## .Net动态代码生成技术—Emit

.Net的母语是什么？C#，VB，F#？非也，.Net的母语是MSIL，就是常常被我们挂在嘴边说的中间语言。所有基于.Net平台的语言都要编译成中间语言交给运行时去执行，所以说运行时在运行你的程序的时候并不知道你的程序是用什么语言写的。那这里要说到的Emit到底是个什么东西？它是framework提供给我们的一套api，能够在运行时动态地生成IL。

听起来似乎是个很高级的东西，但是这部分内容并不是什么被微软藏起来不让你看到的东西，作为.Net技术的一部分，在很多技术书籍里都有专门的章节来讲，你甚至可以直接编写IL代码，在.net开发包里提供了直接将IL代码编译成程序集的工具：ilasam，不熟悉它只是因为实际中用到的机会很少。但是用不到并不等于这东西不重要，动态代码生成可以用于动态代理技术，可以用于优化关键代码的执行效率；.net中的表达式树编译成动态方法的时候底层用的就是emit；网上流行的一个轻型开源ORM框架Dapper大量地使用了Emit，据说性能非常好。

用Emit生成IL代码的过程就是依次向一个执行流里注入操作码，所以Emit的意思“发射”，“发出”，感觉还是挺形象的。要使用这个工具，需要对常用的IL操作码以及IL基于栈的执行模型有个基本的了解，下面要说的并不是关于具体怎么使用Emit，我想用一个实际的例子来体现一下Emit的效率，以及我觉得比较好的使用它的方式。这个例子就是在很多ORM框架里经常要把数据行映射到一个实体对象，很多做法是用反射，但是反射效率非常低下，一种替代的方式就是进行手动的转换，但是这又会降低开发效率，使用Emit将会是个两全其美的方法。

比如有下面的代码：

private static Entity DataRowToEntity(DataRow row)

{

Entity p = new Entity();

var columns = row.Table.Columns;

if (columns.Contains(""))

{

p.String = row.Field<string>("Name");

}

if (columns.Contains("Age"))

{

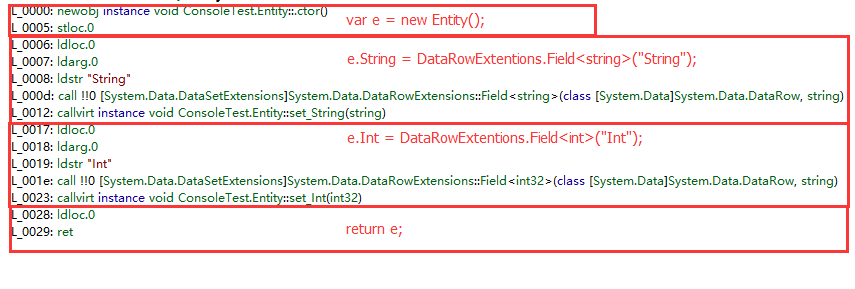
p.Byte = row.Field<byte>("Age");

}

return p;

}

这段代码是一个静态的手动转换实体的方式，针对每个类型的实体，都需要去编写大体相同的代码，这是执行效率最高的，却是开发效率最低的。现在的目的是：在运行时针对每种类型动态地生成对应的转换代码，且不损失效率。Emit可以用于动态生成代码，不损失效率换句话说就是生成的IL代码和编译上面产生的IL代码一致。所以要玩emit我认为比较方便的是先针对某个具体类型编写静态代码，然后查看编译后的IL代码，最后就是照着反编译的IL代码模板，调用emit相关的api编写动态生成代码。注意：记得在vs项目属性的生成选项卡里勾选“优化代码”，否则生成的IL代码有许多冗余代码，干扰视线。上面的代码生成的IL如下图：



有关IL比较重要的一个点是IL代码的执行模型是基于栈的，所有数据必须先入栈，然后再出栈进行具体操作。比如我要给一个本地的int变量赋值，那么栈顶必须事先有一个能够赋值给int的数据，这个数据可以是先前通过入栈对应的操作码ldXXX进行显式入栈，或者有可能是之前调用了有返回值的方法后的返回值隐式入栈。入栈和出栈操作都有显式和隐式的，大多数出栈操作都是隐式的，比如调用一个具有3个参数的方法，栈顶的3个参数会自动出栈传给方法。显式的入栈操作很多都是由一个带有ld（load）前缀的ldXXX操作码给出，隐式的入栈操作大多数是表达式的返回值自动入栈，比如算数运算，有返回值的方法调用结果。

对于常用的操作码的作用，这些都可以查阅。

接下来的任务就是根据生成的IL代码进行emit，这个过程就是用代码依次生成上图中的那些IL指令。我们的代码生成逻辑放在一个DataRowToEntityHelper<T>的静态类里边，类型参数代表实体类型，代码生成过程如下：

1. 创建IL生成器：

DynamicMethod dm = new DynamicMethod("DataRowToEntity", typeof(T), new Type[] { typeof(DataRow) });

ILGenerator il = dm.GetILGenerator();

