А.В. Макаров, С.Ю. Скоробогатов, А.М. Чеповский

Учебный курс “CIL и системное программирование в Microsoft .NET”

Лекция 16.

Введение в динамическую генерацию кода

Введение

- Динамическая генерация кода – это прием программирования, заключающийся в том, что фрагменты кода порождаются и запускаются непосредственно во время выполнения программы
- Этот прием был известен достаточно давно, но усложнение архитектуры компьютеров, и, что особенно важно, усложнение наборов команд процессоров привело к тому, что в последние 10-15 лет динамическая генерация кода в некоторой степени потеряла популярность

Динамическая генерация => специализация

- Целью динамической генерации кода является использование информации, доступной только во время выполнения программы, для повышения качества исполняемого кода
- В терминах метавычислений можно сказать, что динамическая генерация кода позволяет специализировать фрагменты программы по данным, известным во время выполнения
- В некотором смысле, любой JIT-компилятор как раз использует динамическую генерацию кода. При этом можно считать, что тип процессора – эта как раз та часть информации, которая становится известной только во время выполнения программы

Область применения динамической генерации кода

- Применение динамической генерации оправдано, если:
  1. процесс вычислений в некотором фрагменте программы преимущественно определяется информацией, известной только во время выполнения
  2. запуск этого фрагмента осуществляется многократно
  3. выполнение фрагмента связано с существенными затратами времени процессора

Динамическая генерация кода в .NET

- В .NET доступно два способа организации динамической генерации кода:
  1. порождение программы на языке C# и вызов компилятора C#
  2. непосредственное порождение метаданных и CIL-кода
- Порождение C#-программы несколько проще, нежели генерация CIL-кода
- Генерация CIL-кода выполняется на порядок быстрее и дает большую гибкость

Пример: интегрирование функции

- В этой лекции мы рассмотрим простой пример программы на языке C#, выполняющей численное интегрирование функции:
  - Функция задается в виде строки (считаем, что она становится известна только на момент выполнения программы)
  - Характерной особенностью задачи численного интегрирования является необходимость многократного вычисления значения функции в разных точках
  - Вычисление значения функции связано со значительными затратами времени процессора
- Таким образом, данная задача по всем признакам подходит для использования динамической генерации кода

Три способа вычисления значения функции

- Мы будет выполнять вычисление значения функции тремя способами:
  - без динамической генерации кода (путем непосредственной интерпретации выражения)
  - путем динамической генерации программы на языке C#
  - путем динамической генерации метаданных и CIL-кода
- Затем мы сравним эффективность каждого способа

16.1. Обобщенный алгоритм интегрирования

- Для интегрирования функций нам потребуется некое представление функции, которое бы не зависело от конкретного способа вычисления значения функции. Идеальным вариантом такого представления является абстрактный класс Function :

```
public abstract class Function
{
        public abstract double Eval(double x);
}
```

Алгоритм интегрирования

- Реализация алгоритма интегрирования:

```
static double Integrate
    (Function f, double a, double b, int n)
{
    double h = (b-a)/n, sum = 0.0;

    for (int i = 0; i < n; i++)
        sum += h*f.Eval((i+0.5)*h);

    return sum;
}
```

Тестовый класс для проверки работоспособности алгоритма интегрирования

- Класс TestFunction реализует вычисление функции

  f(x) = x * sin(x)

```
public class TestFunction: Function
{
        public override double Eval(double x)
        {
                return x * Math.Sin(x);
        }
}
```

16.2. Представление выражений

- Парсер выражений переводит их из строковой формы в дерево синтаксического разбора
- Каждый узел дерева является объектом класса, наследующего от абстрактного класса Expression:

```
public abstract class Expression
{
        public abstract string GenerateCS();
        public abstract void GenerateCIL(ILGenerator il);
        public abstract double Evaluate(double x);
}
```

- Мы не будем рассматривать детали синтаксического разбора и организации дерева

Методы GenerateCS() и GenerateCIL(): пример работы

- Пусть дано выражение "2*x*x*x+3*x*x+4*x+5"
- Тогда GenerateCS() возвратит:
  ((((((2)*(x))*(x))*(x))+(((3)*(x))*(x)))+((4)*(x)))+(5)
- А GenerateIL() сгенерирует код:

```
ldc.r8 2.0
ldarg.1; mul; ldarg.1; mul; ldarg.1; mul
ldc.r8 3.0
ldarg.1; mul; ldarg.1; mul
add
ldc.r8 4.0
ldarg.1; mul
add
ldc.r8 5.0
add
```

