Code Generation

Code generation is often used by serialization frameworks, Object/Relational Mappers (ORMs), dynamic proxies, and other performance-sensitive code that needs to work with dynamic, unknown types. There are several ways to dynamically generate code in the .NET Framework, and there are many third-party code generation frameworks, such as LLBLGen and T4 templates.

- Lightweight Code Generation (LCG), a.k.a. `DynamicMethod`. This API can be used to generate a method without creating a type and assembly to contain it. For small chunks of code, this is the most efficient code generation mechanism. Emitting code in an LCG method requires the `ILGenerator` class, which works directly with IL instructions.
- The `System.Reflection.Emit` namespace contains APIs that can be used to generate assemblies, types, and methods—on the IL level.
- Expression trees (in the `System.Linq.Expression` namespace) can be used to create lightweight expressions from a serialized representation.
- The `CSharpCodeProvider` class can be used to directly compile C# source code (provided as a string or read from a file) to an assembly.

Generating Code from Source

Suppose that you are implementing a serialization framework that writes out an XML representation of arbitrary objects. Using Reflection to obtain non-null public field values and write them out recursively is fairly expensive, but makes for a simple implementation:

```
//Rudimentary XML serializer - does not support collections, cyclic references, etc.
public static string XmlSerialize(object obj) {
  StringBuilder builder = new StringBuilder();
  Type type = obj.GetType();
  builder.AppendFormat(" < {0} Type = '{1}'" > ", type.Name, type.AssemblyQualifiedName);
  if (type.IsPrimitive || type == typeof(string)) {
    builder.Append(obj.ToString());
  } else {
    foreach (FieldInfo field in type.GetFields()) {
    object value = fi eld.GetValue(obj);
    if (value ! = null) {
    builder.AppendFormat(" < {0} > {1}</{0} > ", field.Name, XmlSerialize(value));
    }
    }
  }
  builder.AppendFormat("</{0} > ", type.Name);
  return builder.ToString();
}
```

Instead, we could generate strongly-typed code to serialize a specific type and invoke that code. Using `CSharpCodeProvider`, the gist of the implementation looks as follows:

```
public static string XmlSerialize<T>(T obj){
  Func<T,string> serializer = XmlSerializationCache<T>.Serializer;
  if (serializer == null){
    serializer = XmlSerializationCache<T>.GenerateSerializer();
  }
  return serializer(obj);
}

private static class XmlSerializationCache < T > {
  public static Func < T,string > Serializer;
  public static Func < T,string > GenerateSerializer() {
    StringBuilder code = new StringBuilder();
    code.AppendLine("using System;");
    code.AppendLine("using System.Text;");
    code.AppendLine("public static class SerializationHelper {");
    code.AppendFormat("public static string XmlSerialize({0} obj) {{", typeof(T).FullName);
    code.AppendLine("StringBuilder result = new StringBuilder();");
    code.AppendFormat("result.Append(\" < {0} Type = '{1}'" > \");", typeof(T).Name, typeof(T).AssemblyQualifiedName);
    if (typeof(T).IsPrimitive || typeof(T) == typeof(string)) {
    code.AppendLine("result.AppendLine(obj.ToString());");
    } else {
    foreach (FieldInfo field in typeof(T).GetFields()) {
       code.AppendFormat("result.Append(\" < {0} > \");", field.Name);
       code.AppendFormat("result.Append(XmlSerialize(obj.{0}));", field.Name);
       code.AppendFormat("result.Append(\"</{0} > \");", field.Name);
       }
    }
    code.AppendFormat("result.Append(\"</{0} > \");", typeof(T).Name);
    code.AppendLine("return result.ToString();");
    code.AppendLine("}");
    code.AppendLine("}");

    CSharpCodeProvider compiler = new CSharpCodeProvider();
    CompilerParameters parameters = new CompilerParameters();
    parameters.ReferencedAssemblies.Add(typeof(T).Assembly.Location);
    parameters.CompilerOptions = "/optimize + ";
    CompilerResults results = compiler.CompileAssemblyFromSource(parameters, code.ToString());
    Type serializationHelper = results.CompiledAssembly.GetType("SerializationHelper");
    MethodInfo method = serializationHelper.GetMethod("XmlSerialize");
    Serializer = (Func <T,string>)Delegate.CreateDelegate(typeof(Func <T,string>), method);
    return Serializer;
  }
}
```

The Reflection-based part has moved such that it is only used once to generate strongly-typed code—the result is cached in a static field and reused every time a certain type must be serialized. Note that the serializer code above has not been extensively tested; it is merely a proof of concept that demonstrates the idea of code generation. Simple measurements indicate that the code-generation-based approach is more than twice as fast as the original, Reflection-only code.

