C attribute reflection performance

14 December 2016 - 3347 words It's common knowledge that the reflection on .NET is slow, but why is it so? This post is designed to figure out that looking at what reflection does under-the-hood. CLR Type System Design Goals But first it is worth pointing out that part of the reason for the reflection is not fast is that it was never designed to have high performance as one of its goals, from Type System Overview – Design goals and non-objectives: Goals Access to information needed runtime from execution (non-reflection) code is very fast. Accessing the information needed to generate code during compilation is easy. Garbage collector/stackwalker has access to the necessary information without locks or separating memory. Minimum types are loaded at the same time. Certain types of minimum quantities are loaded during the type loading. The data structures of the type system shall be recorded in NGEN images. Non-purpose All data contained in metadata is directly reflected in clr data structures. All reflection uses are fast. and in the same way from Type Loader Design - Key Data Structures: EEClass MethodTable data is divided into hot and cold structures to improve work sets and cache usage. MethodTable itself is only intended to store hot data that is required for the program's steady state. EEClass stores cold data that is usually only required by type loading, JITing or reflection. Each MethodTable points to one EEClass. How does reflection work? So we know that ensuring reflection was not a quick goal, but what does it do, what takes extra time? Well there are a number of things that take place, to illustrate this allows you to view the managed and unmanaged code of the call-stack that the reflection of the speech goes through. System.Reflection.RuntimeMethodInfo.Invoke (..) - source code link calling System.Reflection.RuntimeMethodInfo.UnsafeInvokeInternal(.) System.RuntimeMethodHandle.PerformSecurityCheck(..) - a link to calling System.GC.KeepAlive (..) System.Reflection.RuntimeMethodInfo.UnsafeInvokeInternal(..) - link calling stub System.RuntimeMethodHandle.InvokeMethod (..) stub System.RuntimeMethodHandle..) - link Even if you do not click on links and look at individual C #/cpp methods, you can intuitively say that there is a lot of code executed along the way. But for you, for example, the final method where most of the work is done, System.RuntimeMethodHandle.InvokeMethod is over 400 LOC! But it's a nice insight, but what exactly is he doing? Before you can call the field/property/method through reflection, you must receive the FieldInfo/PropertyInfo/MethodInfo handle for it using this code: Type t = typeof(Person); FieldInfo m = t.GetField(Name); As shown in the previous section, it is mild m.o, because the relevant metadata must be retrieved, parsed, etc. Interestingly, runtime helps us keep an internal cache for all fields/ properties/methods. This cache has been applied The RuntimeTypeCache class and one example of its use are the RuntimeMethodInfo class. You can see the cache in action by running code in this topic that uses enough reflection to check the runtime internally! Before you have done any discussion to get FieldInfo, code gist prints this: Type: ReflectionAir.Program Reflection Type: System.RuntimeType (BaseType: System.Reflection.TypeInfo) m_fieldInfoCache is zero, cache is not initialized yet But if you have pulled even one field, you will print the following: Type: ReflectionStressi.Program Reflection Type: System.RuntimeType (BaseType: System.Reflection.TypeInfo) RuntimeTypeCache: System.RuntimeType+RuntimeTypeCache, m_cacheComplete = True, 4 items cache [0] - Int32 TestField1 - Private [1] - System.String TestField2 - Private [2] - Int32 <TestProperty1>k__BackingField - Private [3] - System.String TestField3 - Private, Static, if ReflectionOverhead.Program looks like this: Class Program { private int TestField1; private string TestField2; private static string TestField3; private int TestProperty1 { get; set; } } This means that recurring calls to GetField or GetFields are cheaper because the runtime is only filtered to an existing list that has already been created. The same applies to GetMethod and GetProperty when you call them for the first time methodinfo or propertyinfo cache is built. Validation and error handling of the argument, but once you have received MethodInfo, there is still a lot of work to do when you call invoke. Imagine that you wrote some code like this: PropertyInfo stringLengthField = typeof(string). GetProperty(Length, BindingFlags.Instance | BindingFlags.Public); var length = stringLengthField.GetGetMethod(). Invoke(new Uri(), new object[0]); When you run this you get the following exception: System.Reflection.TargetException: The object does not match the target type. at System.Reflection.RuntimeMethodInfo.CheckConsistency (..) at System.Reflection.RuntimeMethodInfo.InvokeArgumentsCheck (..) at System.Reflection.RuntimeMethodInfo.Invoke(.) at System.Reflection.RuntimePropertyInfo.GetValue(..) This is because we have received PropertyInfo length property string class, but relied on this Uri object, which is clearly the wrong type! In addition, there must also be