16.3. Трансляция выражений в C# (слайд 1)

- Метод, транслирующий дерево выражения в C#-код, а затем вызывающий компилятор C#:

```
static Function CompileToCS(Expression expr);
```

- Инициализация компилятора C#:

```
ICodeCompiler compiler =
    new CSharpCodeProvider().CreateCompiler();
CompilerParameters parameters =
    new CompilerParameters();

parameters.ReferencedAssemblies.Add("System.dll");
parameters.ReferencedAssemblies.Add("Integral.exe");
parameters.GenerateInMemory = true;
```

Трансляция выражений в C# (слайд 2)

- Генерация строки с текстом программы на C#:

```
string e = expr.GenerateCS();

string code =
    "public class FunctionCS: Function\n"+
    "{\n"+
    "    public override double Eval(double x)\n"+
    "    {\n"+
    "        return "+e+";\n"+
    "    }\n"+
    "}\n";
```

Трансляция выражений в C# (слайд 3)

- Вызов компилятора C#, получение динамической сборки, создание объекта-функции:

```
CompilerResults compilerResults =
    compiler.CompileAssemblyFromSource(parameters,code);

Assembly assembly = compilerResults.CompiledAssembly;

return assembly.CreateInstance("FunctionCS") as Function;
```

16.4. Трансляция выражений в CIL (слайд 1)

- Метод, транслирующий дерево выражения в CIL-код через библиотеку рефлексии:

```
static Function CompileToCIL(Expression expr);
```

- Создание динамической сборки:

```
AppDomain appDomain = Thread.GetDomain();

AssemblyName assemblyName = new AssemblyName();
assemblyName.Name = "f";
AssemblyBuilder assembly =
    appDomain.DefineDynamicAssembly(
        assemblyName,
        AssemblyBuilderAccess.RunAndSave
    );
```

Трансляция выражений в CIL (слайд 2)

- Создание модуля в сборке и типа FunctionCIL:

```
ModuleBuilder module =
    assembly.DefineDynamicModule("f.dll", "f.dll");

TypeBuilder typeBuilder =
    module.DefineType(
        "FunctionCIL",
        TypeAttributes.Public | TypeAttributes.Class,
        typeof(Function)
    );
```

Трансляция выражений в CIL (слайд 3)

- Создание конструктора класса:

```
ConstructorBuilder cons =
    typeBuilder.DefineConstructor(
        MethodAttributes.Public,
        CallingConventions.Standard,
        new Type[] { }
    );

ILGenerator consIl = cons.GetILGenerator();
consIl.Emit(OpCodes.Ldarg_0);
consIl.Emit(OpCodes.Call,
    typeof(object).GetConstructor(new Type[0]));
consIl.Emit(OpCodes.Ret);
```

Трансляция выражений в CIL (слайд 4)

- Создание метода eval():

```
MethodBuilder evalMethod =
    typeBuilder.DefineMethod(
        "Eval",
        MethodAttributes.Public |
  MethodAttributes.Virtual
            | MethodAttributes.HideBySig,
        typeof(double),
        new Type[] { typeof(double) }
    );

ILGenerator il = evalMethod.GetILGenerator();
expr.GenerateCIL(il);
il.Emit(OpCodes.Ret);
```

Трансляция выражений в CIL (слайд 5)

- Завершение генерации сборки, создание объекта функции:

```
Type type = typeBuilder.CreateType();

ConstructorInfo ctor =
    type.GetConstructor(new Type[0]);

return ctor.Invoke(null) as Function;
```

16.5. Сравнение эффективности трех способов вычисления выражений

Способ вычисления значения функции	Время на создание динамической сборки, мс	Время вычисления интеграла функции, мс
Интерпретация дерева выражения	–	29422
Предварительная компиляция в C#	547	172
Предварительная компиляция в CIL	63	172

- Для измерений использовался компьютер с процессором Intel Pentium 4 3000 МГц и 1 Гб оперативной памяти