Getting all types that implement an interface

497

131

Using reflection, how can I get all types that implement an interface with C# 3.0/.NET 3.5 with the least code, and minimizing iterations?

This is what I want to re-write:

```
foreach (Type t in this.GetType().Assembly.GetTypes())
    if (t is IMyInterface)
        ; //do stuff
```

c# optimization reflection lambda c#-3.0

edited Sep 30 '14 at 12:20

Ondrej Janacek
9,668 13 46 83

asked Aug 25 '08 at 19:57

juan
39.9k 46 145 178

1 Does the example code works? I've got false negatives with your if condition. – Emperor Orionii Dec 15 '12 at 15:20

3 The if statement in the code above will always be false because you are testing if an instance of the Type class (t) implements your interface which it won't unless Type inherits IMyInterface (in which case it will always be true). – Liazy Jun 26 '13 at 12:09

13 Answers

734

Mine would be this in c# 3.0 :)

```
var type = typeof(IMyInterface);
var types = AppDomain.CurrentDomain.GetAssemblies()
    .SelectMany(s => s.GetTypes())
```

```
loop assemblies
 loop types
  see if implemented.
```

edited Sep 6 '13 at 18:41

answered Aug 25 '08 at 20:11

Darren Kopp
61.8k 9 67 87

174 Note that the list may also include the interface itself. Change the last line to `.Where(p => type.IsAssignableFrom(p) && !p.IsInterface);` to filter it out (or `p.IsClass`). – jtpereyda Dec 2 '13 at 21:21

36 Note:This answer is wrong!, this checks "Assignment compatibility" not whether interface is implemented are not. For example `List<string>` doesn't implement `IEnumerable<object>` but this method will return true in .Net 4.0 due to covariance which is wrong indeed. Correct answer is here – Sriram Sakthivel Apr 8 '14 at 7:29

18 @SriramSakthivel first off, generic values weren't specified. Second, this question pre-dates covariance. Third, you make the assumption that covariant return is not something they want. – Darren Kopp Apr 8 '14 at 13:45

23 You're absolutely right darren, I know this is a old thread, I just registered my comment just for the future users to make aware of such problem exist. Not to offend you. and as question title says if OP is asking for *Getting all types that implement an interface* this code isn't doing that. but almost all the cases **it works**, no doubt. there are corner cases too as I said. Just to be aware of it; – Sriram Sakthivel Apr 8 '14 at 16:13

9 Will also need to make sure the class isn't abstract => `.Where(p => type.IsAssignableFrom(p) && p.IsClass && !p.IsAbstract` – Jonesopolis Dec 23 '15 at 18:11

58

This worked for me. It loops though the classes and checks to see if they are derrived from myInterface

```
foreach (Type mytype in System.Reflection.Assembly.GetExecutingAssembly().GetTypes()
                .Where(mytype => mytype
.GetInterfaces().Contains(typeof(myInterface)))) {
    //do stuff
}
```

edited Aug 11 '17 at 15:06

answered Sep 26 '12 at 12:55

is better to have the framework do the heavy lifting. Then filter down farther when found. If relevant, please update your answer. Include List<T> reasoning. var classTypesImplementingInterface = AppDomain.CurrentDomain.GetAssemblies().SelectMany(x => x.GetTypes()).Where(mytype => typeof(myInterface).IsAssignableFrom(mytype) && mytype.GetInterfaces().Contains(typeof(myInterface))); foreach(var item in items) Console.Log(item.Name); – TamusJRoyce Aug 29 '14 at 2:37

54

To find all types in an assembly that implement IFoo interface:

```
var results = from type in someAssembly.GetTypes()
              where typeof(IFoo).IsAssignableFrom(type)
              select type;
```

Note that Ryan Rinaldi's suggestion was incorrect. It will return 0 types. You cannot write

```
where type is IFoo
```

because type is a System.Type instance, and will never be of type IFoo. Instead, you check to see if IFoo is assignable from the type. That will get your expected results.

Also, Adam Wright's suggestion, which is currently marked as the answer, is incorrect as well, and for the same reason. At runtime, you'll see 0 types come back, because all System.Type instances weren't IFoo implementors.

edited Aug 25 '08 at 20:44

answered Aug 25 '08 at 20:20

Judah Gabriel Himango

43.4k 32 145 200

46

I appreciate this is a very old question but I thought I would add another answer for future users as all the answers to date use some form of `Assembly.GetTypes` .

Whilst GetTypes() will indeed return all types, it does not necessarily mean you could activate them and could thus potentially throw a `ReflectionTypeLoadException` .

```
Class A // in AssemblyA
Class B : Class A, IMyInterface // in AssemblyB
Class C // in AssemblyC which references AssemblyB but not AssemblyA
```

If in `ClassC` which is in `AssemblyC` we then do something as per accepted answer:

```
var type = typeof(IMyInterface);
var types = AppDomain.CurrentDomain.GetAssemblies()
    .SelectMany(s => s.GetTypes())
    .Where(p => type.IsAssignableFrom(p));
```

Then it will throw a `ReflectionTypeLoadException` .

