

LinksPlatform's Platform.Reflection.Sigil Class Library

./Platform.Reflection.Sigil/DelegateHelpers.cs

```

1  using System;
2  using System.Collections.Generic;
3  using Sigil;
4  using Platform.Exceptions;
5
6  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
7
8  namespace Platform.Reflection.Sigil
9  {
10     public static class DelegateHelpers
11     {
12         public static TDelegate Compile<TDelegate>(Action<Emit<TDelegate>> emitCode)
13         {
14             var @delegate = default(TDelegate);
15             try
16             {
17                 var emitter = Emit<TDelegate>.NewDynamicMethod();
18                 emitCode(emitter);
19                 @delegate = emitter.CreateDelegate();
20             }
21             catch (Exception exception)
22             {
23                 exception.Ignore();
24             }
25             finally
26             {
27                 if (EqualityComparer<TDelegate>.Default.Equals(@delegate, default))
28                 {
29                     var factory = new NotSupportedExceptionDelegateFactory<TDelegate>();
30                     @delegate = factory.Create();
31                 }
32             }
33             return @delegate;
34         }
35     }
36 }

```

./Platform.Reflection.Sigil/EmitExtensions.cs

```

1  using System;
2  using Sigil;
3
4  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
5
6  namespace Platform.Reflection.Sigil
7  {
8     public static class EmitExtensions
9     {
10         public static Emit<TDelegate> LoadConstantOne<TDelegate>(this Emit<TDelegate> emitter,
11             ↪ Type constantType)
12         {
13             if (constantType == typeof(float))
14             {
15                 emitter.LoadConstant(1F);
16             }
17             else if (constantType == typeof(double))
18             {
19                 emitter.LoadConstant(1D);
20             }
21             else if (constantType == typeof(long))
22             {
23                 emitter.LoadConstant(1L);
24             }
25             else if (constantType == typeof(ulong))
26             {
27                 emitter.LoadConstant(1UL);
28             }
29             else if (constantType == typeof(int))
30             {
31                 emitter.LoadConstant(1);
32             }
33             else if (constantType == typeof(uint))
34             {
35                 emitter.LoadConstant(1U);
36             }
37             else if (constantType == typeof(short))
38             {
39                 emitter.LoadConstant(1);
40             }
41         }
42     }
43 }

```

```

39         emitter.Convert<short>();
40     }
41     else if (constantType == typeof(ushort))
42     {
43         emitter.LoadConstant(1);
44         emitter.Convert<ushort>();
45     }
46     else if (constantType == typeof(sbyte))
47     {
48         emitter.LoadConstant(1);
49         emitter.Convert<sbyte>();
50     }
51     else if (constantType == typeof(byte))
52     {
53         emitter.LoadConstant(1);
54         emitter.Convert<byte>();
55     }
56     else
57     {
58         throw new NotSupportedException();
59     }
60     return emitter;
61 }
62
63 public static Emit<TDelegate> LoadConstant<TDelegate>(this Emit<TDelegate> emitter, Type
→ constantType, object constantValue)
64 {
65     if (constantType == typeof(float))
66     {
67         emitter.LoadConstant((float)constantValue);
68     }
69     else if (constantType == typeof(double))
70     {
71         emitter.LoadConstant((double)constantValue);
72     }
73     else if (constantType == typeof(long))
74     {
75         emitter.LoadConstant((long)constantValue);
76     }
77     else if (constantType == typeof(ulong))
78     {
79         emitter.LoadConstant((ulong)constantValue);
80     }
81     else if (constantType == typeof(int))
82     {
83         emitter.LoadConstant((int)constantValue);
84     }
85     else if (constantType == typeof(uint))
86     {
87         emitter.LoadConstant((uint)constantValue);
88     }
89     else if (constantType == typeof(short))
90     {
91         emitter.LoadConstant((short)constantValue);
92         emitter.Convert<short>();
93     }
94     else if (constantType == typeof(ushort))
95     {
96         emitter.LoadConstant((ushort)constantValue);
97         emitter.Convert<ushort>();
98     }
99     else if (constantType == typeof(sbyte))
100    {
101        emitter.LoadConstant((sbyte)constantValue);
102        emitter.Convert<sbyte>();
103    }
104    else if (constantType == typeof(byte))
105    {
106        emitter.LoadConstant((byte)constantValue);
107        emitter.Convert<byte>();
108    }
109    else
110    {
111        throw new NotSupportedException();
112    }
113    return emitter;
114 }
115

