL(面向对象语言)

二次编译：为了一次编译，不同平台使用；加一层中间层，更灵活。

.Net的应用程序由几个部分：‘程序集(Assembly)’、‘模块(Module)’、‘类型(class)’组成

Assembly类可以获得正在运行的装配件信息，也可以动态的加载装配件，以及在装配件中查找类型信息，并创建该类型的实例。  
Type类可以获得对象的类型信息，此信息包含对象的所有要素：方法、构造器、属性等等，通过Type类可以得到这些要素的信息，并且调用之。  
MethodInfo包含方法的信息，通过这个类可以得到方法的名称、参数、返回值等，并且可以调用之。  
诸如此类，还有FieldInfo、EventInfo等等，这些类都包含在System.Reflection命名空间下。

以下是上面几个类的使用方法：  
（1）使用Assembly定义和加载程序集，加载在程序集清单中列出模块，以及从此程序集中查找类型并创建该类型的实例。   
（2）使用Module了解包含模块的程序集以及模块中的类等，还可以获取在模块上定义的所有全局方法或其他特定的非全局方法。   
（3）使用ConstructorInfo了解构造函数的名称、参数、访问修饰符（如pulic 或private）和实现详细信息（如abstract或virtual）等。使用Type的GetConstructors或 GetConstructor方法来调用特定的构造函数。   
（4）使用MethodInfo了解方法的名称、返回类型、参数、访问修饰符（如pulic 或private）和实现详细信息（如abstract或virtual）等。使用Type的GetMethods或GetMethod方法来调用特定的方法。   
（5）使用FiedInfo了解字段的名称、访问修饰符（如public或private）和实现详细信息（如static）等，并获取或设置字段值。   
（6）使用EventInfo了解事件的名称、事件处理程序数据类型、自定义属性、声明类型和反射类型等，添加或移除事件处理程序。   
（7）使用PropertyInfo了解属性的名称、数据类型、声明类型、反射类型和只读或可写状态等，获取或设置属性值。   
（8）使用ParameterInfo了解参数的名称、数据类型、是输入参数还是输出参数，以及参数在方法签名中的位置等。

**什么是反射？**

System.Reflection .net框架提供帮助类库。可以读取并使用metadata

反射指程序可以访问、检测和修改它本身状态或行为的一种能力。

您可以使用反射动态地创建类型的实例，将类型绑定到现有对象，或从现有对象中获取类型。然后，可以调用类型的方法或访问其字段和属性。

反射的作用：  
它允许在运行时查看特性（attribute）信息。

* 它允许审查集合中的各种类型，以及实例化这些类型。
* 它允许延迟绑定的方法和属性（property）。
* 它允许在运行时创建新类型，然后使用这些类型执行一些任务。

|  |
| --- |
| Assembly程序集 的几个方法：使用Assembly定义和加载程序集  Assembly assembly = Assembly.**Load**("07反射");//dll名称 当前目录加载  Assembly assembly1 = Assembly**.LoadFile**("C:\\Users\\j\\source\\repos\\基础练习\\07反射\\bin\\Debug\\netcoreapp2.1\\07反射.dll");//文件完整路劲的加载  Assembly assembly2 = Assembly**.LoadFrom**("07反射.dll");//带后缀或者完整路劲  若要获取[Assembly](https://docs.microsoft.com/zh-cn/dotnet/api/system.reflection.assembly?view=netframework-4.8)当前正在执行的程序集的对象， **[GetExecutingAssembly](https://docs.microsoft.com/zh-cn/dotnet/api/system.reflection.assembly.getexecutingassembly?view=netframework-4.8)**请使用方法。  string assemblyName = Assembly.GetExecutingAssembly().GetName().Name; |