Generating Code Using Dynamic Lightweight Code Generation

Another example stems from the network protocol parsing domain. Suppose that you have a large stream of binary data, such as network packets, and you have to parse it to retrieve from it packet headers and select parts of the payload. For example, consider the following packet header structure (this is a completely fictional example—TCP packet headers are not arranged that way):

```
public struct TcpHeader {
  public uint SourceIP;
  public uint DestIP;
  public ushort SourcePort;
  public ushort DestPort;
  public uint Flags;
  public uint Checksum;
}
```

Retrieving such a structure from a stream of bytes is a trivial task in C/C++, and doesn't even require copying any memory if accessed by pointer. In fact, retrieving *any* structure from a stream of bytes would be trivial:

```
template < typename T>
const T* get_pointer(const unsigned char* data, int offset) {
  return (T*)(data + offset);
}
template < typename T>
const T get_value(const unsigned char* data, int offset) {
  return *get_pointer(data, offset);
}
```

In C# things are more complicated, unfortunately. There are numerous ways of reading arbitrary data from a stream. One possibility would be to use Reflection to go over the type's fields and read them from the stream of bytes individually:

```
//Supports only some primitive fields, does not recurse
public static void ReadReflectionBitConverter < T > (byte[] data, int offset, out T value) {
  object box = default(T);
  int current = offset;
  foreach (FieldInfo field in typeof(T).GetFields()) {
    if (field.FieldType == typeof(int)) {
       field.SetValue(box, BitConverter.ToInt32(data, current));
       current + = 4;
    } else if (field.FieldType == typeof(uint)) {
       field.SetValue(box, BitConverter.ToUInt32(data, current));
       current + = 4;
    } else if (field.FieldType == typeof(short)) {
       field.SetValue(box, BitConverter.ToInt16(data, current));
       current + = 2;
    } else if (field.FieldType == typeof(ushort)) {
       field.SetValue(box, BitConverter.ToUInt16(data, current));
       current + = 2;
    }
    //. . .many more types omitted for brevity
    value = (T)box;
}
```

When executed 1,000,000 times over a 20-byte `TcpHeader` structure on one of our test machines, this method took an average of 170ms to execute. Although the running time might seem not too bad, the amount of memory allocated by all the boxing operations is considerable. Additionally, if you consider a realistic network rate of 1Gb/s, it's reasonable to expect tens of millions of packets per second, which means we would have to spend most of our CPU time reading structures from the incoming data.

A considerably better approach involves the `Marshal.PtrToStructure` method, which is designed to convert an unmanaged chunk of memory to a managed structure. Using it requires pinning the original data to retrieve a pointer to its memory:

```
public static void ReadMarshalPtrToStructure < T > (byte[] data, int offset, out T value) {
  GCHandle gch = GCHandle.Alloc(data, GCHandleType.Pinned);
  try {
    IntPtr ptr = gch.AddrOfPinnedObject();
    ptr + = offset;
    value = (T)Marshal.PtrToStructure(ptr, typeof(T));
  } finally {
    gch.Free();
  }
}
```

This version fares considerably better, at an average of 39ms for 1,000,000 packets. This is a significant performance improvement, but `Marshal.PtrToStructure` still forces a heap memory allocation because it returns an `object` reference, and this is still not fast enough for tens of millions of packets per second.

In Chapter 8 we discussed C# pointers and unsafe code, and it seems to be a great opportunity to use them. After all, the C++ version is so easy precisely because it uses pointers. Indeed, the following code is much faster at 0.45ms for 1,000,000 packets—an incredible improvement!

```
public static unsafe void ReadPointer(byte[] data, int offset, out TcpHeader header) {
  fixed (byte* pData = &data[offset]) {
    header = *(TcpHeader*)pData;
  }
}
```

Why is this method so fast? Because the entity responsible for copying data around is no longer an API call like `Marshal.PtrToStructure`—it's the JIT compiler itself. The assembly code generated for this method can be inlined (indeed, the 64-bit JIT compiler chooses to do so) and can use 3-4 instructions to copy memory around (e.g., using the `MOVQ` instruction on 32-bit systems to copy 64 bits at a time). The only problem is that the `ReadPointer` method we devised is not generic, unlike its C++ counterpart. The knee-jerk reaction is to implement a generic version of it—

```
public static unsafe void ReadPointerGeneric < T > (byte[] data, int offset, out T value) {
  fixed (byte* pData = &data[offset]) {
    value = *(T*)pData;
  }
}
```

—which does not compile! Specifically, `T*` is not something you can write in C#, anywhere, because there is no generic constraint to guarantee that a pointer to `T` can be taken (and only blittable types, discussed in Chapter 8, can be pinned and pointed to). Because no generic constraint is available to express our intent, it would seem that we have to write a separate version of `ReadPointer` for each type, which is where code generation comes back into play.