第一行代码是创建方法签名，第二行代码是获得方法对应的IL生成器，之后的代码注入都是在调用ILGenerator这个类的相关方法进行的。

1. 创建对象（对应上图中的第一部分）：

var entity = il.DeclareLocal(typeof(T));

il.Emit(OpCodes.Newobj, typeof(T).GetConstructor(new Type[0]));

il.Emit(OpCodes.Stloc, entity);

1. 对属性赋值，每个属性的赋值代码基本类似，可以在一个循环里进行：

var props = from p in typeof(T).GetProperties(BindingFlags.Instance |

BindingFlags.Public)

where p.CanWrite && p.GetSetMethod().IsPublic

select Tuple.Create(p.GetSetMethod(), p.Name, p.PropertyType);

var rowFieldMethod = typeof(DataRowExtensions).GetMethod("Field",

new Type[] { typeof(DataRow), typeof(string) });

foreach (var p in props)

{

il.Emit(OpCodes.Ldloc, entity);

il.Emit(OpCodes.Ldarg\_0);

il.Emit(OpCodes.Ldstr, p.Item2);

il.Emit(OpCodes.Call, rowFieldMethod.MakeGenericMethod(p.Item3));

il.Emit(OpCodes.Call, p.Item1);

}

1. 返回值：

il.Emit(OpCodes.Ldloc, entity);

il.Emit(OpCodes.Ret);

1. 返回最终生成的方法指针（委托）：

return (Func<DataRow, T>)dm.CreateDelegate(typeof(Func<DataRow, T>));

比较IL代码和我们写的代码基本上区别不大，按照这种方式，玩emit还是很方便的。

对上面的代码进行几处改进

1. 属性赋值时判断列是否存在：

同样对应的静态代码为：

Entity e = new Entity();

var columns = row.Table.Columns;

if (columns.Contains("String"))

{

e.String = row.Field<string>("String");

}

if (columns.Contains("Int"))

{

e.Int = row.Field<int>("Int");

}

return e;

对应的IL代码为：



这里多出来的IL代码包括声明columns变量，属性赋值的判断逻辑代码。上面用箭头标出来的两个地方分别对应两处分支判断逻辑，对于高级语言，分支控制逻辑有if，while，for，switch，等，但是对IL代码来说，用于分支控制的指令只有跳转，所有高级语言层面的分支控制最终都转化成一系列IL跳转指令。跳转指令需要指定跳转目标，这个目标用一个lable（标签，图中用小框标出的部分）标识。上面的属性赋值的emit代码对应如下：

Label lbNext;

foreach (var p in props)

{

lbNext = il.DefineLabel();

l.Emit(OpCodes.Ldloc, locColums);

il.Emit(OpCodes.Ldstr, p.Item2);

il.Emit(OpCodes.Call, typeof(DataColumnCollection).GetMethod("Contains"));

il.Emit(OpCodes.Brfalse\_S, lbNext);

il.Emit(OpCodes.Ldloc, locEntity);

il.Emit(OpCodes.Ldarg\_0);

il.Emit(OpCodes.Ldstr, p.Item2);

il.Emit(OpCodes.Call, rowFieldMethod.MakeGenericMethod(p.Item3));

il.Emit(OpCodes.Call, p.Item1);

il.MarkLabel(lbNext);

}

可以看到基本上和实际的IL代码没有太大的区别，分支控制通过调用ILGenerator的两个方法：DefineLabel和MarkLabel，前者用于定义一个标签，后者将标签应用于指令流的当前位置。

1. 配置属性对应的列名：

到现在为止，每个属性名和列名是相同的，很多时候我们需要自定义列名，比如根据属性名上的某个特性类确定列名。对于这个需求只需要稍微改动代码就行：方法传入一个Func<PropertyInfo，string> 类型的委托columnNameForProp，在查找类型的属性信息时应用这个委托，代码如下：

var props = from p in typeof(T).GetProperties(BindingFlags.Instance | BindingFlags.Public)

where p.CanWrite && p.GetSetMethod().IsPublic

select Tuple.Create(p.GetSetMethod(), columnNameForProp(p), p.PropertyType);

1. 配置值转换方法：

到目前为止给属性赋值的代码调用的是DataRowExtentions的Field方法，这个方法要求值的实际类型要和属性类型兼容，遇到比如decimal转int的情况就会失败。为了方便扩展，需要将对应的转换方法由外界传入。只要将原先写死的：

var rowFieldMethod = typeof(DataRowExtensions).GetMethod("Field",

new Type[] { typeof(DataRow), typeof(string) });

赋值成由外界传入的方法信息。注意传入的方法签名需要和先前的一致：一个接收DataRow和string参数，返回类型T的泛型方法；

Emit的效率

在我的机器上测试最终Emit版本的方法，手写的方法，以及由反射实现的方法，测试的实体类型属性包含各种基本类型，执行1000000次的结果如下：

Emit：1344ms

Default：1177ms

Reflection：6093ms

可见Emit和正常写的代码的效率相当，而反射会有5-6倍的差距；

Emit执行效率之所以高，是因为它的代码和静态编译出来的代码基本没有什么差别；

除效率外，反射的代码在运行时基本不会导致进程工作集增加，而emit会，因为它需要针对不同类型生成不同的代码，不过这个在大多数情况下并不成问题。