```

```

116 public static Emit<TDelegate> Increment<TDelegate>(this Emit<TDelegate> emitter, Type
    ↳ valueType)
117 {
118     emitter.LoadConstantOne(valueType);
119     emitter.Add();
120     return emitter;
121 }
122
123 public static Emit<TDelegate> Decrement<TDelegate>(this Emit<TDelegate> emitter, Type
    ↳ valueType)
124 {
125     emitter.LoadConstantOne(valueType);
126     emitter.Subtract();
127     return emitter;
128 }
129
130 public static Emit<TDelegate> LoadArguments<TDelegate>(this Emit<TDelegate> emitter,
    ↳ params ushort[] arguments)
131 {
132     for (var i = 0; i < arguments.Length; i++)
133     {
134         emitter.LoadArgument(arguments[i]);
135     }
136     return emitter;
137 }
138
139 public static Emit<TDelegate> CompareGreaterThan<TDelegate>(this Emit<TDelegate> emitter,
    ↳ bool isSigned)
140 {
141     if (isSigned)
142     {
143         emitter.CompareGreaterThan();
144     }
145     else
146     {
147         emitter.UnsignedCompareGreaterThan();
148     }
149     return emitter;
150 }
151
152 public static Emit<TDelegate> CompareLessThan<TDelegate>(this Emit<TDelegate> emitter,
    ↳ bool isSigned)
153 {
154     if (isSigned)
155     {
156         emitter.CompareLessThan();
157     }
158     else
159     {
160         emitter.UnsignedCompareLessThan();
161     }
162     return emitter;
163 }
164
165 public static Emit<TDelegate> BranchIfGreaterOrEqual<TDelegate>(this Emit<TDelegate>
    ↳ emitter, bool isSigned, Label label)
166 {
167     if (isSigned)
168     {
169         emitter.BranchIfGreaterOrEqual(label);
170     }
171     else
172     {
173         emitter.UnsignedBranchIfGreaterOrEqual(label);
174     }
175     return emitter;
176 }
177
178 public static Emit<TDelegate> BranchIfLessOrEqual<TDelegate>(this Emit<TDelegate>
    ↳ emitter, bool isSigned, Label label)
179 {
180     if (isSigned)
181     {
182         emitter.BranchIfLessOrEqual(label);
183     }
184     else
185     {
186         emitter.UnsignedBranchIfLessOrEqual(label);

```

```

187         }
188         return emitter;
189     }
190 }
191 }

```

./Platform.Reflection.Sigil/NotSupportedExceptionDelegateFactory.cs

```

1 using System;
2 using Sigil;
3 using Platform.Interfaces;
4
5 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
6
7 namespace Platform.Reflection.Sigil
8 {
9     public class NotSupportedExceptionDelegateFactory<TDelegate> : IFactory<TDelegate>
10     {
11         public TDelegate Create()
12         {
13             var emitter = Emit<TDelegate>.NewDynamicMethod();
14             emitter.NewObject<NotSupportedException>();
15             emitter.Throw();
16             return emitter.CreateDelegate();
17         }
18     }
19 }

```

./Platform.Reflection.Sigil.Tests/InlineTests.cs

```

1 using System;
2 using System.Linq;
3 using System.Reflection;
4 using System.Reflection.Emit;
5 using Xunit;
6 using TheSigil = Sigil;
7
8 namespace Platform.Reflection.Sigil.Tests
9 {
10     public static class InlineTests
11     {
12         [Fact]
13         public static void SimpleInlineTest()
14         {
15             var disassembledOuterMethod = TheSigil.Disassembler<Action>.Disassemble(OuterMethod);
16             var emitter = TheSigil.Emit<Action>.NewDynamicMethod();
17
18             foreach (var operation in disassembledOuterMethod)
19             {
20                 if (operation.Opcode == OpCodes.Nop)
21                 {
22                     continue;
23                 }
24                 if (operation.Opcode == OpCodes.Call)
25                 {
26                     var firstParameter = operation.Parameters.First();
27                     if (firstParameter is MethodInfo methodInfo)
28                     {
29                         if (methodInfo.Name == nameof(InnerMethod))
30                         {
31                             var disassembledInnerMethod =
32                                 ↪ TheSigil.Disassembler<Action>.Disassemble(InnerMethod);
33                             var i = 0;
34                             foreach (var innerOperation in disassembledInnerMethod)
35                             {
36                                 // There is no way to replay operation at emitter
37                                 // emitter.Replay(innerOperation)
38
39                                 // There is also no way to rewrite operations in the
40                                 ↪ disassembledOuterMethod
41                                 // disassembledOuterMethod[i++] = innerOperation;
42                             }
43
44                             // So the only way (but it is not practical for now) is to use:
45                             emitter = disassembledInnerMethod.EmitAll();
46                             // That means we just use InnerMethod method directly,
47                             // but if OuterMethod will contain something else we will fail with
48                             ↪ current support of disassembling in the Sigil.
49                         }
50                     }
51                 }
52             }
53         }
54     }
55 }

```

```
49         }
50
51         Action result = emitter.CreateDelegate();
52         result();
53     }
54
55     private static void InnerMethod()
56     {
57         Console.WriteLine("Inner method.");
58     }
59
60     private static void OuterMethod()
61     {
62         InnerMethod();
63     }
64 }
65 }
```

Index

./Platform.Reflection.Sigil.Tests/InlineTests.cs, 4
./Platform.Reflection.Sigil/DelegateHelpers.cs, 1
./Platform.Reflection.Sigil/EmitExtensions.cs, 1
./Platform.Reflection.Sigil/NotSupportedExceptionDelegateFactory.cs, 4