TYPEDREFERENCE AND TWO UNDOCUMENTED C# KEYWORDS

Desperate times call for desperate measures, and in this case the desperate measures are pulling out two undocumented C# keywords, `__makeref` and `__refvalue` (supported by equally undocumented IL opcodes). Together with the `TypedReference` struct these keywords are used in some low-level interoperability scenarios with C-style variable length method argument lists (which require another undocumented keyword, `__arglist`).

`TypedReference` is a small struct that has two `IntPtr` fields—`Type` and `Value`. The `Value` field is a pointer to a value, which can be a value type or a reference type, and the `Type` field is its method table pointer. By creating a `TypedReference` that points to a value type's location, we can reinterpret memory in a strongly-typed fashion, as our scenario requires, and use the JIT compiler to copy memory, as in the `ReadPointer` case.

```
//We are taking the parameter by ref and not by out because we need to take its address,
//and __makeref requires an initialized value.
public static unsafe void ReadPointerTypedRef < T > (byte[] data, int offset, ref T value) {
   //We aren't actually modifying 'value' -- just need an lvalue to start with
   TypedReference tr = __makeref(value);
   fixed (byte* ptr = &data[offset]) {
```

```
        //The first pointer-sized field of TypedReference is the object address, so we
        //overwrite it with a pointer into the right location in the data array:
        *(IntPtr*)&tr = (IntPtr)ptr;
        //__refvalue copies the pointee from the TypedReference to 'value'
        value = __refvalue(tr, T);
      }
    }
```

This nasty compiler magic still has a cost, unfortunately. Specifically, the `__makeref` operator is compiled by the JIT compiler to call `clr!`

To avoid writing a separate copy of the `ReadPointer` method for each type, we'll use Lightweight Code Generation (the `DynamicMethod` class) to generate code. First, we inspect the IL generated for the `ReadPointer` method:

```
.method public hidebysig static void ReadPointer( uint8[] data, int32 offset, [out] valuetype TcpHeader& header) cil managed
{
  .maxstack 2
  .locals init ([0] uint8& pinned pData)
  ldarg.0
  ldarg.1
  ldelema uint8
  stloc.0
  ldarg.2
  ldloc.0
  conv.i
  ldobj TcpHeader

  stobj TcpHeader

  ldc.i4.0
  conv.u
  stloc.0
  ret
}
```

Now all we have to do is emit IL where `TcpHeader` is replaced by the generic type argument. In fact, thanks to the excellent *ReflectionEmitLanguage* plugin for .NET Reflector (available at http://reflectoraddins.codeplex.com/wikipage?title=ReflectionEmitLanguage), which converts methods to the `Reflection.Emit` API calls required to generate them, we don't even have to write the code by hand—although it admittedly required a few minor fixes:

```
static class DelegateHolder < T>
{
  public static ReadDelegate < T > Value;
  public static ReadDelegate < T > CreateDelegate() {
    DynamicMethod dm = new DynamicMethod("Read", null,
    new Type[] { typeof(byte[]), typeof(int), typeof(T).MakeByRefType() },
    Assembly.GetExecutingAssembly().ManifestModule);
    dm.DefineParameter(1, ParameterAttributes.None, "data");
    dm.DefineParameter(2, ParameterAttributes.None, "offset");
    dm.DefineParameter(3, ParameterAttributes.Out, "value");
    ILGenerator generator = dm.GetILGenerator();
    generator.DeclareLocal(typeof(byte).MakePointerType(), pinned: true);
    generator.Emit(OpCodes.Ldarg_0);
    generator.Emit(OpCodes.Ldarg_1);
    generator.Emit(OpCodes.Ldelema, typeof(byte));
    generator.Emit(OpCodes.Stloc_0);
    generator.Emit(OpCodes.Ldarg_2);
    generator.Emit(OpCodes.Ldloc_0);
    generator.Emit(OpCodes.Conv_I);
    generator.Emit(OpCodes.Ldobj, typeof(T));
    generator.Emit(OpCodes.Stobj, typeof(T));
    generator.Emit(OpCodes.Ldc_I4_0);
    generator.Emit(OpCodes.Conv_U);
    generator.Emit(OpCodes.Stloc_0);
    generator.Emit(OpCodes.Ret);
    Value = (ReadDelegate < T>)dm.CreateDelegate(typeof(ReadDelegate < T>));
    return Value;
  }
}
public static void ReadPointerLCG < T > (byte[] data, int offset, out T value)
{
  ReadDelegate < T > del = DelegateHolder < T > .Value;
  if (del == null) {
    del = DelegateHolder<T>.CreateDelegate();
  }
  del(data, offset, out value);
}
```

This version takes 1.05ms for 1,000,000 packets, which is more than twice as long as `ReadPointer`, but still more than two orders of magnitude faster than the original Reflection-based approach—another win for code generation. (The performance loss compared to `ReadPointer` is due the need to fetch the delegate from a static field, check for null, and invoke the method through a delegate.)