Cecil.Inject
Porting code from ReiPatcherPlus

Geoffrey "denikson" Horsington
Github
HongFire

October 10, 2015

1 Abstract

ReiPatcherPlus is an extension library to ReiPatcher that provides simple injection methods for Mono.Cecil. The initial versions (1.x) provide very basic means of hooking and redirecting IL instruction execution. The updated versions (2.x) also allow to explicitly search for methods and hooks based on a method's name, parameters and additional flags. However, ReiPatcherPlus is lacking in several ways:

- Lack of additional methods, like method search by `System.Type` parameters and more advanced access changing methods
- Poor method hierarchy and old methods left for the sake of compatibility
- Dependency on ReiPatcher even though almost no methods use it

To address these issues, nearly every method in ReiPatcherPlus was revised, rewritten and fixed where needed. Moreover, the dependency on ReiPatcher was removed and all functionality moved to separate classes to extend Mono.Cecil. Thus, being a massive update and an extension to Mono.Cecil rather than ReiPatcher, the library was renamed to Mono.Cecil.Inject (hereafter Cecil.Inject) to better represent the purpose the library.

This document is a quick reference guide for those who come from ReiPatcherPlus and wish to port their code to Cecil.Inject. Only the most crucial changes are outlined and only a basic set of new methods are shown. Old methods from ReiPatcherPlus and new ones from Cecil.Inject are shown side-by-side for easier comparison. The exact definitions of *some* new methods are picked from the official documentation of Cecil.Inject and are displayed in framed boxes. For the complete list of all methods in Cecil.Inject, refer to the official documentation.

2 Importing the library

The easiest way to begin migration is to remove the reference to ReiPatcherPlus, add the reference to Cecil.Inject and replace the old namespace import with the new one:

```
using Mono.Cecil.Inject;
```

Attempting to compile the project will cause numerous errors; use them to find the code that requires migration.

3 Loading assemblies

ReiPatcherPlus provides the following method to load assemblies into Mono.Cecil:

```
AssemblyDefinition LoadAssembly(this PatchBase patch, string name);
```

As the method signature suggests, the method is an extension to `PatchBase` class found in ReiPatcher. Because of removed dependency on ReiPatcher, the functionality of the method was changed a bit. The new definition found in Cecil.Inject is as follows:

METHOD DEFINITION

In `AssemblyLoader`:

```
public static AssemblyDefinition LoadAssembly(string path);
```

Description

Loads the assembly specified by path into Mono.Cecil.

Parameters

- `path` – Path to the assembly to load. Can be absolute or relative. In the latter case the path must be relative to the executing assembly which invokes this method.

Returns

An instance of `AssemblyDefinition` of the loaded assembly.

Examples

```
AssemblyDefinition ad1 =
↪  AssemblyLoader.LoadAssembly("SBPR.HelloWorld.dll");

// NOTE: This is equivalent of ReiPatcherPlus' method call:
// this.LoadAssembly("CM3D2.HelloWorld.Hook.dll");
AssemblyDefinition ad2 =
↪  AssemblyLoader.LoadAssembly(Path.Combine(AssembliesDir,
↪  "CM3D2.HelloWorld.Hook.dll"));
```

Note especially that

- This method is not an extension any more and therefore must be referred to with the class name (`AssemblyLoader`)
- The root of the relative path is now the directory of the executing assembly (in case of ReiPatcher it is the directory ReiPatcher.exe lies in) instead of the game assembly directory. Remember to use `Path.Combine` with the property `AssembliesDir` provided by `PatchBase` in order to achieve the same functionality as before.