## 反射加载dll,读取module、类、方法、特性

### 2.1创建类

|  |
| --- |
| /// <summary>  /// 单例模式  /// </summary>  namespace a.b.f  {  public class Singleton  {  private static Singleton \_Singleton = null;  private Singleton()  {  Console.WriteLine("Singleton被构造");  }  static Singleton()  {  \_Singleton = new Singleton();  }  public static Singleton GetInstance()  {  return \_Singleton;  }  }  } |
| static void Main(string[] args)  {  // Singleton s = new Singleton(); //因为是私有的构造函数 不能被调用  Singleton s1 = Singleton.GetInstance(); //静态类只能被初始化一次  Singleton s3 = Singleton.GetInstance();  Singleton s4 = Singleton.GetInstance();  //反射  Assembly a = Assembly.Load("a.b.f");  Type type = a.GetType("a.b.f.Singleton");  Singleton s2 = (Singleton)Activator.CreateInstance(type, true); //相当于Singleton s2 = new Singleton();  Singleton s5 = (Singleton)Activator.CreateInstance(type, true); //相当于Singleton s5 = new Singleton();  Singleton s6 = (Singleton)Activator.CreateInstance(type, true); //相当于Singleton s6 = new Singleton();  Console.ReadKey();  } |
| 如果是 泛型类  namespace a.b.f  {  public class GenericClass<T,W,X>  {  public void show(T t,W w,X x)  {  Console.WriteLine("t.type={0},w.type={1},x.type={2}",t.GetType().Name, w.GetType().Name, x.GetType().Name);  }  }  public class GenericMethod  {  public void Show<T,W,X>(T t,W w,X x)  {  Console.WriteLine("t.type={0},w.type={1},x.type={2}", t.GetType().Name, w.GetType().Name, x.GetType().Name);  }  }  } |
| {  //反射  Assembly a = Assembly.Load("a.b.f");  Type type = a.GetType("a.b.f.GenericClass`3"); //class GenericClass<T,W,X>  //object o1 = Activator.CreateInstance(type);  //MakeGenericType() 替代由当前泛型类型定义的类型参数组成的类型数组的元素，并返回表示结果构造类型的Type对象  Type newType = type.MakeGenericType(new Type[] { typeof(int), typeof(string), typeof(DateTime) });  object o1 = Activator.CreateInstance(newType);  } |

## 3.反射创建对象。反射+简单工厂+配置文件 选修：破坏单例 创建泛型

第一步：Assembly assembly = Assembly.Load("07反射");//dll名称 当前目录加载 加载dll

第二部：Type type=assembly .GetType();//获取类型信息

第三步：object obj=Activator.CreateInstance(type)；//创建对象

第四步：类型转换

第五步：方法调用

|  |
| --- |
|  |

配置文件编辑

App.config

<appSettings>

<add key="HelperAttributeConfig" value="a.b.f.HelperAttribute,a.b.f"/>

</appSettings>

读取配置文件

引用：System.Configuration

Private static string HelperAttributeConfig = ConfigurationManager.AppSettings["HelperAttributeConfig"];

## 反射调用实例方法、静态方法、重载方法 选修：调用私有方法、调用泛型方法

|  |
| --- |
| 类  using System;  using System.Collections.Generic;  using System.Linq;  using System.Text;  using System.Threading.Tasks;  namespace a.b.f  {  public class ReflectionTest  {  public ReflectionTest()  {  Console.WriteLine("这里是{0}无参构造函数",this.GetType());  }  public ReflectionTest(string name)  {  Console.WriteLine("这里是{0}有参构造函数", this.GetType());  }  public ReflectionTest(int d)  {  Console.WriteLine("这里是{0}有参构造函数", this.GetType());  }  #region Method  public void Show1()  {  Console.WriteLine("这里是{0}的show1", this.GetType());  }  public void Show2(int d)  {  Console.WriteLine("这里是{0}的show2", this.GetType());  }  public void Show3(string name)  {  Console.WriteLine("这里是{0}的show3", this.GetType());  }  private void Show4(string name)  {  Console.WriteLine("这里是{0}的show4", this.GetType());  }  public static void Shhow5(string namme)  {  Console.WriteLine("这里是{0}的show5", typeof(ReflectionTest));  }  public void Show3()  {  Console.WriteLine("这里是{0}的show3\_1", this.GetType());  }  public void Show3(string name,int id)  {  Console.WriteLine("这里是{0}的show3\_2", this.GetType());  }  public void Show3(int id)  {  Console.WriteLine("这里是{0}的show3\_3", typeof(ReflectionTest));  }  #endregion  }  } |
| Console.WriteLine("================method======================");  {  //反射  Assembly a = Assembly.Load("a.b.f");  Type type = a.GetType("a.b.f.ReflectionTest");  ReflectionTest r1 = (ReflectionTest)Activator.CreateInstance(type);  foreach(var item in type.GetMethods())  {  Console.WriteLine(item.Name);  }  {  MethodInfo method=type.GetMethod("Show1");  method.Invoke(r1, null);  }  {  MethodInfo method = type.GetMethod("Show2");  method.Invoke(r1, new object[] {123 });  }  {  //重载  MethodInfo method = type.GetMethod("Show3",new Type[] { typeof(int)});  method.Invoke(r1, new object[] { 123 });  }  {  //重载  MethodInfo method = type.GetMethod("Show3", new Type[] { typeof(string) });  method.Invoke(r1, new object[] { "果然" });  }  {  //重载  MethodInfo method = type.GetMethod("Show3", new Type[] { typeof(string), typeof(int) });  method.Invoke(r1, new object[] { "果然",123 });  }  调用私有方法  {  MethodInfo method = type.GetMethod("Show4", BindingFlags.Instance|BindingFlags.NonPublic);  method.Invoke(r1, new object[] { "果然"});  } |
| 结果 |