This is because without a reference to `AssemblyA` in `AssemblyC` you would not be able to:

```
var bType = typeof(ClassB);
var bClass = (ClassB)Activator.CreateInstance(bType);
```

In other words `ClassB` is not *loadable* which is something that the call to GetTypes checks and throws on.

So to safely qualify the result set for loadable types then as per this Phil Haacked article Get All Types in an Assembly and Jon Skeet code you would instead do something like:

```
public static class TypeLoaderExtensions {
    public static IEnumerable<Type> GetLoadableTypes(this Assembly assembly) {
        if (assembly == null) throw new ArgumentNullException("assembly");
        try {
            return assembly.GetTypes();
        } catch (ReflectionTypeLoadException e) {
            return e.Types.Where(t => t != null);
        }
    }
}
```

And then:

edited May 23 '17 at 11:54
Community ♦
1 1

answered Mar 31 '15 at 22:41
rism
8,235 13 59 100

3 This helped me deal with a super weird problem, where in my test project GetTypes would fail and only in our CI-environment. GetLoadableTypes was a fix for this solution. The error wouldn't be reproducible in the local environment and it was this: System.Reflection.ReflectionTypeLoadException: Unable to load one or more of the requested types. Retrieve the LoaderExceptions property for more information. More specifically it was complaining that there was a type that didn't have a concrete implementation and it happened in the unit test project. Thanks for this! – Lari Tuomisto Nov 18 '15 at 7:44

1 This answer should be marked as solution, it saved my ass today, because like @Lari Tuomisto said, on local env we couldn't re-product similar error – Lightning3 Dec 29 '15 at 23:19

2 In case it helps someone else: this solution worked for me, but I had to modify it to remove the interface type from the list. I wanted to activate `CreateInstance` for all of them, and an exception was thrown when it was trying to create the actual interface (which had me confused for a while when I thought the actual interface was out of the way in this solution). So I changed the code to `GetLoadableTypes(assembly).Where(interfaceType.IsAssignableFrom).Where(t => !(t.Equals(interfaceType))).ToList(); .` – Xavier Peña Aug 7 '16 at 17:38

19

Other answers here use `IsAssignableFrom`. You can also use `FindInterfaces` from the `System` namespace, as described here.

Here's an example that checks all assemblies in the currently executing assembly's folder, looking for classes that implement a certain interface (avoiding LINQ for clarity).

```
static void Main() {
    const string qualifiedInterfaceName = "Interfaces.IMyInterface";
    var interfaceFilter = new TypeFilter(InterfaceFilter);
    var path = Path.GetDirectoryName(Assembly.GetExecutingAssembly().Location);
    var di = new DirectoryInfo(path);
    foreach (var file in di.GetFiles("*.dll")) {
        try {
            var nextAssembly = Assembly.ReflectionOnlyLoadFrom(file.FullName);
            foreach (var type in nextAssembly.GetTypes()) {
                var myInterfaces = type.FindInterfaces(interfaceFilter,
qualifiedInterfaceName);
                if (myInterfaces.Length > 0) {
                    // This class implements the interface
```

```
}

public static bool InterfaceFilter(Type typeObj, Object criteriaObj) {
    return typeObj.ToString() == criteriaObj.ToString();
}
```

You can set up a list of interfaces if you want to match more than one.

edited Jun 7 '17 at 12:19

answered Jun 8 '12 at 11:18

hillstuk

901 8 12

This one looks for string interface name which is what I was looking for. – senthil Jan 9 '13 at 16:54

Works when loading an assembly in a different domain, as the type has to be serialized into a string. very awesome! – TamusJRoyce May 20 '15 at 4:12

17

loop through all loaded assemblies, loop through all their types, and check if they implement the interface.

something like:

```
Type ti = typeof(IYourInterface);
foreach (Assembly asm in AppDomain.CurrentDomain.GetAssemblies()) {
    foreach (Type t in asm.GetTypes()) {
        if (ti.IsAssignableFrom(t)) {
            // here's your type in t
        }
    }
}
```

edited Apr 28 '17 at 12:40

burræquete

5,685 4 23 51

answered Aug 25 '08 at 20:05

Lasse Vågsæther Karlsen

299k 86 536 728

8

```
Type lookupType = typeof (IMenuItem);
IEnumerable<Type> lookupTypes = GetType().Assembly.GetTypes().Where(
        t => lookupType.IsAssignableFrom(t) && !t.IsInterface);
```

edited Jun 29 '11 at 9:55

takrl

5,423 3 51 62

answered Nov 12 '10 at 14:30

Carl Nayak

81 1 1

5

Edit: I've just seen the edit to clarify that the original question was for the reduction of iterations / code and that's all well and good as an exercise, but in real-world situations you're going to want the fastest implementation, regardless of how cool the underlying LINQ looks.