4 Checking patch flags

ReiPatcherPlus provides the following method to check for patch attributes:

```
bool HasAttribute(this PatchBase patch,
                  AssemblyDefinition assembly,
                  string attribute);
```

In Cecil.Inject the method **has been removed** because it was only usable for ReiPatcher. As of this moment no features to add similar functionality are planned. Instead, you can use the following method to replicate the behaviour of `HasAttribute` found in ReiPatcherPlus:

```
public static bool HasAttribute(PatchBase patch,
                                AssemblyDefinition assembly,
                                string attribute)
{
        return patch.GetPatchedAttributes(assembly)
                    .Any(a => a.Info == attribute);
}
```

5 `TypeDefinition` extensions

Many of the extension methods for `TypeDefinition` of Mono.Cecil were ported from ReiPatcherPlus to Cecil.Inject. Namely, the following familiar methods are found in Cecil.Inject:

```
MethodDefinition GetMethod(this TypeDefinition self, string methodName);
MethodDefinition GetMethod(this TypeDefinition self,
                           string methodName,
                           params TypeReference[] paramTypes);
FieldDefinition GetField(this TypeDefinition self, string memberName);
```

In addition, `ChangeAccess` has been moved, fixed and has some new parameters. Because of this, we shall take a closer look at it

METHOD DEFINITION

Extension to `TypeDefinition` (in `TypeDefinitionExtensions`):

```
public static void ChangeAccess(this TypeDefinition type,
                                string member,
                                bool makePublic = true,
                                bool makeVirtual = true,
                                bool makeAssignable = true,
                                bool recursive = false);
```

Description

A method to quickly make any class member public and writeable.

Parameters

- `type` – type the members of which to modify. Will be inferred by the compiler if used as an extension method.
- `member` – the name of the member to modify. Can be either set to a literal name or a regular expression.
- `makePublic` – make the member public if it isn't already.
  Default: `true`
- `makeVirtual` – make the member virtual, if it is a method.
  Default: `true`
- `makeAssignable` – make the member assignable, if it is marked as `readonly`.
  Default: `true`
- `recursive` – If set to `true` will recursively apply this method to every nested class in the type.
  Default: `false`

Remarks

Note that you can now select the members with regular expressions. Therefore, if your `name` contains some special characters used by RegEx (like dots), they are treated as a regular expression. To avoid that, remember to escape those special characters.

The method does not make the base type (specified in `type` parameter) public, since it can be done easily by using Mono.Cecil itself.

Examples

```
TypeDefinition myType = myAssembly.MainModule.GetType("MyType");

// This is the intended way to use this method
myType.ChangeAccess("someMember1");

// This also works, but is not advised
TypeDefinitionExtensions.ChangeAccess(myType, "someMember1");
```

```
// You can use RegEx to match multiple members
// You can also set one or many optional parameter(s) by either
↪  referencing them by name in any order or by assigning them values in
↪  the order they appear in the signature
myType.ChangeAccess("someMember.*", makeAssignable: false);
```

To make Cecil.Inject more compatible with System.Reflection, new methods were added. Here are some of them

METHOD DEFINITION

Extension to `TypeDefinition` (in `TypeDefinitionExtensions`):

```
public static MethodDefinition GetMethod(this TypeDefinition self,
                                         string methodName,
                                         params Type[] types);
```

Description

Searches for the method with the given name and parameter types.

Parameters

- `self` – type that contains the method to search. Will be inferred by the compiler if used as an extension method.
- `methodName` – name of the method to find.
- `types` – type of parameters in the method to search. Must be specified in the order they are defined in the method signature.

Returns

An instance of `MethodDefinition` for the found method. Returns `null`, if no fitting method found.

Remarks

Use `System.Type` to construct types.

If generic types are needed, use `ParamHelper.CreateDummyType` to easily create parameters with custom names.

Examples

```
TypeDefinition myType = myAssembly.MainModule.GetType("MyType");

// Searches for method signature: MyMethod1(int, ref bool) (or
↪  MyMethod1(int, out bool))
MethodDefiniton md1 = myType.GetMethod("MyMethod1",
                               typeof(int),
                               typeof(bool).MakeByReferenceType());

// Searches for method signature: MyMethod2<T>(List<T>);
MethodDefinition md2 =
  myType.GetMethod("MyMethod2",
     typeof(List<>).MakeGenericType(ParamHelper.CreateDummyType("T")));
```

There is also a version to retrieve all the overloads of a method

METHOD DEFINITION

Extension to `TypeDefinition` (in `TypeDefinitionExtensions`):

```
public static MethodDefinition[] GetMethods(this TypeDefinition self,
                                            string methodName);
```

Description

Retrieves all the overloads of the given method name.