validation of all the arguments you rely on the method you are relying on. To make the argument run by the work, the reflection APIs take a parameter that is an array object, one for each argument. So if you are using the reflection call method Add(int x, int y), you would have to rely on calling methodInfo.Invoke (.., new [] { 5, 6 }). The amount and types of values passed on must be checked at the runtime, in this case to ensure that there are 2 and both are inti. One downside of this work is that it often involves boxing, which has additional costs, but hopefully it will be minimised </TestProperty1>Security check The second main task on the way is to have several security checks. For example, it turns out that you are not allowed to use reflection to call just every method you want. There are some limited or dangerous methods that can only be called trusted .NET framework code. In addition to the black list, there are also dynamic security checks that depend on the current code access security rights that must be verified during the call. How much does reflection cost? Now that we know what the reflection is doing behind the scenes, it's a good time to see what it's going to cost us. Note that these criteria are compared to reading/writing an asset directly v through reflection. .NET properties are actually a few Get/Set methods that the compiler creates for us, but if the property is just backing out of the .NET JIT inlines method requires performance reasons. This means that the use of reflection to access the property indicates reflection in a worse possible light, but it was chosen as the most common case of use that appears in ormi, Json serialization/deserialization libraries, and object mapping tools. Below are the raw results as shown in BenchmarkDotNet, followed by the same results displayed in 2 separate tables. (full reference code available) Reading a Property ('Get') Method Mean StdErr Scaled Bytes Allocated/Op GetViaProperty 0.2159 ns 0.0047 ns 1.00 0.00 GetViaDelegate 1.8903 ns 0.0082 ns 8.82 0.00 GetViaILEmit 2.9236 ns 0.0067 ns 13.64 0.00 GetViaCompiledExpressionTrees 12.3623 ns 0.0200 ns 57.65 0.00 GetViaFastMember 35.9199 ns 0.0528 ns 167.52 0.00 GetViaReflectionWithCaching 125.3878 ns 0.2017 ns 584.78 0.00 GetViaReflection 197.9258 ns 0.2704 ns 923.08 0.01 GetViaDelegateDynamicInvoke 842.9131 ns 1.2649 ns 3,931.17 419.04 Writing a Property ('Set') Method Mean StdErr Scaled Bytes Allocated/Op SetViaProperty 1.4043 ns 0.0200 ns 6.55 0.00 SetViaDelegate 2.8215 ns 0.0078 ns 13.16 0.00 SetViaILEmit 2.8226 ns 0.0061 ns 13.16 0.00 SetViaCompiledExpressionTrees 10.7329 ns 0.0221 ns 50.06 0.00 SetViaFastMember 36.6210 ns 0.0393 ns 170.79 0.00 SetViaReflectionWithCaching 214.4321 ns 0.3122 ns 1,000.07 98.49 SetViaReflection 287.1039 ns 0.3288 ns 1,338.99 115.63 SetViaDelegateDynamicInvoke 922.4618 ns 2.9192 ns 4,302.17 390.99 So we can clearly see that regular reflection code (GetViaReflection and SetViaReflection) is considerably slower than accessing the property directly (GetViaProperty and SetViaProperty). But what happens to other results allows you to examine them in more detail. Setup First, we start with a TestClass that looks like this: public class TestClass { public TestClass (String data) { Data = data; } private string data { send { return data; } set { data = value; } } } } } } common code that all options can use: // Setup code, do only once TestClass testClass = new TestClass(A String); Type @class = BindingFlags = BindingFlags.Instance | BindingFlags.NonPublic | BindingFlags.Public; Regular analysis First, we use a regular reference code that acts as a starting point and in the worst case: [Benchmark] public string GetViaReflection() { PropertyInfo property = @class. GetProperty(Data, bindingFlags); returns the (string) attribute. GetValue(testClass, zero); Next, we get a low speed increase by keeping a reference to PropertyInfo instead of pulling it every time. But we are still much slower than accessing the property directly, which indicates that there is a significant cost of calling part of the reflection. Setup code, do only once propertyinfo cachedPropertyInfo = @class. GetProperty(Data, bindingFlags); [Benchmark] Public String GetViaReflection() { return(string)cachedPropertyInfo.GetValue(testClass, zero); } Option 2 – Use FastMember Here we use Marc Gravell's excellent Fast Member library, which as you can see is very easy to use! Setup code, make only once typeaccessor accessor = TypeAccessor.Create(@class, allowNonPublicAccessors: true); [Benchmark] Public String GetViaFastMember() { return(string)accessor[testClass, Data]; } Note that it makes something a little different than other options. This creates a TypeAccessor that allows access to all