|  |
| --- |
| 泛型方法 泛型类  #region 泛型类 泛型方法  public class GenericDouble<T>  {  public void Show<W, X>(T t, W w, X x)  {  Console.WriteLine("t.type={0},w.type={1},x.type={2}", t.GetType().Name, w.GetType().Name, x.GetType().Name);  }  }  #endregion |
| Console.WriteLine("================泛型方法method======================");  {  //反射  Type typeGenericDouble = a.GetType("a.b.f.GenericDouble`1"); //GenericDouble<T>  Type newType = typeGenericDouble.MakeGenericType(new Type[] { typeof(int) });  object genericDouble = Activator.CreateInstance(newType);  MethodInfo method = newType.GetMethod("Show");  MethodInfo newMethod = method.MakeGenericMethod(new Type[] {typeof(string), typeof(DateTime) });  newMethod.Invoke(genericDouble, new object[] { 123, "诗人", DateTime.Now });  } |

## 反射字段和属性，分别获取值和设置值

|  |
| --- |
| namespace a.b.f  {  public class People  {  public People()  {  Console.WriteLine("{0}被创建", this.GetType().FullName);  }  public int Id { get; set; }  public string Name { get; set; }  public string Description { get; set; }  }  } |
| Console.WriteLine("================类及属性=====================");  {  People p = new People  {  Id=1,  Name="花花",  Description="三好学生"  };  Console.WriteLine($"people.Id={p.Id}");  Console.WriteLine($"people.Name={p.Name}");  Console.WriteLine($"people.Description={p.Description}");  Type tPeople = typeof(People);  object oPeople = Activator.CreateInstance(tPeople);  Console.WriteLine("++++++++++GetProperties++++++++++++");  foreach (var prop in tPeople.GetProperties())  {  Console.WriteLine(tPeople.Name);  Console.WriteLine(prop.Name);  Console.WriteLine(prop.GetValue(oPeople));  if (prop.Name.Equals("Id"))  {  prop.SetValue(oPeople, 2);  }else if(prop.Name.Equals("Name"))  {  prop.SetValue(oPeople, "xiaoxiao");  }  Console.WriteLine($"{tPeople.Name}.{prop.Name}={prop.GetValue(oPeople)}");  }  Console.WriteLine("++++++++++GetFields++++++++++++");  foreach (var field in tPeople.GetFields())  {  Console.WriteLine(tPeople.Name);  Console.WriteLine(field.Name);  Console.WriteLine(field.GetValue(oPeople));  if (field.Name.Equals("Id"))  {  field.SetValue(oPeople, 2);  }  else if (field.Name.Equals("Name"))  {  field.SetValue(oPeople, "xiaoxiao");  }  Console.WriteLine($"{tPeople.Name}.{field.Name}={field.GetValue(oPeople)}");  }  } |
| 结果  ================类及属性=====================  a.b.f.People被创建  people.Id=1  people.Name=花花  people.Description=三好学生  a.b.f.People被创建  ++++++++++GetProperties++++++++++++  People  Id  0  People.Id=2  People  Name  People.Name=xiaoxiao  People  Description  People.Description=  ++++++++++GetFields++++++++++++ |

应用实例

|  |
| --- |
| using System;  using System.Collections.Generic;  using System.Linq;  using System.Text;  using System.Threading.Tasks;  namespace a.b.f  {  public class People  {  public People()  {  Console.WriteLine("{0}被创建", this.GetType().FullName);  }  public int Id { get; set; }  public string Name { get; set; }  public string Description { get; set; }  }  public class PeopleDTO  {  public PeopleDTO()  {  Console.WriteLine("{0}被创建", this.GetType().FullName);  }  public int Id { get; set; }  public string Name { get; set; }  public string Description { get; set; }  }  } |
| Console.WriteLine("================应用实例====================");  {  People p = new People()  {  Id = 1,  Name = "花花",  Description = "三好学生"  };  {  PeopleDTO peopleDTO = new PeopleDTO()  {  Id = p.Id,  Name = p.Name,  Description = p.Description  };//硬编码  }  //利用反射赋值 将对象People的属性值赋值给PeopleDTO  {  Type typePeople = typeof(People);  Type typePeopleDTO = typeof(PeopleDTO);  object peopleDTO = Activator.CreateInstance(typePeopleDTO);  foreach (var prop in typePeopleDTO.GetProperties())  {  //if (prop.Name.Equals("Id"))  //{  // //object value=typePeople.GetProperty("Id").GetValue(p);  // object value = typePeople.GetProperty(prop.Name).GetValue(p);  // prop.SetValue(peopleDTO, value);  //}  //else if (prop.Name.Equals("Name"))  //{  // object value = typePeople.GetProperty(prop.Name).GetValue(p);  // prop.SetValue(peopleDTO, value);  //}  object value = typePeople.GetProperty(prop.Name).GetValue(p);  prop.SetValue(peopleDTO, value);  }  }    } |

## 反射的好处和局限优点

优点：动态

缺点：写起来复杂、编译器检查不到、性能问题