Parameters

- `self` – type that contains the methods to search. Will be inferred by the compiler if used as an extension method.
- `methodName` – name of the method to search for.

Returns

An array of `MethodDefinition` containing all the methods the name of which is `methodName`.

Examples

```
TypeDefinition myType = myAssembly.MainModule.GetType("MyType");

// Gets all overloads of MyMethod
MethodDefinition[] mds = myType.GetMethods("MyMethod");
```

6 `Parameter` is now `ParamHelper`

To create generic types for `GetMethod`, ReiPatcherPlus has a helper class `Parameter`. In Cecil.Inject, such class still exists, albeit with some of the methods having been removed so as to force the developers to prefer the newer version of `GetMethod` described in the previous section.

More exactly, the following methods were ported from ReiPatcherPlus:

```
public static TypeReference CreateGeneric(string name);
public static TypeReference FromType<T>();
public static TypeReference FromType(Type type);
```

Note that `FromType` takes only one argument now. In order to construct reference and/or generic types, use the methods provided by `System.Type`. However, to aid the creation of type with generic types (like `List<T>`), one can use the following method in `ParamHelper`:

```
public static Type CreateDummyType(string name);
```

which allows to create a generic type with a custom name.

7 `MethodHook` is now `InjectionDefinition`

Just like in ReiPatcherPlus, method injection is the main feature of Cecil.Inject. Therefore, more care was put into reorganising the injection methods. The main differences and updates are found in the following subsections.

7.1 `MethodFeatures` is `InjectFlags`

`MethodFeatures` – the enumeration containing flags that specify how to inject method – has been renamed into `InjectFlags` to better represent its meaning. Additionally, the flags have been renamed to show their effect on the injection process more clearly. To summarise, below is the table that shows how the names have been changed.

Old flag name in `MethodFeatures`	New flag name in `InjectFlags`
`None`	`None`
`PassCustomTag`	`PassTag`
`PassTargetType`	`PassInvokingInstance`
`PassReturn`	`ModifyReturn`
`PassMemberReferences`	`PassFields`
`PassLocalReferences`	`PassLocals`
`PassMethodParametersByValue`	`PassParametersVal`
`PassMethodParametersByReference`	`PassParametersRef`
`All_ByValue`	`All_Val`
`All_ByReference`	`All_Ref`

Other than changed names the flags are fully identical.

7.2 Extensions to `InjectFlags`

One of the new additions to Cecil.Inject is the ability ease the construction and manipulation of `InjectFlags`. Namely, the following extension methods were added to `InjectFlags`:

```
public static bool IsSet(this InjectFlags flags, InjectFlags flag);
public static InjectValues ToValues(this InjectFlags flags);
```

Moreover, a structure `InjectValues` was added that can be used to represent `InjecFlags` as boolean parameters and vice versa.

We shall omit the exact documentation – feel free to experiment yourself or consult the official documentation.

7.3 Obtaining `InjectionDefinition`

To get an instance of `InjectionDefinition`, you can do one of the following

- Create an instance through the constructor of `InjectionDefinition` (equivalent to ReiPatcherPlus' `MethodHook.FromMethodDefinition`)
- Use the extension method to `TypeDefintion` which is `GetInjectionMethod` (equivalent to ReiPatcherPlus' `GetHookMethod`).
- Use `GetInjector` (extension method to `MethodDefinition`) to get an instance of `InjectionDefinition`. Like `GetInjectionMethod`, the method searches for a fitting hook in the provided `type`.

The exact method signatures are the same as in ReiPatcherPlus and therefore they shall be omitted. It must be noted, however, that the behaviour of `GetInjectionMethod` differes a little bit from that of `GetHookMethod`. Namely

1. The method is an extension to `TypeDefinition` now. Therefore, `hookType` is the first parameter that is inferred by the compiler when you call the method on some `TypeDefinition`.
2. The method does not attempt to fix obvious issues with the specified `InjectFlags`. Instead, when any mismatch occurs, the method simply returns `null`.