Here's my Utils method for iterating through the loaded types. It handles regular classes as well as interfaces, and the excludeSystemTypes option speeds things up hugely if you are looking for implementations in your own / third-party codebase.

```
public static List<Type> GetSubclassesOf(this Type type, bool excludeSystemTypes) {
    List<Type> list = new List<Type>();
    IEnumerator enumerator = Thread.GetDomain().GetAssemblies().GetEnumerator();
    while (enumerator.MoveNext()) {
        try {
            Type[] types = ((Assembly) enumerator.Current).GetTypes();
            if (!excludeSystemTypes || (excludeSystemTypes && !((Assembly)
enumerator.Current).FullName.StartsWith("System."))) {
                IEnumerator enumerator2 = types.GetEnumerator();
                while (enumerator2.MoveNext()) {
                    Type current = (Type) enumerator2.Current;
                    if (type.IsInterface) {
                        if (current.GetInterface(type.FullName) != null) {
                            list.Add(current);
                        }
                    } else if (current.IsSubclassOf(type)) {
                        list.Add(current);
                    }
                }
            }
        } catch {
        }
```

edited Apr 28 '17 at 12:40

buræquete
5,685 4 23 51

answered Aug 25 '08 at 20:12

tags2k
30k 30 71 100

2 Enumerators implement IDisposable which isn't being disposed in a try/finally. It is better to use a foreach or linq. – TamusJRoyce May 20 '15 at 4:09

4

Other answer were not working with a **generic interface.**

This one does, just replace typeof(ISomeInterface) by typeof (T).

```
List<string> types = AppDomain.CurrentDomain.GetAssemblies().SelectMany(x =>
x.GetTypes())
            .Where(x => typeof(ISomeInterface).IsAssignableFrom(x) && !x.IsInterface &&
!x.IsAbstract)
            .Select(x => x.Name).ToList();
```

So with

```
AppDomain.CurrentDomain.GetAssemblies().SelectMany(x => x.GetTypes())
```

we get all the assemblies

```
!x.IsInterface && !x.IsAbstract
```

is used to exclude the interface and abstract ones and

```
.Select(x => x.Name).ToList();
```

to have them in a list.

It is not superior or lower, the other answers didn't work for me and I bothered to share it. – Antonin GAVREL Sep 20 '18 at 8:24

My comment was just about your answer being code-only, so I asked you to add some explanation. – Lukas Körfer Sep 20 '18 at 8:37

yes no worries, I added it :) have a good day~ – Antonin GAVREL Sep 20 '18 at 8:48

2

There's no easy way (in terms of performance) to do what you want to do.

Reflection works with assemblys and types mainly so you'll have to get all the types of the assembly and query them for the right interface. Here's an example:

```
Assembly asm = Assembly.Load("MyAssembly");
Type[] types = asm.GetTypes();
Type[] result = types.where(x => x.GetInterface("IMyInterface") != null);
```

That will get you all the types that implement the IMyInterface in the Assembly MyAssembly

answered Aug 25 '08 at 20:13

Jorge Córdoba
32.6k 9 71 119

1

Even better when choosing the Assembly location. Filter most of the assemblies if you know all your implemented interfaces are within the same Assembly.DefinedTypes.

```
// We get the assembly through the base class
var baseAssembly = typeof(baseClass).GetTypeInfo().Assembly;

// we filter the defined classes according to the interfaces they implement
var typeList = baseAssembly.DefinedTypes.Where(type =>
type.ImplementedInterfaces.Any(inter => inter == typeof(IMyInterface))).ToList();
```

I got exceptions in the linq-code so I do it this way (without a complicated extension):

0

```
private static IList<Type> loadAllTypes(Types[] interfaces)
{
    IList<Type> objects = new List<Type>();

    // find all types
    foreach (var interfaceType in interfaces)
        foreach (var currentAsm in AppDomain.CurrentDomain.GetAssemblies())
            try
            {
                foreach (var currentType in currentAsm.GetTypes())
                    if (interfaceType.IsAssignableFrom(currentType) &&
currentType.IsClass && !currentType.IsAbstract)
                        objects.Add(currentType);
            }
            catch { }

    return objects;
}
```

answered Feb 27 '18 at 10:47

user6537157
391 2 15

You could use some LINQ to get the list:

-3

```
var types = from type in this.GetType().Assembly.GetTypes()
            where type is ISomeInterface
            select type;
```

But really, is that more readable?

6 It might be more readable, if it worked. Unfortunately, your where clause is checking to see if an instance of the System.Type class implements ISomeInterface, which will never be true, unless ISomeInterface is really IReflect or ICustomAttributeProvider, in which case it will always be true. – Joel Mueller May 29 '09 at 20:22

Carl Nayak answer above has the answer to correcting the where clause: IsAssignableFrom. Easy mistake for an answer. – TamusJRoyce May 20 '15 at 4:06