properties of type, not just one. But the downside is that as a result, it takes longer to run. This is because internally you first receive the delegate attribute you requested (in this case, Data) before pulling this value. However, this overhead is pretty small, FastMember is still as fast as Reflection and it's very easy to use, so I suggest you take a look at it first. This option and all the following convert the reflection code to the delegate, which can be directly invoked without overhead reflection each time, thus increasing the speed! Although it is worth noting that creating a delegate is a cost (see Read more for more information). So in short, the speed increase is because we do expensive work once (security check, etc.) and storage a heavily printed delegate that we can use time and time again for a little overhead. You wouldn't use these techniques if you did reflect once, but if you do it only if it wasn't a performance bottleneck, so you wouldn't care if it was slow! The reason why the reading attribute through the delegate is not as fast as reading directly is because the .NET JIT does not text the delegate method call as it does access the attribute. So with the delegate, we have to pay the cost of the method speech, which is not possible through direct access. Option 3 – Create a delegate For this option, we use the CreateDelegate function to make our PropertyInfo a regular delegate: / / Setup code, done only if the PropertyInfo attribute = @class. GetProperty(Data, bindingFlags); Func<TestClass, string=> getDelegate =<TestClass, string=>(Func)Delegate.CreateDelegate(</TestClass,> </TestClass,>string>), property. GetGetMethod(nonPublic: true)); [Benchmark] Public String GetViaDelegate() { return getDelegate(testClass); } The drawback is that you need to know the specific type of compile-time, ie Func<TestClass, string=> part of the code above (no you can not use Func<object, string=>, if you throw it an exception!). In most situations, when you make a reflection you don't have that luxury, otherwise you don't use reflection in the first place, so it's not a complete solution. For a very interesting/mind-bending way to get around it, see MagicMethodHelper code for a fantastic blog post by Jon Skeet Making Reflection fly and explore delegates or read more about Options 4 or 5 below. Option 4 - Compiled Expression Trees Here we create a delegate, but the difference is that we do not move the object, so we can circle the limit option 3. We use the .NET expression tree API, which allows dynamic code creation: // Setup code, to perform only once the PropertyInfo attribute = @class. GetProperty(Data, bindingFlags); ParameterExpression = Expression.Parameter(typeof(object), instance; UnaryExpression instanceCast = ! attribute. DeclaringType.IsValueType ? Expression.TypeAs(instance, attribute. DeclaringType) : Expression.Convert(instance, attribute. declaration type); Func<object, object=> GetDelegate = Expression.Lambda <> <object, object=> (Expression.TypeAs(Expression.Call(instanceCast, attribute. GetGetMethod(nonPublic: true)), typeof(object)), instance). Make up(); [Benchmark] Public String GetViaCompiledExpressionTrees() { return(string)GetDelegate(testClass); } The full code expression based approach is available in a blog post faster reflection using Expression Trees Option 5 – Dynamic code-gen IL Emit Finally we reach the lowest level of approach, emit raw IL, albeit with high power, must be a great responsibility: / / Setup code, do only once propertyinfo property = @class. GetProperty(Data, bindingFlags); Sigil.Emit getterEmiter = Beam <> <object, string=>> . NewDynamicMethod(GetTestClassDataProperty) . LoadArgument(0) . CastClass(@class) . Call (property. GetGetMethod(nonPublic: true)) . (); Func<object, string=> getter = getterEmiter.CreateDelegate(); [Benchmark] public string GetViaILEmit() { back getter (testClass); } Using Expression tress (as shown in Option 4) does not give you as much flexibility as direct allocation of IL codes, although this does not prevent you from giving out invalid code! Because that if you ever find yourself in need emil IL I really recommend using the excellent Sigil library because it gives better error alerts if you get things wrong! The conclusion of Take-away is that if (and only if) you find yourself in a performance issue, if you use reflection, there are several different ways you can get it faster. These speed increases are achieved by getting a delegate that allows you to access the Property/ Field / method directly without any overheads</object,> </object,> </object,> </object,> </TestClass,> </TestClass,> every time by reflection. Discuss this post /r/programming and /r/csharp re-reading Reference, below is the call stack or code flow that run time goes through the delegate createDelegate(type, MethodInfo method) delegate CreateDelegate (type type, MethodInfo method, bool throwOndBinFailure) delegate CreateDeleIntergatenal (RuntimeType rtType, RuntimeMethodInfo rtMethod, object firstArgument, DelegateBindingFlags flags, ref StackCrawlMark stackMark) delegate UnsafeCreateDelegate (RuntimeType rtType, RuntimeMethodInfo rtMethod, object firstArgument, DelegateBindingFlags flags) bool BindToMethodInfo (object target, IRuntimeMethodMethodInfo method, RuntimeType methodType, DelegateBindFlainggs flags); FCIMPL5(FC_BOOL_RET, COMDelegate::BindToMethodInfo, object * refThisUNSAFE, object * targetUNSAFE, ReflectMethodObject *pMethodUNSAFE, ReflectClassBaseObject *pMethodTypeUNSAFE, int flags) COMDelegate::BindToMethod(DELEGATEREF *pRefThis, OBJECTREF *pRefFirstArg, MethodDesc *pTargetMethod, MethodTable *pExactMethodType, BOOLFSecurityCheck) fCheck) fCheck)