7.4 Injection

Owing to compatibility, ReiPatcherPlus contains a plethora of different overloads intended for method injection. In Cecil.Inject, all the methods were combined into two. Since they are the most important methods, here are their full definitions:

METHOD DEFINITION

In `InjectionDefinition`

```
public void Inject(int startCode = 0,
                   int token = 0,
                   InjectDirection direction = InjectDirection.Before);

public void Inject(Instruction startCode,
                   int token = 0,
                   InjectDirection direction = InjectDirection.Before);
```

Description

Applies the injection specified by `InjectionDefinition`.

Parameters

- ▷ `startCode` – the instruction in the method from which to start to inject. Can be specified either as an index (integer) or as an `Instruction`.
- ▷ `token` – if the hook method can receive custom tags (specified with `InjecFlags.PassTag`), this is the tag that will be passed to it.
- ▷ `direction` – the direction in which to inject the method call. The injection can be either done before the specified `startCode` or after it.

Remarks

If `startCode` is specified as an index, it can be either positive (specifies code from the top of the method) or negative (specifies code from the end of the method).

Examples

```
TypeDefinition myHooks = myAssembly.GetType("MyHooks);
TypeDefinition myTarget = targetAssembly.GetType("TargetType");

MethodDefinition target = myTarget.GetMethod("MyTargetMethod");

// Searches for hook with signature MyHookMethod() and injects it in
↪  front of the first instruction of target method
myHooks.GetInjectionMethod("MyHookMethod",
                           target,
                           InjectFlags.None).Inject();

// ALTERNATIVE: Create InjectionDefinition manually
InjectionDefinition id =
     new InjectionDefinition(target,
                             myHooks.GetMethod("MyHookMethod"),
                             InjectFlags.None);
id.Inject();
```

Alternatively, you can use the following extension to `MethodDefinition` to inject with a single expression.

METHOD DEFINITION

Extension to `MethodDefinition` (in `MethodDefinitionExtensions`):

```
public static void InjectWith(this MethodDefinition method,
                        MethodDefinition injectionMethod,
                        int codeOffset = 0,
                        int tag = 0,
                        InjectFlags flags = InjectFlags.None,
                        InjectDirection dir = InjectDirection.Before,
                        int[] localsID = null,
                        FieldDefinition[] typeFields = null);
```

Descriptipn

Creates an instance of `InjectionDefinition` and calls `Inject`.

Parameters

- `method` – the method which to inject. Will be inferred by the compiler if used as an extension method.
- `injectionMethod` – the hook method. The call to this method will be injected into `method`.
- `codeOffset` – the index of the instruction in `method` from which to start injecting.

- ▷ `tag` – if the hook method can receive custom tags (specified with `InjecFlags.PassTag`), this is the tag that will be passed to it.
- ▷ `flags` – the flags which specify what and how to inject the method. Different flags can be combined with a logical OR (`|`) or by using `InjectValues`.
- ▷ `dir` – the direction in which to inject the method call. The injection can be either done before the specified `startCode` or after it.
- ▷ `localID` – an array of target method's local indices to pass to the hook. Used only if `InjectFlags.PassLocals` is specified.
- ▷ `typeFields` – Fields to pass to the hook. Used only if `InjectFlags.PassFields` is specified.

Remarks

Acts exactly like `AttachMethod` in ReiPatcherPlus.

Examples

```
TypeDefinition myHooks = myAssembly.GetType("MyHooks);
TypeDefinition myTarget = targetAssembly.GetType("TargetType");

// Injects MyTargetMethod with (for example) MyHook1(ref int myLocal, ref
↪  int tField)
myTarget.GetMethod("MyTargetMethod").InjectWith(
                    myHooks.GetMethod("MyHook1"),
                    flags: InjectFlags.PassLocals
                         | InjectFlags.PassFields,
                    localsID: new[] {0},
                    typeFields: new[] {myTarget.GetField("tField")});
```

8 Summary

We discussed the most important changes between ReiPatcherPlus and Cecil.Inject thus making migration between the libraries relatively easy. If you wish to inspect the methods of Cecil.Inject more thoroughly and get a better explanation on how to use the library, consider reading the full documentation of Cecil.Inject.